ArcadeDB Manual

Version 26.11.1

Table of Contents

Documentation . . . 1
1. Welcome . . . 2
1.1. ArcadeDB Documentation . . . 2
1.1.1. Tutorials . . . 2
1.1.2. ArcadeDB Academy . . . 2
1.1.3. Use Cases . . . 2
1.1.4. Community & Resources . . . 2
2. Get Started . . . 3
2.1. Get Started with ArcadeDB . . . 3
2.1.1. Choose your starting point . . . 3
2.1.2. Connect from your application . . . 3
2.1.3. Pick a query language . . . 3
2.1.4. Next steps . . . 3
2.2. What is ArcadeDB? . . . 4
2.2.1. TL;DR . . . 4
2.2.2. The problem with running many databases . . . 4
2.2.3. One engine, every model . . . 5
2.2.4. The graph model is our core — and it's native . . . 6
2.2.5. The other models you get . . . 6
2.2.6. Pick the query language your team already knows . . . 7
2.2.7. Embedded or server, your choice . . . 8
2.2.8. Why it's so fast . . . 8
2.2.9. Apache 2.0, free for commercial use . . . 8
2.2.10. Where to next . . . 8
2.3. Run ArcadeDB . . . 9
2.3.1. Embedded . . . 9
2.3.2. Client-Server . . . 9
2.3.3. High Availability (HA) . . . 9
2.3.4. Getting Started . . . 9
2.4. Running ArcadeDB with Docker . . . 10
2.4.1. Quick start with the OpenBeer database . . . 11
2.4.2. Persistence . . . 13
2.4.3. Tuning . . . 16
3. Examples . . . 18
3.1. Vector Search Tutorial . . . 18
3.1.1. What You Will Build . . . 18
3.1.2. Prerequisites . . . 18
3.1.3. Step 1: Create the Schema . . . 18

3.1.4. Step 2: Create a Vector Index . . . 18
3.1.5. Step 3: Insert Data with Embeddings . . . 18
3.1.6. Step 4: Query by Similarity . . . 19
3.1.7. Step 5: Add Graph Relationships . . . 19
3.1.8. Step 6: Combine Vector Search with Graph Traversal . . . 20
3.1.9. Step 7: Use Different Similarity Metrics . . . 20
3.1.10. Step 8: Control Search Quality with efSearch . . . 21
3.1.11. Step 9: Enable Quantization for Large Datasets . . . 21
3.1.12. Next Steps . . . 21
3.2. Time Series Tutorial . . . 21
3.2.1. Prerequisites . . . 22
3.2.2. Step 1: Create a TimeSeries Type . . . 22
3.2.3. Step 2: Ingest Data via InfluxDB Line Protocol . . . 22
3.2.4. Step 3: Ingest Data via SQL . . . 23
3.2.5. Step 4: Query with Time Range Filters . . . 23
3.2.6. Step 5: Aggregate with Time Bucketing . . . 23
3.2.7. Step 6: Calculate Rates and Percentiles . . . 23
3.2.8. Step 7: Fill Gaps with Interpolation . . . 24
3.2.9. Step 8: Query with PromQL . . . 24
3.2.10. Step 9: Create a Continuous Aggregate . . . 25
3.2.11. Step 10: Set Up Retention and Downsampling . . . 25
3.2.12. Next Steps . . . 25
3.3. Recommendation Engine . . . 25
3.3.1. Architecture Overview . . . 26
3.3.2. Key Queries . . . 26
3.3.3. Try It Yourself . . . 27
3.3.4. Related Documentation . . . 27
3.4. Knowledge Graphs . . . 27
3.4.1. Architecture Overview . . . 27
3.4.2. Key Queries . . . 27
3.4.3. Try It Yourself . . . 28
3.4.4. Related Documentation . . . 28
3.5. Graph RAG . . . 29
3.5.1. Architecture Overview . . . 29
3.5.2. Key Queries . . . 29
3.5.3. Try It Yourself . . . 30
3.5.4. Related Documentation . . . 30
3.6. Fraud Detection . . . 30
3.6.1. Architecture Overview . . . 30
3.6.2. Key Queries . . . 31
3.6.3. Try It Yourself . . . 31

3.6.4. Related Documentation . . . 31
3.7. Realtime Analytics . . . 32
3.7.1. Architecture Overview . . . 32
3.7.2. Key Queries . . . 32
3.7.3. Try It Yourself . . . 32
3.7.4. Related Documentation . . . 33
3.8. Social Network Analytics . . . 33
3.8.1. Architecture Overview . . . 33
3.8.2. Key Queries . . . 33
3.8.3. Try It Yourself . . . 34
3.8.4. Related Documentation . . . 34
3.9. Supply Chain . . . 34
3.9.1. Architecture Overview . . . 34
3.9.2. Key Queries . . . 35
3.9.3. Try It Yourself . . . 35
3.9.4. Related Documentation . . . 35
3.10. Identity & Access Management . . . 36
3.10.1. Architecture Overview . . . 36
3.10.2. Key Queries . . . 36
3.10.3. Try It Yourself . . . 37
3.10.4. Related Documentation . . . 37
3.11. Customer 360 . . . 37
3.11.1. Architecture Overview . . . 37
3.11.2. Key Queries . . . 37
3.11.3. Try It Yourself . . . 38
3.11.4. Related Documentation . . . 39
3.12. Semantic Search (Planned) . . . 39
3.12.1. Planned Features . . . 39
3.12.2. Reference Hybrid Query . . . 39
3.12.3. Target Scenarios . . . 40
3.13. Geospatial Analytics (Planned) . . . 40
3.13.1. Planned Features . . . 40
3.13.2. Target Scenarios . . . 40
3.14. Content Management (Planned) . . . 40
3.14.1. Planned Features . . . 40
3.14.2. Target Scenarios . . . 41
3.15. Network Monitoring (Planned) . . . 41
3.15.1. Planned Features . . . 41
3.15.2. Target Scenarios . . . 41
3.16. Data Lineage (Planned) . . . 41
3.16.1. Planned Features . . . 42

3.16.2. Target Scenarios . . . 42
4. Languages & Drivers . . . 43
4.1. Languages & Drivers . . . 43
4.2. ArcadeDB Java Tutorial . . . 43
4.2.1. Embedded . . . 43
4.2.2. Remote . . . 53
4.3. Database API . . . 54
4.3.1. `DatabaseFactory` Class . . . 54
4.3.2. close() . . . 54
4.3.3. exists() . . . 54
4.3.4. Database create() . . . 54
4.3.5. Database open() . . . 55
4.3.6. Database open(MODE mode) . . . 55
4.3.7. `Database` Interface . . . 55
4.3.8. Methods . . . 55
4.4. `DatabaseAsyncExecutor` Interface . . . 74
4.5. Java Schema API . . . 85
4.5.1. Creating many types at once . . . 86
4.5.2. Working with buckets . . . 86
4.5.3. Working with indexes . . . 86
4.5.4. Indexing Edges . . . 88
4.5.5. Database Configuration . . . 89
4.5.6. Schema Changes in a High Availability Cluster . . . 89
4.6. Native Select API . . . 89
4.6.1. Quick Start . . . 90
4.6.2. Building Queries . . . 90
4.6.3. Terminal Operations . . . 93
4.6.4. Pagination and Ordering . . . 94
4.6.5. Compiled Queries . . . 95
4.6.6. Parallel Execution . . . 96
4.6.7. Timeout . . . 97
4.6.8. Index Detection . . . 97
4.6.9. Graph Traversal . . . 98
4.6.10. Vector k-NN Search . . . 100
4.6.11. Complete Example . . . 101
4.7. Embedded Documents . . . 102
4.8. Events . . . 104
4.9. Graph Batch . . . 106
4.9.1. Quick Start . . . 106
4.9.2. How It Works . . . 107
4.9.3. Edges with Properties . . . 107

4.9.4. Builder Options . . . 108
4.9.5. Auto-Tuning Batch Size . . . 109
4.9.6. Key Methods . . . 109
4.9.7. Transactions . . . 110
4.9.8. One Batch at a Time . . . 110
4.9.9. Recovering an Interrupted Batch . . . 111
4.9.10. Performance . . . 111
4.9.11. Importing into an Existing Graph . . . 111
4.9.12. HTTP Batch Endpoint . . . 111
4.9.13. Relationship with GraphImporter . . . 112
4.10. Working with Vector Embeddings . . . 112
4.10.1. Key Features . . . 112
4.10.2. Creating Vector Indexes with SQL . . . 112
4.10.3. Memory and Performance Tuning . . . 119
4.10.4. Vector Quantization . . . 126
4.10.5. Querying Vector Indexes with SQL . . . 133
4.10.6. Using the Java API . . . 135
4.10.7. Migration from HnswVectorIndexRAM . . . 138
4.10.8. Best Practices . . . 140
4.10.9. Common Use Cases . . . 140
4.11. Java API (Remote) . . . 141
4.11.1. RemoteServer Class . . . 141
4.11.2. RemoteDatabase Class . . . 143
4.11.3. 10-Minute Tutorial . . . 143
4.11.4. RemoteImmutableVertex and RemoteMutableVertex . . . 144
4.12. Java API (gRPC) . . . 149
4.12.1. Why Use gRPC for ArcadeDB? . . . 149
4.12.2. RemoteGrpcServer Class . . . 150
4.12.3. RemoteGrpcDatabase Class . . . 150
4.12.4. Maven Dependencies . . . 150
4.12.5. 10-Minute Tutorial . . . 151
4.12.6. Transaction Management . . . 159
4.12.7. Error Handling . . . 160
4.12.8. Performance Tips . . . 161
4.12.9. Complete Example . . . 161
4.12.10. See Also . . . 163
4.13. JDBC Driver . . . 163
4.14. Neo4j BOLT Protocol . . . 164
4.14.1. Choosing a protocol for low latency . . . 164
4.14.2. Setup . . . 165
4.14.3. Configuration . . . 165

4.14.4. Access Control . . . 166
4.14.5. TLS/SSL . . . 167
4.14.6. Server Identity and Feature Envelope . . . 167
4.14.7. Compatible Drivers . . . 168
4.14.8. Connection URI Schemes . . . 169
4.14.9. Java Example . . . 169
4.14.10. Python Example . . . 170
4.14.11. JavaScript Example . . . 170
4.14.12. Go Example . . . 171
4.14.13. Cypher Query Examples . . . 172
4.14.14. Transactions . . . 173
4.14.15. Listing and terminating running statements . . . 175
4.14.16. Causal Consistency (Bookmarks) . . . 175
4.14.17. Data Type Mapping . . . 175
4.14.18. Error Codes . . . 176
4.14.19. Routing and High Availability . . . 178
4.14.20. Current Limitations . . . 178
4.14.21. Troubleshooting . . . 179
4.14.22. See Also . . . 179
4.14.23. Numeric ids . . . 179
4.15. Native Drivers . . . 180
4.15.1. HTTP or gRPC? . . . 180
4.15.2. The result envelope, and why `truncated` matters . . . 181
4.15.3. Versions and server compatibility . . . 181
4.15.4. Upgrading to 0.4.0 . . . 181
4.15.5. Upgrading to 0.3.0 . . . 182
4.16. Python Embedded Bindings . . . 182
4.16.1. Install . . . 182
4.16.2. Open a database and query it . . . 183
4.16.3. Pick the embedded bindings or the HTTP driver . . . 184
4.16.4. Where to go next . . . 184
4.17. Python HTTP Driver . . . 184
4.17.1. Install . . . 184
4.17.2. Connect and query . . . 185
4.17.3. Sync and async . . . 186
4.17.4. Close your client . . . 186
4.17.5. Transactions . . . 186
4.17.6. Two error models . . . 187
4.17.7. Labelling work with a query tag . . . 188
4.17.8. Duplex insert session . . . 188
4.17.9. Administration . . . 189

4.17.10. Sessions and API tokens . . . 189
4.17.11. Running work . . . 190
4.17.12. Choosing a protocol from Python . . . 190
4.17.13. Next steps . . . 191
4.18. Python gRPC Driver . . . 192
4.18.1. Install . . . 192
4.18.2. Connect and query . . . 192
4.18.3. `raw`, `raw_admin` and the wrappers . . . 193
4.18.4. Authentication is attached to the channel . . . 195
4.18.5. Streaming queries . . . 196
4.18.6. Streaming inserts . . . 196
4.18.7. Transactions . . . 197
4.18.8. Errors . . . 198
4.18.9. Async . . . 199
4.18.10. Next steps . . . 200
4.19. ArcadeDB Python Quickstart . . . 200
4.19.1. Prerequisites . . . 201
4.19.2. Connect to ArcadeDB . . . 201
4.19.3. Create a Schema . . . 201
4.19.4. Query with SQL . . . 202
4.19.5. Query with Cypher . . . 202
4.19.6. Vector Search . . . 202
4.19.7. Full Example . . . 203
4.19.8. HTTP / JSON API . . . 204
4.19.9. Embedded Python Bindings . . . 204
4.19.10. Next Steps . . . 204
4.20. JavaScript / TypeScript HTTP Driver . . . 204
4.20.1. Install . . . 205
4.20.2. Connect and query . . . 205
4.20.3. The result envelope . . . 206
4.20.4. Transactions . . . 207
4.20.5. Two error models . . . 207
4.20.6. Labelling work with a query tag . . . 208
4.20.7. Duplex insert session . . . 209
4.20.8. Administration . . . 209
4.20.9. Sessions and API tokens . . . 209
4.20.10. Running work . . . 210
4.20.11. Next steps . . . 211
4.21. JavaScript / TypeScript gRPC Driver . . . 211
4.21.1. Install . . . 211
4.21.2. Runtime targets: Node, Bun, Deno — not browsers . . . 211

4.21.3. Connect and query . . . 212
4.21.4. `record.properties` and the `GrpcValue` wrapper . . . 213
4.21.5. Authentication and the insecure guard . . . 214
4.21.6. Streaming queries and `retrievalMode` . . . 214
4.21.7. Streaming inserts . . . 216
4.21.8. Transactions . . . 216
4.21.9. Errors . . . 217
4.21.10. Next steps . . . 218
4.22. JavaScript / TypeScript . . . 219
4.22.1. Prerequisites . . . 219
4.22.2. Connect to ArcadeDB . . . 219
4.22.3. Create a Schema . . . 220
4.22.4. Query with SQL . . . 220
4.22.5. Query with Cypher . . . 220
4.22.6. Vector Search . . . 221
4.22.7. Full Example . . . 221
4.22.8. Next Steps . . . 222
4.23. HTTP / JSON from JavaScript . . . 222
4.23.1. Using fetch . . . 223
4.23.2. Using axios . . . 224
4.23.3. TypeScript . . . 224
4.23.4. Further Reading . . . 225
4.24. Go HTTP Driver . . . 225
4.24.1. Install . . . 225
4.24.2. Connect and query . . . 225
4.24.3. Streaming queries . . . 227
4.24.4. Transactions . . . 228
4.24.5. Two error models . . . 229
4.24.6. Beyond queries . . . 230
4.24.7. Labelling work with a query tag . . . 230
4.24.8. Duplex insert session . . . 231
4.24.9. Administration . . . 231
4.24.10. Sessions and API tokens . . . 232
4.24.11. Running work . . . 232
4.24.12. Next steps . . . 233
4.25. Go gRPC Driver . . . 233
4.25.1. Install . . . 233
4.25.2. Connect and query . . . 233
4.25.3. `Raw()` and the facade . . . 235
4.25.4. Authentication is attached to the connection . . . 236
4.25.5. Streaming queries . . . 236

4.25.6. Streaming inserts . . . 237
4.25.7. Transactions . . . 238
4.25.8. Errors . . . 239
4.25.9. Administration: `RawAdmin` . . . 240
4.25.10. Next steps . . . 240
4.26. ArcadeDB Go Quickstart . . . 241
4.26.1. Prerequisites . . . 241
4.26.2. Connect to ArcadeDB . . . 241
4.26.3. Create a Schema . . . 242
4.26.4. Query with SQL . . . 243
4.26.5. Query with Cypher . . . 243
4.26.6. Vector Search . . . 244
4.26.7. Full Example . . . 245
4.26.8. Next Steps . . . 246
4.27. Connecting from C# . . . 246
4.27.1. Sample Code . . . 247
4.28. Connecting from C . . . 247
4.28.1. PostgreSQL Wire Protocol — `libpq` . . . 248
4.28.2. HTTP/JSON API — `libcurl` . . . 248
4.29. C++ . . . 249
4.29.1. PostgreSQL Wire Protocol — `libpqxx` . . . 249
4.29.2. HTTP/JSON API — `cpp-httplib` . . . 250
4.30. Connecting from Rust . . . 250
4.30.1. Neo4j BOLT — `neo4rs` . . . 250
4.30.2. PostgreSQL Wire Protocol — `postgres` . . . 251
4.30.3. HTTP/JSON API — `reqwest` . . . 251
4.31. Connecting from Ruby . . . 252
4.31.1. Neo4j BOLT — `neo4j-ruby-driver` . . . 252
4.31.2. PostgreSQL Wire Protocol — `pg` . . . 253
4.31.3. HTTP/JSON API — `Net::HTTP` . . . 253
4.32. Connecting from R . . . 254
4.32.1. PostgreSQL Wire Protocol — `RPostgres` . . . 254
4.32.2. HTTP/JSON API — `httr2` . . . 254
4.33. Connecting from Elixir . . . 255
4.33.1. Configuration . . . 255
4.33.2. Sample Code . . . 255
5. API & Integration . . . 257
5.1. API & Integration . . . 257
5.2. HTTP/JSON API . . . 257
5.2.1. Introduction . . . 258
5.2.2. Token-Based Authentication . . . 258

5.2.3. Server-Side Transactions . . . 260
5.2.4. Transaction Scripts . . . 261
5.2.5. Streaming Change Events . . . 261
5.2.6. Streaming Inserts over WebSocket . . . 263
5.2.7. Responses . . . 266
5.2.8. Retrying Requests Safely (`X-Request-Id`) . . . 266
5.2.9. Tutorial . . . 267
5.2.10. Reference . . . 268
5.2.11. Numbers and arrays in `params` . . . 298
5.3. gRPC API . . . 314
5.3.1. Enabling the gRPC server . . . 314
5.3.2. Services . . . 315
5.3.3. Cancellation and running statements . . . 325
5.3.4. Metrics . . . 326
5.3.5. Message Types . . . 327
5.4. Bolt Driver Compatibility Matrix . . . 332
5.4.1. Certified Feature Areas . . . 333
5.4.2. Error Codes . . . 334
5.4.3. Documented Limitations . . . 335
5.4.4. Certified Drivers and Versions . . . 335
5.4.5. See Also . . . 335
5.5. Postgres Protocol Plugin . . . 335
5.5.1. Other query languages . . . 337
5.5.2. TLS/SSL . . . 338
5.5.3. Current limitations . . . 338
5.5.4. System catalog and schema introspection . . . 340
5.5.5. COPY … TO STDOUT . . . 344
5.5.6. Column type mapping . . . 345
5.5.7. Transactions . . . 346
5.5.8. Session settings (SET, SHOW, RESET) . . . 348
5.5.9. Cancelling a statement . . . 350
5.5.10. Creating an object whose name is taken . . . 351
5.5.11. Naming an object that does not exist . . . 351
5.5.12. A refused value and an unknown function . . . 351
5.5.13. Schema changes and command tags . . . 352
5.5.14. Generated keys (RETURNING) . . . 352
5.5.15. Postgres Tools Known to Work . . . 352
5.6. MCP Server . . . 353
5.6.1. Endpoint . . . 354
5.6.2. Configuration . . . 354
5.6.3. Tool Profiles . . . 355

5.6.4. Per-Database Scoping . . . 357
5.6.5. Configuration Endpoint . . . 358
5.6.6. MCP Protocol Methods . . . 358
5.6.7. Available Tools . . . 359
5.6.8. Resources . . . 377
5.6.9. Prompts . . . 379
5.6.10. Transport . . . 381
5.6.11. Listing and terminating tool calls . . . 383
5.6.12. Security . . . 384
5.6.13. Using with AI Clients . . . 384
6. Data Modeling . . . 386
6.1. Multi-Model Architecture . . . 386
6.1.1. Why Multi-Model Matters . . . 386
6.1.2. The Latency Problem with Multiple Databases . . . 387
6.1.3. Supported Data Models . . . 387
6.1.4. Unified Query Layer . . . 390
6.1.5. Type Inheritance Across Models . . . 390
6.1.6. Schema Flexibility . . . 391
6.1.7. Cross-Model Queries . . . 391
6.2. Graph Database . . . 391
6.2.1. Graph Components . . . 391
6.2.2. Graph Database Types . . . 392
6.2.3. Why (Property) Graph Databases? . . . 392
6.2.4. See Also . . . 392
6.3. Time Series . . . 392
6.3.1. Creating a TimeSeries Type . . . 393
6.3.2. Altering a TimeSeries Type . . . 395
6.3.3. Compacting a TimeSeries Type . . . 397
6.3.4. Dropping a TimeSeries Type . . . 398
6.3.5. Deleting Samples . . . 398
6.3.6. Ingesting Data . . . 400
6.3.7. Querying Time Series Data . . . 408
6.3.8. Moving the bucket grid . . . 410
6.3.9. Query Languages . . . 416
6.3.10. PromQL Query Language . . . 417
6.3.11. Continuous Aggregates . . . 422
6.3.12. Retention Policies . . . 425
6.3.13. Downsampling Policies . . . 425
6.3.14. Grafana Integration . . . 426
6.3.15. Studio TimeSeries Explorer . . . 427
6.3.16. HTTP API Reference . . . 427

6.3.17. High Availability . . . 428
6.3.18. Integrity Checking . . . 429
6.3.19. Comparison with Other TimeSeries Databases . . . 431
6.3.20. Choosing the layout: `COMPACTION_INTERVAL` and `SHARDS` . . . 432
6.3.21. Architecture . . . 433
6.4. Vector Search . . . 434
6.4.1. How Vector Search Works . . . 434
6.4.2. LSMVectorIndex Architecture . . . 434
6.4.3. Flat vs Hierarchical Structure . . . 434
6.4.4. Similarity Functions . . . 435
6.4.5. Quantization . . . 435
6.4.6. Vector Encoding (Pre-Quantized Ingest) . . . 438
6.4.7. Store Embeddings in an `EXTERNAL` Property . . . 439
6.4.8. Key Parameters . . . 440
6.4.9. efSearch and the Default Beam . . . 446
6.4.10. Filtered Search . . . 447
6.4.11. Multi-Modal Search . . . 448
6.4.12. Integration with Other Models . . . 449
6.4.13. SQL Example . . . 449
6.4.14. Java Example . . . 449
6.4.15. Configuration Parameters . . . 450
6.4.16. Supported Similarity Functions . . . 450
6.4.17. Sparse Vector Search . . . 450
6.4.18. Hybrid Search with `vector.fuse` . . . 454
6.4.19. Group-By Retrieval . . . 456
6.4.20. Further Reading . . . 457
6.5. Geospatial Index . . . 458
6.5.1. Creating a Geospatial Index . . . 458
6.5.2. Inserting Geometries . . . 458
6.5.3. Querying with Spatial Predicates . . . 459
6.5.4. How Index-Accelerated Queries Work . . . 459
6.5.5. geo.* Functions Reference . . . 460
6.5.6. Practical Examples . . . 466
6.5.7. Notes and Limitations . . . 467
6.6. Full-Text Index and BM25 Ranking . . . 468
6.6.1. Creating a Full-Text Index . . . 469
6.6.2. Configuring the Analyzer . . . 469
6.6.3. Searching . . . 471
6.6.4. Lucene Query Syntax . . . 473
6.6.5. Relevance Scoring . . . 475
6.6.6. More Like This . . . 479

6.6.7. Practical Examples . . . 482
6.6.8. Maintenance: BM25 Corpus Statistics . . . 484
6.6.9. Behavior Changes When Upgrading . . . 485
6.6.10. Getting BM25 on Existing Indexes . . . 486
6.6.11. BM25 Indexes on Lists and Maps Built Before v26.11.1 . . . 486
6.6.12. Notes . . . 486
6.7. Key/Value . . . 487
6.7.1. How Keys Work in ArcadeDB . . . 487
6.7.2. Key/Value with the Redis Wire Protocol . . . 487
6.7.3. Key/Value with the Java API . . . 488
6.7.4. Key/Value over HTTP . . . 489
6.7.5. Comparison with Other Models . . . 489
6.7.6. Why ArcadeDB for Key/Value . . . 489
6.7.7. Further Reading . . . 490
6.8. Record . . . 490
6.8.1. Document . . . 490
6.8.2. Vertex . . . 491
6.8.3. Edge . . . 491
6.8.4. Record ID . . . 491
6.8.5. Types . . . 492
6.8.6. Buckets . . . 492
6.8.7. Types vs. Buckets in Queries . . . 493
6.8.8. One bucket per period . . . 493
6.8.9. One bucket per location . . . 493
6.8.10. Bucket Selection Strategy . . . 494
6.8.11. Relationships . . . 498
6.8.12. Graph Connections . . . 498
6.8.13. Referenced Relationships . . . 498
6.8.14. Embedded Relationships . . . 499
6.8.15. 1:1 and n:1 Embedded Relationships . . . 499
6.8.16. 1:n and n:n Embedded Relationships . . . 499
6.8.17. Inverse Relationships . . . 499
6.8.18. Edge Constraints . . . 500
6.8.19. Relationship Traversal Complexity . . . 500
6.9. Database . . . 500
6.9.1. Database URL . . . 501
6.9.2. Database lifecycle . . . 501
6.9.3. Creating Databases . . . 501
6.10. Schema . . . 503
6.10.1. Types . . . 503
6.10.2. Properties . . . 503

6.10.3. Buckets . . . 506
6.10.4. Indexes . . . 507
6.10.5. Triggers . . . 507
6.10.6. Materialized Views . . . 507
6.11. Inheritance . . . 507
6.12. Indexes . . . 509
6.12.1. Why (and when) to add an index . . . 509
6.12.2. Index Types at a Glance . . . 509
6.12.3. LSM Tree (default) . . . 510
6.12.4. Hash Index . . . 512
6.12.5. Dense Vector Index (LSM_VECTOR) . . . 514
6.12.6. Sparse Vector Index (LSM_SPARSE_VECTOR) . . . 514
6.12.7. Full-Text Index . . . 515
6.12.8. Geospatial Index . . . 515
6.12.9. Case-Insensitive Indexes (COLLATE CI) . . . 515
6.12.10. Building a large index over existing data (sorted bulk build) . . . 517
6.12.11. What can be indexed . . . 518
6.12.12. Looking up a value the property cannot hold . . . 519
6.12.13. Which index a SQL query uses when several apply . . . 520
6.12.14. See also . . . 520
6.13. Transactions . . . 520
6.13.1. ACID Properties . . . 521
6.13.2. Atomicity . . . 521
6.13.3. Consistency . . . 521
6.13.4. Isolation . . . 522
6.13.5. Durability . . . 526
6.13.6. Optimistic Transaction . . . 531
6.13.7. A statement that fails halfway . . . 531
6.13.8. Nested transactions and propagation . . . 532
6.14. Data Types . . . 532
6.15. Binary Types (BLOB) . . . 537
6.16. Managing Dates . . . 538
6.17. Materialized Views . . . 550
6.17.1. Refresh Modes . . . 551
6.17.2. View Status . . . 553
6.17.3. Querying a Materialized View . . . 553
6.17.4. Java API . . . 553
6.17.5. Behavior and Constraints . . . 555
6.17.6. Error Handling . . . 556
6.17.7. Limitations . . . 556
6.17.8. Example: Sales Dashboard . . . 557

6.18. Vector Embeddings . . . 557
6.18.1. Choosing an Embedding Model . . . 558
6.18.2. Creating a Vector Index . . . 558
6.18.3. Choosing a Similarity Function . . . 559
6.18.4. Quantization Trade-offs . . . 559
6.18.5. INT8 Pre-Quantized Ingest . . . 560
6.18.6. Tuning for Recall vs Speed . . . 561
6.18.7. Tuning efSearch . . . 561
6.18.8. Returning Only RIDs and Distances . . . 562
6.18.9. Multi-Modal Embeddings . . . 563
6.18.10. Searching a Type Hierarchy . . . 563
6.18.11. Hybrid Search: Dense + Sparse + Full-Text . . . 564
6.18.12. Batch Ingestion . . . 565
6.18.13. Searching Vectors Not Yet in the Graph . . . 566
6.18.14. Rebuild Memory and Deferred Rebuilds . . . 567
6.18.15. Global Configuration . . . 570
6.18.16. Further Reading . . . 570
7. Import & Migration . . . 571
7.1. Importer . . . 571
7.1.1. Additional Settings . . . 574
7.1.2. Transactions . . . 574
7.1.3. CSV rows and the header . . . 575
7.1.4. Column types of CSV and XML sources . . . 575
7.1.5. TimeSeries samples in a JSONL file . . . 576
7.1.6. Import report . . . 576
7.2. Graph Importer . . . 581
7.2.1. How It Works . . . 581
7.2.2. Memory and Progress . . . 581
7.2.3. Command-Line Usage . . . 582
7.2.4. Java API . . . 582
7.2.5. JSON Configuration . . . 583
7.2.6. Complete Example . . . 588
7.2.7. Identities and Missing Edges . . . 589
7.2.8. Supported File Formats . . . 590
7.2.9. Data Sources (Java API) . . . 590
7.3. JSON Importer . . . 590
7.4. OrientDB Importer . . . 595
7.4.1. What the import preserves . . . 595
7.4.2. Differences with OrientDB . . . 595
7.4.3. Architecture . . . 595
7.4.4. Data Model . . . 596

7.4.5. Geospatial Data (OPoint and other spatial types) . . . 596
7.4.6. SQL . . . 597
7.4.7. API . . . 597
7.4.8. Connectivity . . . 598
7.4.9. Migration . . . 598
7.5. Neo4j Importer . . . 598
7.5.1. Performance . . . 599
7.5.2. Multi-label handling . . . 599
7.5.3. Importing via SQL . . . 599
7.5.4. Importing via command line . . . 600
7.5.5. Memory considerations . . . 601
7.5.6. Differences with Neo4j . . . 601
7.5.7. What's compatible . . . 601
7.5.8. Where ArcadeDB goes further . . . 602
7.5.9. Performance comparison . . . 602
7.5.10. Connecting an existing Neo4j application . . . 602
7.5.11. Importing a Neo4j database . . . 603
7.5.12. Multi-label nodes . . . 604
7.5.13. Schema and constraints . . . 604
7.5.14. Embedding ArcadeDB . . . 604
7.5.15. Further reading . . . 605
8. Self-Managed Deployment . . . 606
8.1. Installations from Binaries . . . 606
8.1.1. Binaries . . . 606
8.1.2. macOS via Homebrew . . . 606
8.1.3. Manual installation on macOS . . . 607
8.1.4. Windows via Scoop . . . 607
8.1.5. Custom Package Builder . . . 608
8.1.6. Prerequisites . . . 608
8.1.7. Quick Start . . . 608
8.1.8. Available Modules . . . 608
8.1.9. Usage Examples . . . 609
8.1.10. Command-Line Options . . . 610
8.1.11. Output Files . . . 610
8.2. Installation with Docker . . . 610
8.3. Native Image (Experimental) . . . 611
8.3.1. Getting the Native Image . . . 611
8.3.2. CPU Requirements . . . 612
8.3.3. Running the Native Server . . . 613
8.3.4. What the Native Image Includes . . . 615
8.3.5. Limitations Compared to the JVM Distribution . . . 615

8.3.6. Building the Native Image Yourself . . . 616
8.4. Running ArcadeDB on Kubernetes . . . 616
8.4.1. Prerequisites . . . 616
8.4.2. Quickstart with kind . . . 617
8.4.3. Installing the chart . . . 618
8.4.4. Configuration reference . . . 618
8.4.5. Operating the cluster . . . 634
8.4.6. High Availability on Kubernetes . . . 636
8.4.7. Troubleshooting . . . 638
8.4.8. Reference deployment . . . 639
8.5. Installation on AWS (Marketplace AMI) . . . 639
8.5.1. Launch an instance . . . 640
8.5.2. Read the root password . . . 640
8.5.3. Connect . . . 640
8.5.4. Operate the server . . . 640
8.5.5. Upgrade . . . 641
8.6. Embedded Server . . . 641
8.6.1. Java 17 notes . . . 642
8.6.2. Start the server in the JVM . . . 642
8.6.3. Create custom HTTP commands . . . 644
8.6.4. HTTPS connection . . . 645
8.6.5. Replication between boxes . . . 646
8.7. Server . . . 646
8.7.1. Start server hint . . . 649
8.7.2. Server modes . . . 649
8.7.3. User changes on open connections . . . 652
8.7.4. Create default database(s) . . . 652
8.7.5. Closing idle databases . . . 654
8.7.6. Logging . . . 656
8.7.7. Server Plugins (Extend The Server) . . . 657
8.7.8. Metrics . . . 659
8.8. Changing Settings . . . 659
8.8.1. Environment Variables . . . 659
8.8.2. RAM Configuration . . . 660
8.9. High Availability . . . 661
8.9.1. Leader-Replica Replication Model . . . 662
8.9.2. Server Roles and Election . . . 662
8.9.3. Replication and the Raft Log . . . 663
8.9.4. Write Quorum . . . 663
8.9.5. Read Consistency . . . 664
8.9.6. Automatic Failover . . . 664

8.9.7. Split-Brain Protection . . . 664
8.9.8. Snapshots and Catch-Up . . . 665
8.9.9. Cluster Bootstrap from Pre-Staged Data . . . 665
8.9.10. Dynamic Membership . . . 665
8.9.11. Cluster Discovery and Configuration . . . 665
8.9.12. Continuous Correctness Testing with Jepsen . . . 666
8.10. Configuring High Availability . . . 668
8.10.1. Quick Start . . . 668
8.10.2. Server List Entry Format . . . 668
8.10.3. Client Routing Ports (bolt, grpc) . . . 670
8.10.4. Peer-to-Peer Endpoints (http, https) . . . 671
8.10.5. Named Peers . . . 672
8.10.6. Architecture . . . 673
8.10.7. Schema Replication . . . 674
8.10.8. When a follower falls back to a full schema load (from v26.10.2) . . . 675
8.10.9. Renaming a type on a cluster (from v26.10.2) . . . 676
8.10.10. Schema Delta Replication . . . 677
8.10.11. Database settings (from v26.10.2) . . . 677
8.10.12. Fixed-width float arrays (from v26.11.1) . . . 678
8.10.13. Write Quorum . . . 678
8.10.14. Read Consistency . . . 679
8.10.15. Load Shedding and Backpressure . . . 680
8.10.16. Witness / Read-Scale Nodes . . . 680
8.10.17. Automatic Failover . . . 681
8.10.18. Replication-Channel Self-Healing . . . 681
8.10.19. Durable Raft Storage . . . 683
8.10.20. Committed-remotely response contract . . . 684
8.10.21. Outcome-unknown response contract . . . 684
8.10.22. Writes during a leadership transfer . . . 684
8.10.23. Writes through a replica during its resync . . . 685
8.10.24. Creating a database in a cluster . . . 685
8.10.25. Creating or dropping a database through a replica . . . 685
8.10.26. Dropping and creating a database again under the same name . . . 686
8.10.27. Long statements forwarded to the leader . . . 686
8.10.28. When a replica refuses a write and names the leader . . . 686
8.10.29. Running statements on a cluster . . . 687
8.10.30. Offline Cluster Bootstrap . . . 688
8.10.31. When a node has stopped for good . . . 689
8.10.32. Cluster Management REST API . . . 693
8.10.33. Verify Outcomes . . . 698
8.10.34. Studio Cluster Dashboard . . . 699

8.10.35. Security . . . 699
8.10.36. TLS/SSL for peer-to-peer transfers . . . 701
8.10.37. mTLS for the Raft gRPC transport . . . 703
8.10.38. Kubernetes . . . 708
8.10.39. Troubleshooting . . . 708
8.10.40. HA Settings . . . 712
8.11. Backup of a Database . . . 717
8.11.1. Consistency of TimeSeries data . . . 717
8.11.2. Vector indexes . . . 718
8.11.3. Configuration . . . 718
8.11.4. Cloud Backups . . . 719
8.12. Automatic Backup Scheduler . . . 719
8.12.1. Features . . . 719
8.12.2. Configuration File . . . 719
8.12.3. Configuration Options . . . 720
8.12.4. CRON Expressions . . . 722
8.12.5. HA Cluster Support . . . 723
8.12.6. Backup Directory Structure . . . 723
8.12.7. Monitoring . . . 724
8.12.8. Studio Integration . . . 724
8.12.9. Triggering Manual Backups . . . 725
8.12.10. Database Lifecycle . . . 727
8.12.11. Troubleshooting . . . 727
8.13. Restore a Database . . . 728
8.13.1. Restore via HTTP API (recommended) . . . 728
8.13.2. Restore via command line . . . 728
8.13.3. Configuration . . . 729
8.14. Monitoring . . . 729
8.14.1. Enabling Metrics . . . 730
8.14.2. Prometheus Integration . . . 730
8.14.3. Available Metrics . . . 730
8.14.4. Periodic Metric Dumps . . . 733
8.14.5. Grafana Dashboard Setup . . . 734
8.14.6. Health Checks . . . 734
8.14.7. Server health warnings . . . 734
8.14.8. Further Reading . . . 734
8.15. Cloud Observability . . . 735
8.15.1. Metrics depth . . . 735
8.15.2. Distributed tracing . . . 738
8.15.3. Structured logging and correlation IDs . . . 740
8.15.4. Health probes . . . 740

8.15.5. Further reading . . . 742
8.16. Logging . . . 742
8.16.1. The default logger (java.util.logging) . . . 743
8.16.2. Per-package verbosity . . . 743
8.16.3. Log directory . . . 745
8.16.4. Console format . . . 745
8.16.5. Routing through SLF4J when embedding . . . 745
8.16.6. Custom logger implementations . . . 747
8.16.7. Further Reading . . . 747
8.17. Upgrade ArcadeDB . . . 747
8.17.1. Breaking changes . . . 747
8.17.2. Single-server upgrade . . . 800
8.17.3. What to copy . . . 801
8.17.4. HA cluster upgrade . . . 802
8.17.5. Docker / Kubernetes . . . 803
8.17.6. Downgrade ArcadeDB . . . 803
8.17.7. Downgrading a node of an HA cluster . . . 804
8.17.8. Downgrading from 26.10.2 or later with hash indexes . . . 804
8.18. Versioning, Compatibility and Deprecation Policy . . . 804
8.18.1. Version numbers . . . 804
8.18.2. On-disk compatibility . . . 805
8.18.3. Public interfaces covered . . . 805
8.18.4. Deprecation process . . . 805
8.18.5. Exceptions: security and correctness fixes . . . 806
8.18.6. Which releases receive fixes . . . 806
8.18.7. License commitment . . . 806
9. Security . . . 807
9.1. Users . . . 807
9.1.1. Via Console . . . 809
9.1.2. Via Cypher . . . 809
9.1.3. Password hash format . . . 809
9.2. Groups . . . 811
9.3. Security Policy . . . 814
9.4. Handling Secrets . . . 814
9.5. Gremlin Security . . . 816
9.5.1. Execution Engines . . . 816
9.5.2. Who Can Evaluate Groovy . . . 816
9.5.3. Groovy Scripts Are Always Treated as Writes . . . 817
9.5.4. Configuration . . . 818
9.5.5. Security Best Practices . . . 818
9.5.6. Security Limitations (Groovy Engine) . . . 818

9.5.7. Migration from Groovy to Java Engine . . . 819
9.5.8. Security Considerations . . . 819
10. Tools . . . 821
10.1. Studio . . . 821
10.1.1. Main menu . . . 821
10.1.2. Login and themes . . . 822
10.2. Query Panel . . . 822
10.2.1. Left icon sidebar . . . 823
10.2.2. Editor and toolbar . . . 824
10.2.3. Index search (Vector, Hybrid, Full-text) . . . 824
10.2.4. Result views . . . 825
10.3. Database Panel . . . 830
10.3.1. Schema . . . 830
10.3.2. Buckets . . . 831
10.3.3. Indexes . . . 831
10.3.4. Dictionary . . . 832
10.3.5. Time Series . . . 832
10.3.6. Metrics . . . 832
10.3.7. Database Settings . . . 833
10.3.8. Backup . . . 834
10.4. Time Series Sub-tab . . . 835
10.4.1. Query . . . 835
10.4.2. Schema . . . 836
10.4.3. Quick Tour . . . 836
10.5. Server Panel . . . 837
10.5.1. Summary . . . 838
10.5.2. Metrics . . . 838
10.5.3. Sessions . . . 839
10.5.4. Running Queries . . . 839
10.5.5. Events . . . 840
10.5.6. Server Settings . . . 840
10.5.7. Backup . . . 841
10.5.8. MCP . . . 841
10.6. Profiler Panel . . . 842
10.6.1. Toolbar . . . 842
10.6.2. Summary . . . 843
10.6.3. Query list . . . 843
10.6.4. Query detail . . . 843
10.6.5. AI analysis . . . 844
10.7. Security Panel . . . 845
10.7.1. Users . . . 845

10.7.2. User Groups . . . 846
10.7.3. API Tokens . . . 846
10.7.4. Group and Token Changes During a Rolling Upgrade . . . 847
10.8. Cluster Panel . . . 848
10.8.1. Overview . . . 848
10.8.2. Metrics . . . 849
10.8.3. Management . . . 849
10.9. API Panel . . . 849
10.9.1. Endpoint sidebar . . . 850
10.9.2. Detail pane . . . 850
10.9.3. Try It playground . . . 851
10.10. Info Panel . . . 852
10.10.1. Documentation sidebar . . . 853
10.10.2. Feature overview . . . 853
10.11. AI Assistant Panel . . . 853
10.11.1. Subscription . . . 854
10.11.2. Activation . . . 854
10.11.3. Using the assistant . . . 856
10.11.4. Modes . . . 856
10.11.5. Executing suggested commands . . . 856
10.11.6. Privacy and security notes . . . 857
10.12. Settings Panel . . . 857
10.12.1. Appearance . . . 857
10.13. Console . . . 858
10.13.1. Tutorial . . . 860
10.13.2. Settings . . . 863
10.13.3. Scripting . . . 863
10.13.4. Console-Server Interaction . . . 864
10.14. Swagger-UI . . . 864
10.15. Compatible Tools . . . 866
10.15.1. gdotv . . . 866
10.15.2. JetBrains DataGrip/Database Plugin . . . 868
10.15.3. DBeaver . . . 870
10.15.4. DbVisualizer . . . 875
10.15.5. DbGate . . . 880
10.15.6. LibreOffice Base . . . 882
10.15.7. adbcBridge . . . 887
11. BI & Analytics . . . 889
11.1. BI and Analytics Integration . . . 889
11.1.1. Connection paths . . . 889
11.1.2. Supported tools . . . 889

11.1.3. Setting up the Postgres endpoint . . . 890
11.1.4. Sending non-SQL queries from a Postgres-only tool . . . 890
11.1.5. Suggesting a missing tool . . . 890
11.2. Apache Superset . . . 891
11.2.1. Setup . . . 891
11.2.2. Usage . . . 891
11.2.3. Known Limitations . . . 892
11.3. Grafana . . . 892
11.3.1. ArcadeDB Grafana Plugin (Recommended) . . . 892
11.3.2. PostgreSQL Data Source (Fallback) . . . 894
11.3.3. Grafana Time Series Endpoints . . . 894
11.4. Metabase . . . 894
11.4.1. Setup . . . 894
11.4.2. Usage . . . 895
11.4.3. Known Limitations . . . 895
11.5. Power BI . . . 895
11.5.1. Setup . . . 896
11.5.2. Usage . . . 896
11.5.3. Known Limitations . . . 896
11.6. Tableau . . . 897
11.6.1. Setup . . . 897
11.6.2. Usage . . . 897
11.6.3. Known Limitations . . . 898
12. Advanced . . . 899
12.1. Performance Tuning . . . 899
12.1.1. Memory Configuration . . . 899
12.1.2. Index Selection . . . 900
12.1.3. Query Optimization . . . 900
12.1.4. Bucket Configuration . . . 901
12.1.5. Page Size Tuning . . . 902
12.1.6. Connection Pool Settings . . . 902
12.1.7. Concurrent Requests (Query Admission Gate) . . . 902
12.1.8. Write Optimization . . . 904
12.1.9. Read Optimization . . . 904
12.1.10. High Availability Resync . . . 905
12.1.11. Further Reading . . . 905
12.2. Settings . . . 905
12.2.1. JVM startup (server/embedded only) . . . 905
12.2.2. SQL (Database Level) . . . 907
12.2.3. Programmatically (Server/Embedded and Database levels) . . . 907
12.2.4. Available settings by scope (in alphabetic order): . . . 908

12.3. SQL Syntax . . . 993
12.4. Graph OLAP Engine . . . 1008
12.4.1. Why Graph OLAP? . . . 1009
12.4.2. Graph Analytical View (GAV) . . . 1009
12.4.3. SQL . . . 1009
12.4.4. Builder Options . . . 1013
12.4.5. Async Build for Large Graphs . . . 1013
12.4.6. Update Modes . . . 1014
12.4.7. CSR Persistence . . . 1016
12.4.8. How CSR Works . . . 1018
12.4.9. Columnar Property Storage . . . 1018
12.4.10. Memory Usage . . . 1019
12.4.11. Graph Algorithms . . . 1020
12.4.12. Shortest Paths with Contraction Hierarchies (CCH) . . . 1021
12.4.13. Query Planner Integration . . . 1023
12.4.14. Lifecycle . . . 1027
12.4.15. Benchmark Results . . . 1027
12.4.16. Limitations . . . 1028
12.5. Storage Internals . . . 1028
12.5.1. Pages and Record IDs . . . 1028
12.5.2. Zero Copy Objects . . . 1028
12.5.3. How Buckets Work . . . 1029
12.5.4. Bucket Partitioning . . . 1031
12.5.5. Add and Remove Buckets from Types . . . 1032
12.5.6. Page Version . . . 1033
12.5.7. Schema File Persistence . . . 1033
12.5.8. Cost of a Schema Change, and Batching Many . . . 1034
12.6. LSM-Tree Algorithm . . . 1035
12.6.1. Quick Overview . . . 1035
13. Help . . . 1038
13.1. Community . . . 1038
13.1.1. Professional Support . . . 1038
13.1.2. Frequently Asked Questions . . . 1038
13.2. Get the Latest Build . . . 1039
13.2.1. Option 1: Build it Yourself . . . 1039
13.2.2. Option 2: Wait ~15 Minutes for the Maven Central SNAPSHOT . . . 1040
13.2.3. Verifying You Have the Build You Expect . . . 1041
13.2.4. When to Use Which Option . . . 1041
13.3. Report an Issue . . . 1042
13.4. Requirements . . . 1042
13.4.1. Memory . . . 1042

13.4.2. Java . . . 1042
Query Languages . . . 1044
14. SQL . . . 1045
14.1. SQL Introduction . . . 1045
14.1.1. Introduction to SQL . . . 1045
14.1.2. Automatic usage of indexes . . . 1045
14.1.3. Extra resources . . . 1046
14.1.4. ArcadeDB SQL dialect . . . 1046
14.1.5. Learning SQL . . . 1047
14.1.6. No JOINs . . . 1048
14.1.7. Projection . . . 1048
14.1.8. System Types . . . 1049
14.1.9. DISTINCT . . . 1050
14.1.10. HAVING . . . 1051
14.1.11. Multiple targets . . . 1051
14.1.12. SQL Differences . . . 1051
14.1.13. Commands . . . 1052
14.1.14. Comments . . . 1052
14.2. SQL - SELECT . . . 1053
14.2.1. LET Block . . . 1057
14.2.2. Sub-query . . . 1058
14.2.3. Unwinding . . . 1058
14.3. SQL - INSERT . . . 1059
14.4. SQL - UPDATE . . . 1064
14.5. SQL - DELETE . . . 1070
14.6. SQL - MATCH . . . 1071
14.6.1. Simplified Syntax . . . 1075
14.6.2. Arrow notation . . . 1078
14.7. SQL - TRAVERSE . . . 1089
14.7.1. Traversal Strategies . . . 1094
14.8. SQL - MOVE VERTEX . . . 1095
14.9. SQL - CREATE VERTEX . . . 1095
14.10. SQL - CREATE EDGE . . . 1097
14.11. Types . . . 1100
14.11.1. SQL - CREATE TYPE . . . 1100
14.11.2. SQL - ALTER TYPE . . . 1103
14.11.3. SQL - DROP TYPE . . . 1108
14.11.4. SQL - TRUNCATE TYPE . . . 1111
14.12. Properties . . . 1112
14.12.1. SQL - CREATE PROPERTY . . . 1112
14.12.2. SQL - ALTER PROPERTY . . . 1117

14.12.3. SQL - DROP PROPERTY . . . 1120
14.13. SQL Indexes . . . 1120
14.13.1. SQL - CREATE INDEX . . . 1120
14.13.2. SQL - REBUILD INDEX . . . 1133
14.13.3. SQL - COMPACT INDEX . . . 1135
14.13.4. SQL - DROP INDEX . . . 1136
14.14. Buckets . . . 1137
14.14.1. SQL - CREATE BUCKET . . . 1137
14.14.2. SQL - DROP BUCKET . . . 1138
14.14.3. SQL - TRUNCATE BUCKET . . . 1138
14.15. Triggers (DDL) . . . 1139
14.15.1. SQL - CREATE TRIGGER . . . 1139
14.15.2. Listing triggers . . . 1142
14.15.3. SQL - DROP TRIGGER . . . 1142
14.16. SQL Materialized Views . . . 1143
14.16.1. SQL - CREATE MATERIALIZED VIEW . . . 1143
14.16.2. SQL - ALTER MATERIALIZED VIEW . . . 1144
14.16.3. SQL - DROP MATERIALIZED VIEW . . . 1145
14.16.4. SQL - REFRESH MATERIALIZED VIEW . . . 1146
14.17. Database Admin . . . 1147
14.17.1. SQL - ALIGN DATABASE . . . 1147
14.17.2. SQL - ALTER DATABASE . . . 1147
14.17.3. SQL - BACKUP DATABASE . . . 1148
14.17.4. SQL - CHECK DATABASE . . . 1149
14.17.5. SQL - EXPORT DATABASE . . . 1161
14.17.6. SQL - IMPORT DATABASE . . . 1166
14.18. Inspection . . . 1167
14.18.1. SQL - EXPLAIN . . . 1167
14.18.2. SQL - PROFILE . . . 1168
14.18.3. SQL - CONSOLE . . . 1169
14.19. Projections . . . 1170
14.19.1. Query result . . . 1171
14.19.2. Nested projections . . . 1173
14.20. Pagination . . . 1180
14.21. Filtering . . . 1181
14.21.1. Operators . . . 1182
14.22. SQL Script . . . 1193
14.22.1. Why use SQL Script . . . 1193
14.22.2. Schema-only scripts run as one schema change . . . 1194
14.22.3. Differences between sql and sqlscript . . . 1194
14.22.4. Syntax reference . . . 1195

14.22.5. BEGIN — start an explicit transaction . . . 1196
14.22.6. COMMIT — commit, retry, fallback . . . 1196
14.22.7. LOCK — explicit type or bucket lock . . . 1197
14.22.8. LET — script variables . . . 1198
14.22.9. SET GLOBAL — database-scoped variables . . . 1199
14.22.10. RETURN — end the script with a value . . . 1199
14.22.11. IF — conditional execution . . . 1200
14.22.12. FOREACH — iterate a collection . . . 1200
14.22.13. WHILE — loop on a condition . . . 1201
14.22.14. BREAK — exit the current loop . . . 1201
14.22.15. SLEEP — pause execution . . . 1202
14.22.16. CONSOLE — log from a script . . . 1202
14.22.17. End-to-end example: a graph upsert with retry . . . 1202
14.22.18. Running scripts . . . 1204
14.23. Functions . . . 1204
14.23.1. Aggregated mode . . . 1204
14.23.2. Inline mode . . . 1205
14.23.3. Function Reference . . . 1205
14.24. Methods . . . 1246
14.24.1. Method Reference . . . 1247
14.25. Triggers . . . 1275
14.25.1. Overview . . . 1275
14.25.2. Trigger Events . . . 1276
14.25.3. Subtypes . . . 1277
14.25.4. SQL Triggers . . . 1277
14.25.5. JavaScript Triggers . . . 1279
14.25.6. Java Triggers . . . 1281
14.25.7. Use Cases . . . 1284
14.25.8. Performance Considerations . . . 1285
14.25.9. Best Practices . . . 1287
14.25.10. Managing Triggers . . . 1288
14.25.11. Limitations . . . 1289
14.25.12. See Also . . . 1289
14.26. User Functions . . . 1289
14.26.1. Overview . . . 1289
14.26.2. Database's Functions . . . 1290
14.26.3. Defining User Functions . . . 1290
14.26.4. User Functions in Java . . . 1294
14.26.5. Best Practices . . . 1298
14.26.6. Advantages Over Neo4j . . . 1298
14.27. SQL SELECT Statements Execution . . . 1298

15. Cypher (openCypher) . . . 1302
15.1. Introduction . . . 1302
15.1.1. Key Features . . . 1302
15.1.2. Using OpenCypher . . . 1302
15.1.3. Using Record IDs (RIDs) . . . 1303
15.1.4. Parameters . . . 1303
15.1.5. Automatic Transaction Handling . . . 1304
15.1.6. Performance Considerations . . . 1304
15.1.7. Resources . . . 1306
15.2. Cypher Tutorial . . . 1306
15.2.1. What is Cypher? . . . 1306
15.2.2. Setting Up . . . 1306
15.2.3. Part 1: Creating Data . . . 1306
15.2.4. Part 2: Reading Data . . . 1308
15.2.5. Part 3: Aggregations . . . 1311
15.2.6. Part 4: Data Modification . . . 1312
15.2.7. Part 5: Advanced Queries . . . 1313
15.2.8. Part 6: Using Functions . . . 1315
15.2.9. Part 7: Best Practices . . . 1316
15.3. Cypher Clauses Reference . . . 1317
15.3.1. MATCH . . . 1317
15.3.2. Path Mode (WALK, TRAIL, ACYCLIC) . . . 1318
15.3.3. Matching a Bound Relationship List . . . 1320
15.3.4. Distinct End Vertices of a Variable-Length Pattern . . . 1321
15.3.5. Named Paths . . . 1321
15.3.6. OPTIONAL MATCH . . . 1322
15.3.7. SEARCH (vector index) . . . 1324
15.3.8. WHERE . . . 1324
15.3.9. RETURN . . . 1326
15.3.10. CREATE . . . 1327
15.3.11. MERGE . . . 1328
15.3.12. SET . . . 1330
15.3.13. DELETE . . . 1332
15.3.14. Reads and Writes in the Same Query . . . 1334
15.3.15. WITH . . . 1336
15.3.16. LET . . . 1338
15.3.17. UNWIND . . . 1338
15.3.18. ORDER BY . . . 1339
15.3.19. SKIP and LIMIT . . . 1340
15.3.20. EXPLAIN . . . 1341
15.3.21. PROFILE . . . 1343

15.3.22. UNION and UNION ALL . . . 1344
15.3.23. Conditional query: WHEN … THEN … ELSE . . . 1345
15.3.24. CALL . . . 1346
15.3.25. FOREACH . . . 1353
15.3.26. LOAD CSV . . . 1354
15.3.27. CREATE CONSTRAINT . . . 1357
15.3.28. DROP CONSTRAINT . . . 1362
15.3.29. SHOW CONSTRAINTS . . . 1363
15.3.30. CREATE INDEX / DROP INDEX . . . 1365
15.3.31. SHOW INDEXES . . . 1368
15.3.32. SHOW USERS . . . 1369
15.3.33. SHOW CURRENT USER . . . 1371
15.3.34. CREATE USER . . . 1371
15.3.35. DROP USER . . . 1372
15.3.36. ALTER USER . . . 1372
15.3.37. SHOW TRANSACTIONS and TERMINATE TRANSACTIONS . . . 1373
15.3.38. Transaction Control (START TRANSACTION, COMMIT, ROLLBACK) . . . 1375
15.3.39. Session Management (SESSION SET, SESSION RESET, SESSION CLOSE) . . . 1376
15.4. Expressions and Functions . . . 1377
15.4.1. Literals . . . 1377
15.4.2. Property Access . . . 1377
15.4.3. CASE Expression . . . 1378
15.4.4. Arithmetic Expressions . . . 1379
15.4.5. Map Literals . . . 1379
15.4.6. List Comprehensions . . . 1380
15.4.7. Map Projections . . . 1381
15.4.8. Aggregation Functions . . . 1382
15.4.9. String Functions . . . 1383
15.4.10. Math Functions . . . 1385
15.4.11. List Functions . . . 1386
15.4.12. Node and Relationship Functions . . . 1387
15.4.13. Path Functions . . . 1390
15.4.14. Type Conversion Functions . . . 1391
15.4.15. SQL Function Bridge . . . 1391
15.4.16. Extended Functions (APOC Compatible) . . . 1393
15.4.17. Operators . . . 1395
15.4.18. Parameters . . . 1396
15.5. Cypher Compatibility Matrix . . . 1397
15.5.1. OpenCypher TCK Compliance . . . 1397
15.5.2. Known Limitations . . . 1398
15.5.3. 1. Temporal Range Limitation (Java Platform Constraint) . . . 1398

15.5.4. 2. Case-Insensitive Identifiers (By Design) . . . 1398
15.5.5. 3. Cross-Type Comparison Strictness (Semantic Edge Case) . . . 1398
15.5.6. 4. User Functions Support . . . 1399
15.5.7. 6. Map-Valued Properties (Multi-Model Boundary) . . . 1399
15.5.8. The Variable of a Variable-Length Relationship Is a List . . . 1400
15.5.9. The WHERE of a WITH . . . 1400
15.5.10. Invalid Inputs Are Refused . . . 1401
15.5.11. A Pattern Expression Is a Boolean Test . . . 1401
15.5.12. Not Implemented Features . . . 1402
15.5.13. Cypher 25 / GQL Strictness . . . 1402
15.5.14. Three-Valued Logic (UNKNOWN) . . . 1402
15.5.15. Zoned DateTime Equality and Ordering . . . 1403
15.5.16. Current Date and Time in a Query . . . 1403
15.5.17. Comparing and Ordering Temporal Values . . . 1404
15.5.18. Duration Arithmetic and Aggregation . . . 1404
15.5.19. Stored TIME, LOCAL TIME, Zoned DATETIME and DURATION Values . . . 1405
15.5.20. Building Temporal Values from a Map or a String . . . 1406
15.5.21. Strict Numeric Types . . . 1406
15.5.22. Temporal and Spatial Value Types . . . 1406
15.5.23. Relationship Types and Sub-types . . . 1407
15.5.24. shortestPath() and Hop Bounds . . . 1407
15.5.25. Self-Loops in Undirected Patterns . . . 1408
15.5.26. Indexes Inherited from a Parent Type . . . 1408
15.5.27. Architectural Differences from Neo4j . . . 1408
15.6. ISO GQL Coverage Statement . . . 1408
15.6.1. Conformance testing . . . 1409
15.6.2. Graph pattern matching . . . 1409
15.6.3. Path modes, path search prefixes and match modes . . . 1410
15.6.4. Query composition . . . 1411
15.6.5. Data modification . . . 1411
15.6.6. Predicates and expressions . . . 1412
15.6.7. Value types . . . 1413
15.6.8. Functions . . . 1413
15.6.9. Transactions and sessions . . . 1414
15.6.10. Catalog, graph types and diagnostics . . . 1415
16. Gremlin . . . 1416
16.1. Gremlin . . . 1416
16.1.1. Gremlin from Java API . . . 1416
16.1.2. Transactions and closing a graph . . . 1417
16.1.3. Vertex property identifiers . . . 1418
16.1.4. Gremlin through Postgres Driver . . . 1418

16.1.5. Gremlin through HTTP/JSON . . . 1418
16.1.6. Use the Gremlin Server . . . 1419
16.1.7. Listing and terminating running traversals . . . 1423
16.1.8. Known Limitations with the Gremlin Implementation . . . 1423
16.1.9. Recommended Tools with Gremlin . . . 1425
17. GraphQL . . . 1426
17.1. GraphQL . . . 1426
17.1.1. Type definition . . . 1426
17.1.2. Introspection of database types . . . 1428
17.1.3. Query arguments . . . 1429
17.1.4. Query variables . . . 1431
17.1.5. Field aliases . . . 1432
17.1.6. Fragments . . . 1432
17.1.7. Supported directives . . . 1433
18. MongoDB QL . . . 1437
18.1. MongoDB Query Language . . . 1437
18.1.1. MongoDB Query Language . . . 1437
18.1.2. MongoDB Protocol Plugin . . . 1438
19. Redis . . . 1446
19.1. Redis Protocol . . . 1446
19.1.1. Installation . . . 1446
19.1.2. How it works . . . 1447
19.1.3. Transient (RAM Only) Commands . . . 1447
19.1.4. Persistent Commands . . . 1449
19.1.5. Redis via HTTP API . . . 1453
20. Graph Algorithms . . . 1458
20.1. Graph Algorithms . . . 1458
20.1.1. Algorithm Index . . . 1458
20.1.2. Categories . . . 1461
20.2. Shortest Path . . . 1461
20.2.1. Choosing an Algorithm . . . 1461
20.2.2. Dijkstra, A* and CCH . . . 1462
20.2.3. Try It: the Florida Road Network . . . 1463
20.2.4. Which Engine Answered? . . . 1469
20.2.5. Performance . . . 1469
20.3. Path Finding . . . 1471
20.3.1. algo.dijkstra . . . 1472
20.3.2. algo.cch.shortestPath . . . 1474
20.3.3. algo.astar . . . 1475
20.3.4. algo.bellmanford . . . 1477
20.3.5. algo.allsimplepaths . . . 1478

20.3.6. algo.kShortestPaths . . . 1480
20.3.7. algo.apsp . . . 1481
20.3.8. algo.bfs . . . 1483
20.3.9. algo.dfs . . . 1484
20.3.10. algo.dijkstra.singleSource . . . 1485
20.3.11. algo.longestPath . . . 1487
20.3.12. algo.msa . . . 1488
20.3.13. algo.steinerTree . . . 1489
20.4. Centrality . . . 1491
20.4.1. algo.pagerank . . . 1491
20.4.2. algo.betweenness . . . 1493
20.4.3. algo.closeness . . . 1494
20.4.4. algo.degree . . . 1495
20.4.5. algo.harmonic . . . 1497
20.4.6. algo.eigenvector . . . 1498
20.4.7. algo.hits . . . 1499
20.4.8. algo.katz . . . 1501
20.4.9. algo.voteRank . . . 1502
20.4.10. algo.eccentricity . . . 1503
20.4.11. algo.personalizedPageRank . . . 1505
20.4.12. algo.articlerank . . . 1507
20.5. Community Detection . . . 1508
20.5.1. algo.wcc . . . 1508
20.5.2. algo.scc . . . 1510
20.5.3. algo.louvain . . . 1511
20.5.4. algo.leiden . . . 1512
20.5.5. algo.labelpropagation . . . 1514
20.5.6. algo.hierarchicalClustering . . . 1516
20.5.7. algo.slpa . . . 1518
20.5.8. algo.maxKCut . . . 1519
20.6. Structural Analysis . . . 1521
20.6.1. algo.kcore . . . 1521
20.6.2. algo.triangleCount . . . 1522
20.6.3. algo.articulationPoints . . . 1523
20.6.4. algo.bridges . . . 1524
20.6.5. algo.mst . . . 1526
20.6.6. algo.topologicalSort . . . 1527
20.6.7. algo.bipartite . . . 1528
20.6.8. algo.graphColoring . . . 1529
20.6.9. algo.cycleDetection . . . 1531
20.6.10. algo.densestSubgraph . . . 1532

20.6.11. algo.clique . . . 1533
20.6.12. algo.kTruss . . . 1534
20.6.13. algo.biconnectedComponents . . . 1536
20.6.14. algo.localClusteringCoefficient . . . 1537
20.6.15. algo.bipartiteMatching . . . 1538
20.7. Similarity and Link Prediction . . . 1540
20.7.1. algo.jaccard . . . 1540
20.7.2. algo.adamicAdar . . . 1541
20.7.3. algo.commonNeighbors . . . 1543
20.7.4. algo.preferentialAttachment . . . 1544
20.7.5. algo.resourceAllocation . . . 1545
20.7.6. algo.simRank . . . 1547
20.7.7. algo.knn . . . 1548
20.7.8. algo.sameCommunity . . . 1549
20.7.9. algo.totalNeighbors . . . 1551
20.8. Network Flow . . . 1552
20.8.1. algo.maxFlow . . . 1552
20.9. Traversal and Sampling . . . 1553
20.9.1. algo.randomWalk . . . 1553
20.10. Network Science Metrics . . . 1555
20.10.1. algo.assortativity . . . 1555
20.10.2. algo.richClub . . . 1556
20.10.3. algo.influenceMaximization . . . 1557
20.11. Community Quality Metrics . . . 1559
20.11.1. algo.modularityScore . . . 1559
20.11.2. algo.conductance . . . 1560
20.12. Graph Statistics . . . 1561
20.12.1. algo.graphSummary . . . 1561
20.13. Node Embedding . . . 1563
20.13.1. algo.fastrp . . . 1563
20.13.2. algo.node2vec . . . 1565
20.13.3. algo.graphsage . . . 1566
20.13.4. algo.hashgnn . . . 1568
20.14. SQL Path Functions . . . 1570
20.14.1. dijkstra() SQL Function . . . 1570
20.14.2. cchShortestPath() SQL Function . . . 1572
20.14.3. duanSSSP() SQL Function . . . 1573
20.14.4. bellmanFord() SQL Function . . . 1574
20.14.5. shortestPath() SQL Function . . . 1575
20.15. Notes . . . 1577
20.15.1. relTypes Parameter . . . 1577

20.15.2. direction Parameter . . . 1577
20.15.3. Unidirectional Edge Types . . . 1578
20.15.4. Dangling Edge Entries . . . 1578
20.15.5. Config Map Parameters . . . 1578
20.15.6. Memory Considerations . . . 1578
21. Extended Functions . . . 1580
21.1. Extended Functions Reference . . . 1580
21.1.1. Overview . . . 1580
21.1.2. APOC Prefix Compatibility . . . 1580
21.1.3. Function Namespaces . . . 1581
21.1.4. Function Categories . . . 1581
21.1.5. Migration Guide . . . 1582
21.2. Aggregation Functions . . . 1583
21.2.1. agg.first() . . . 1584
21.2.2. agg.last() . . . 1584
21.2.3. agg.maxItems() . . . 1584
21.2.4. agg.median() . . . 1585
21.2.5. agg.minItems() . . . 1585
21.2.6. agg.nth() . . . 1585
21.2.7. agg.percentiles() . . . 1585
21.2.8. agg.product() . . . 1586
21.2.9. agg.slice() . . . 1586
21.2.10. agg.statistics() . . . 1586
21.2.11. Collection Functions . . . 1586
21.2.12. coll.avg() . . . 1587
21.2.13. coll.distinct() . . . 1587
21.2.14. coll.flatten() . . . 1588
21.2.15. coll.indexOf() . . . 1588
21.2.16. coll.insert() . . . 1589
21.2.17. coll.max() . . . 1589
21.2.18. coll.min() . . . 1590
21.2.19. coll.pairsMin() . . . 1590
21.2.20. coll.remove() . . . 1591
21.2.21. coll.sort() . . . 1591
21.2.22. coll.sum() . . . 1592
21.2.23. coll.toSet() . . . 1592
21.2.24. coll.union() . . . 1593
21.2.25. coll.unionAll() . . . 1593
21.2.26. Map Functions . . . 1594
21.2.27. map.clean() . . . 1594
21.2.28. map.flatten() . . . 1594

21.2.29. map.fromLists() . . . 1594
21.2.30. map.fromPairs() . . . 1595
21.2.31. map.groupBy() . . . 1595
21.2.32. map.merge() . . . 1595
21.2.33. map.mergeList() . . . 1596
21.2.34. map.removeKey() . . . 1596
21.2.35. map.removeKeys() . . . 1596
21.2.36. map.setKey() . . . 1596
21.2.37. map.sortedProperties() . . . 1597
21.2.38. map.submap() . . . 1597
21.2.39. map.unflatten() . . . 1597
21.2.40. map.values() . . . 1597
21.2.41. Convert Functions . . . 1597
21.2.42. convert.fromJsonList() . . . 1598
21.2.43. convert.fromJsonMap() . . . 1598
21.2.44. convert.toBoolean() . . . 1598
21.2.45. convert.toFloat() . . . 1599
21.2.46. convert.toInteger() . . . 1599
21.2.47. convert.toJson() . . . 1599
21.2.48. convert.toList() . . . 1599
21.2.49. convert.toMap() . . . 1600
21.2.50. convert.toSet() . . . 1600
21.2.51. convert.toString() . . . 1600
21.2.52. Create Functions . . . 1600
21.2.53. create.uuid() . . . 1601
21.2.54. create.uuidBase64() . . . 1601
21.2.55. create.vNode() . . . 1601
21.2.56. create.vRelationship() . . . 1601
21.3. Date & Math Functions . . . 1602
21.3.1. Date Functions (Extended) . . . 1602
21.3.2. date.add() . . . 1602
21.3.3. date.convert() . . . 1602
21.3.4. date.currentTimestamp() . . . 1603
21.3.5. date.field() . . . 1603
21.3.6. date.fields() . . . 1603
21.3.7. date.format() . . . 1603
21.3.8. date.fromISO8601() . . . 1604
21.3.9. date.parse() . . . 1604
21.3.10. date.systemTimezone() . . . 1604
21.3.11. date.toISO8601() . . . 1605
21.3.12. Math Functions (Extended) . . . 1605

21.3.13. math.cosh() . . . 1605
21.3.14. math.maxDouble() . . . 1605
21.3.15. math.maxLong() . . . 1605
21.3.16. math.minLong() . . . 1606
21.3.17. math.round() . . . 1606
21.3.18. math.sigmoid() . . . 1607
21.3.19. math.sigmoidPrime() . . . 1607
21.3.20. math.sinh() . . . 1607
21.3.21. math.tanh() . . . 1607
21.3.22. Number Functions . . . 1608
21.3.23. number.format() . . . 1608
21.3.24. Node Functions . . . 1608
21.3.25. node.degree() . . . 1609
21.3.26. node.degree.in() . . . 1609
21.3.27. node.degree.out() . . . 1609
21.3.28. node.id() . . . 1609
21.3.29. node.labels() . . . 1609
21.3.30. node.relationship.exists() . . . 1610
21.3.31. node.relationship.types() . . . 1610
21.3.32. Relationship Functions . . . 1610
21.3.33. rel.endNode() . . . 1610
21.3.34. rel.id() . . . 1611
21.3.35. rel.startNode() . . . 1611
21.3.36. rel.type() . . . 1611
21.3.37. Algorithm Procedures . . . 1611
21.3.38. algo.allsimplepaths() . . . 1611
21.3.39. algo.astar() . . . 1612
21.3.40. algo.dijkstra() . . . 1613
21.3.41. Path Functions (Extended) . . . 1614
21.3.42. path.combine() . . . 1614
21.3.43. path.create() . . . 1614
21.3.44. path.elements() . . . 1614
21.3.45. path.slice() . . . 1615
21.3.46. Path Expansion Procedures . . . 1615
21.3.47. Relationship and label filters . . . 1615
21.3.48. path.expand() . . . 1618
21.3.49. path.expandconfig() . . . 1619
21.3.50. path.spanningtree() . . . 1620
21.3.51. path.subgraphall() . . . 1621
21.3.52. path.subgraphnodes() . . . 1622
21.3.53. Control Flow Procedures . . . 1622

21.3.54. do.when() . . . 1622
21.3.55. Refactor Procedures . . . 1623
21.3.56. refactor.cloneNodesWithRelationships() . . . 1623
21.3.57. refactor.mergeNodes() . . . 1624
21.4. Schema & Meta Procedures . . . 1626
21.4.1. Meta/Schema Procedures . . . 1626
21.4.2. meta.graph() . . . 1626
21.4.3. meta.nodetypeproperties() . . . 1626
21.4.4. meta.reltypeproperties() . . . 1627
21.4.5. meta.schema() . . . 1627
21.4.6. meta.stats() . . . 1628
21.4.7. Merge Procedures . . . 1629
21.4.8. merge.node() . . . 1629
21.4.9. merge.relationship() . . . 1629
21.4.10. text.byteCount() . . . 1631
21.4.11. text.camelCase() . . . 1631
21.4.12. text.capitalize() . . . 1632
21.4.13. text.capitalizeAll() . . . 1632
21.4.14. text.charAt() . . . 1632
21.4.15. text.code() . . . 1633
21.4.16. text.decapitalize() . . . 1633
21.4.17. text.decapitalizeAll() . . . 1633
21.4.18. text.format() . . . 1634
21.4.19. text.hammingDistance() . . . 1634
21.4.20. text.hexValue() . . . 1635
21.4.21. text.indexOf() . . . 1635
21.4.22. text.jaroWinklerDistance() . . . 1636
21.4.23. text.join() . . . 1637
21.4.24. text.levenshteinDistance() . . . 1637
21.4.25. text.levenshteinSimilarity() . . . 1638
21.4.26. text.lpad() . . . 1638
21.4.27. text.random() . . . 1639
21.4.28. text.regexReplace() . . . 1639
21.4.29. text.replace() . . . 1639
21.4.30. text.rpad() . . . 1640
21.4.31. text.slug() . . . 1640
21.4.32. text.snakeCase() . . . 1641
21.4.33. text.sorensenDiceSimilarity() . . . 1642
21.4.34. text.split() . . . 1642
21.4.35. text.upperCamelCase() . . . 1643
21.5. Utility Functions . . . 1644

21.5.1. util.compress() . . . 1644
21.5.2. util.decompress() . . . 1644
21.5.3. util.md5() . . . 1645
21.5.4. util.sha1() . . . 1645
21.5.5. util.sha256() . . . 1645
21.5.6. util.sha512() . . . 1645
21.5.7. util.sleep() . . . 1646
21.5.8. util.validate() . . . 1646
21.5.9. util.validatePredicate() . . . 1647
21.6. Vector Functions . . . 1647
21.6.1. Aggregation Functions . . . 1647
21.6.2. Basic Operations . . . 1649
21.6.3. Nearest Neighbor Search . . . 1650
21.6.4. Normalization and Norms . . . 1655
21.6.5. Quantization . . . 1657
21.6.6. Scoring and Fusion . . . 1658
21.6.7. Similarity and Distance . . . 1661
21.6.8. Sparse Vectors . . . 1663
21.6.9. Statistics . . . 1665
21.6.10. Utility Functions . . . 1666

Documentation

Chapter 1. Welcome

1.1. ArcadeDB Documentation

1.1.1. Tutorials

Step-by-step guides to learn ArcadeDB's data models, query languages, and drivers.

1.1.2. ArcadeDB Academy

Free, self-paced courses with certification on completion. Hosted at arcadedb.com/academy.

1.1.3. Use Cases

Production-ready example projects you can clone and run with `docker compose up`.

1.1.4. Community & Resources

Chapter 2. Get Started

2.1. Get Started with ArcadeDB

2.1.1. Choose your starting point

2.1.2. Connect from your application

ArcadeDB exposes several wire protocols and drivers, so you can keep your existing client libraries:

- HTTP / JSON REST API — language-agnostic, no driver required
- PostgreSQL wire protocol — existing Postgres clients connect unchanged
- Neo4j Bolt protocol — reuse Neo4j’s official drivers
- MongoDB QL — MongoDB drivers
- Redis commands — Redis clients
- MCP server — for AI agents and LLM tooling
- Native Java API — embedded mode
- JDBC driver — for BI and traditional SQL tools

2.1.3. Pick a query language

ArcadeDB supports six query languages over the same data — use whichever your team already knows:

- SQL — extended with graph, full-text, and vector functions
- Cypher (openCypher) — drop-in replacement for Neo4j Cypher
- Apache TinkerPop Gremlin
- GraphQL
- MongoDB QL
- Redis commands

2.1.4. Next steps

- Records, Documents, Vertices & Edges — how ArcadeDB models data

- Schema — types, properties, constraints
- Transactions — ACID guarantees
- Self-managed deployment — production install
- ArcadeDB Academy — free, self-paced courses with certification

2.2. What is ArcadeDB?

ArcadeDB is a **multi-model database management system**: graph, document, key/value, time-series, vector, and geospatial data live on the **same storage** and inside the **same transaction**. One process, one schema, one query language of your choice — instead of a separate database for each shape of data.

If you've ever stitched together PostgreSQL, MongoDB, Neo4j, Elasticsearch, and a vector database to back a single application, this page explains why ArcadeDB collapses that stack into one engine, and why we believe the **graph model** (more on that below) is a better default than people realise.

2.2.1. TL;DR

- Six data models — graph, document, key/value, full-text, vector, time-series — share the same storage and transaction.
- Six query languages — SQL, Cypher, Gremlin, GraphQL, MongoDB QL, Redis — read and write the same records.
- Native graph: edges are physical pointers between records, every traversal hop is O(1).
- Embeddable in any JVM (16 MB heap minimum) or run as a server with Docker / Kubernetes.
- Apache 2.0 — free for commercial use, no copyleft, no Enterprise edition gate.

2.2.2. The problem with running many databases

Most teams that need to combine relationships, JSON, full-text search and vector similarity end up with **polyglot persistence**: PostgreSQL for transactional data, Neo4j for the graph, Elasticsearch for search, MongoDB for flexible documents, Pinecone or Qdrant for embeddings. Each system brings:

- a separate deployment, backup strategy, security model, and failure mode;
- an ETL pipeline (or change-data-capture queue) to keep data in sync — every sync introduces lag, complexity, and bugs;
- a learning curve per database for every developer on the team;
- a multiplier on infrastructure cost and operational headcount.

A single user request often turns into a chain of sequential calls across these systems: query Postgres, then Neo4j, then Elasticsearch, then stitch the results together in the application layer. The total response time is the **sum** of every hop, plus the cost of serializing data three times.

The animation runs faster than reality, but the proportions hold: by the time the polyglot scenario

finishes one round of three sequential calls, ArcadeDB has completed several full transactions. The numbers below match the animation:

Polyglot persistence (sequential)	**Database**	**Latency**
Your app → PostgreSQL	Relational	~15 ms
then … → MongoDB	Document	~20 ms
then … → Neo4j	Graph	~12 ms
Total: ~47 ms+ (sum of calls + serialization overhead)		

ArcadeDB (single call)	**Engine**	**Latency**
Your app → ArcadeDB (graph + document + search in one query)	Multi-model	~5 ms
Total: ~5 ms (one call, zero chaining, no serialization)		

2.2.3. One engine, every model

ArcadeDB collapses the polyglot stack into a single engine. A single ACID transaction can:

- create a JSON document,
- connect it as a vertex in a graph,
- index its embedding for vector similarity,
- and append a time-series sample for it,

— atomically. There is no replication lag **between models**, because there is no replication: the data exists once, on one storage layer.

The consequences run deeper than just lower latency:

- **Lower latency** — one network round-trip, no waiting on the slowest database in the chain.
- **One operational model** — one backup, one HA topology, one monitoring dashboard, one upgrade path.
- **One learning curve** — your team learns ArcadeDB once, not five products.
- **Real cross-model queries** — a SQL statement can traverse a graph, filter on document fields, and rank by vector similarity in a single pass:

```
SELECT FROM (
  MATCH {type: User, as: u} -Follows-> {type: User, as: f}
  RETURN f
) WHERE vector.cosineSimilarity(f.embedding, ?) > 0.8
```

The graph traversal and the vector comparison run against the same records in the same transaction. No cross-database join, no eventual consistency, no glue code.

2.2.4. The graph model is our core — and it's native

ArcadeDB is a **native graph database**. That phrase carries weight, so let's unpack it.

In a relational database, a relationship between two rows is a foreign key — a value in one table that **matches** a value in another. To follow it, the database engine must look the other side up, typically via an index. The cost grows with the size of the joined tables: a `JOIN` on millions of rows forces the planner to choose between hash join, merge join, or nested loop, and none of those are free. As your data grows, your queries slow down.

In a native graph, **edges are physical links between records.** Following an edge dereferences a pointer; the database does not scan, does not probe an index, does not build a hash table. Every traversal hop is **O(1) regardless of database size.** This is called **index-free adjacency**, and it changes the kind of question you can ask:

- "Friends of friends of friends of Alice" — three pointer hops, milliseconds, no matter how many users you have.
- "Every person within 4 degrees of someone who bought product X in the last hour" — a traversal, not a query plan with three nested joins.
- "Shortest path between two arbitrary entities in a 100-million-vertex graph" — feasible.

In ArcadeDB, edges are first-class records: a `Follows` edge can carry properties like `since`, `weight`, `label`. Lightweight edges exist for relationships with no properties (zero per-edge storage). Both are stored as physical links — there are no join tables, no adjacency lists bolted onto a relational schema, no LINKBAG-style indirection.

This is also why we recommend the graph model as the **default** even for "non-graph" applications. Anything you can model as parent/child, owner/owned, follower/followee, before/after, or "belongs to" is naturally a graph. Modelling it as a graph means a `JOIN` becomes a single-pointer dereference, and your query keeps the same speed when the table grows from a thousand rows to a billion.

- Graph database concepts
- SQL `MATCH`
- Cypher (openCypher)
- Apache TinkerPop Gremlin

2.2.5. The other models you get

Documents

Documents are JSON-like records with nested properties, lists, and maps. Document types can be **schema-full** (every property declared with a type and constraints), **schema-less** (no validation, free-form), or **schema-hybrid** (some properties enforced, others free-form). Types support inheritance — a `Customer` type can extend `Person` and inherit its properties, indexes, and constraints.

- Records, documents, vertices, edges
- Schema modes

Key-value

Every record has a Record ID (RID); fetching by RID is O(1). Buckets can also act as dedicated key/value namespaces — useful for caching, session storage, and configuration without a type schema.

Full-text search

Built-in Lucene engine with fuzzy matching, integrated into the query languages. No external Elasticsearch deployment required for typical search workloads.

- Full-text index

Time-series

Columnar storage with Gorilla XOR compression for floats and Delta-of-Delta for timestamps — 0.4 to 1.4 bytes per sample. Ingestion via the InfluxDB Line Protocol (works with Telegraf, Grafana Agent, etc.); query with PromQL or SQL analytical functions like `ts.timeBucket`, `ts.rate`, `ts.percentile`.

- Time-series concepts

Vector

JVector indexes (HNSW or Vamana / DiskANN) with COSINE, DOT_PRODUCT, and EUCLIDEAN similarity, and INT8 / BINARY quantisation for memory savings. Vector indexes participate in ACID transactions and integrate directly into SQL through `vector.neighbors()` and `vector.cosineSimilarity()`.

- Vector search

Geospatial

Built-in geometric and geographic functions over points, polygons, and bounding boxes — distance queries, containment, intersections — all in the same SQL planner as the rest of the data.

- Geospatial

2.2.6. Pick the query language your team already knows

ArcadeDB speaks the languages your team is comfortable with. They all read and write the **same** records on the **same** storage:

- **SQL** — extended with graph traversal, full-text, vector, and time-series functions
- **Cypher (openCypher)** — drop-in replacement for Neo4j Cypher, 97.8% TCK compliant
- **Gremlin** — Apache TinkerPop graph traversal
- **GraphQL** — schema-driven queries over HTTP
- **MongoDB QL** — document-oriented queries via the MongoDB protocol
- **Redis commands** — key/value access via the Redis wire protocol

A record created through SQL is immediately visible to a Gremlin traversal, and an edge created through Cypher can be queried through GraphQL.

2.2.7. Embedded or server, your choice

ArcadeDB runs in two modes:

- **Embedded** — link the JAR into a JVM application and run the database in-process. No separate server, no network hop, no IPC overhead. Footprint starts at ~16 MB of heap. Apache 2.0; no Enterprise edition required.
- **Server** — run as a process and connect from any language over HTTP/REST, the PostgreSQL wire protocol, the Neo4j Bolt protocol, MongoDB drivers, Redis clients, the gRPC API, or MCP for AI agents.

Many ArcadeDB deployments mix both: an embedded engine inside a hot-path service plus a server cluster for analytics and external clients, all sharing the same database files.

2.2.8. Why it's so fast

ArcadeDB is written in **Low-Level Java** — Java 21+, but without high-level abstractions on the hot path. Few objects are allocated at run time, so the garbage collector rarely needs to act. The kernel uses cache-line-aware data layouts, lock-free reads where possible, and other mechanical-sympathy techniques to make the JVM behave more like a hand-tuned native binary.

The storage engine is an LSM-tree with bucket partitioning, page caching, and a write-ahead log. On the LDBC Graphalytics benchmark, ArcadeDB outperforms popular graph databases by 5×–80× on PageRank, BFS, and weakly-connected-components workloads.

- Storage internals
- LSM-tree algorithm
- Performance comparison vs. Neo4j

2.2.9. Apache 2.0, free for commercial use

ArcadeDB Community Edition is released under the Apache 2.0 license — free for any purpose, commercial use included, no copyleft. You can embed it in proprietary products without publishing your source.

When code is public, anyone can scrutinize, test, report, and resolve issues. Open source moves faster than the proprietary world, and the trust comes from being able to read the code rather than a marketing page.

For mission-critical production deployments, professional support is available with guaranteed response times and SLA coverage. See the SLA page for service-level details.

2.2.10. Where to next

- Run ArcadeDB — install with Docker, native binaries, or Kubernetes

- Multi-model architecture — see how all six models share one engine
- 10-minute Java tutorial — embedded mode quickstart
- Python quickstart — connect via PostgreSQL or HTTP
- ArcadeDB Academy — free, self-paced courses with certification

2.3. Run ArcadeDB

You can run ArcadeDB in the following ways:

- In the cloud, on Amazon AWS, Microsoft Azure, or Google Cloud. On AWS and Azure you can launch the official AWS Marketplace AMI on EC2 or the Azure Marketplace image on Azure Virtual Machines in a few minutes.
- On-premise, on your servers, any OS is good. You can run with Docker, Podman, Kubernetes or bare metal.
- On x86(-64), arm(64), or any other hardware supporting a JRE (Java* Runtime Environment)
- Embedded, if you develop with a language that runs on the JVM (Java* Virtual Machine)*

To reach the best performance, use ArcadeDB in embedded mode to reach two million insertions per second on common hardware. If you need to scale up with the queries, run a HA (high availability) configuration with at least three servers, and a load balancer in front. Run ArcadeDB with Kubernetes to have an automatic setup of servers in HA and a load balancer upfront.

2.3.1. Embedded

This mode is possible only if your application is running in a JVM* (Java* Virtual Machine). In this configuration ArcadeDB runs in the same JVM as your application. In this way you completely avoid the client/server communication cost (TCP/IP, marshalling/unmarshalling, etc.) If the JVM that hosts your application crashes, then also ArcadeDB would crash, but don't worry, ArcadeDB uses a WAL to recover partially committed transactions. Your data is safe! Check the Embedded Server section.

2.3.2. Client-Server

This is the classic way people use a DBMS, like with relational databases. The ArcadeDB server exposes HTTP/JSON API, so you can connect to ArcadeDB from any language without even using drivers. Take a look at the API and Driver Reference chapter for more information.

2.3.3. High Availability (HA)

You can spin up as many ArcadeDB servers as you want to have a HA setup and scale up with queries that can be executed on any servers. ArcadeDB uses a Raft based election system to guarantee the consistency of the database. For more information look at High Availability.

2.3.4. Getting Started

See Installation for guides to install under Linux, MacOS, Windows. Alternatively, you can run

ArcadeDB using Docker:

```
docker run --rm -p 2480:2480 -p 5432:5432 \
    --name arcadedb arcadedata/arcadedb:26.11.1
```

2.4. Running ArcadeDB with Docker

To run the ArcadeDB server with Docker, type this (replace <password> with the root password you want to use):

```
$ docker run --rm -p 2480:2480 --name my_arcadedb \
             --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata" \
             --hostname my_arcadedb arcadedata/arcadedb:26.11.1
```

If there are no errors, Docker prints immediately the container id. You can use that id to stop the container, or execute some commands from it.

Pass ArcadeDB settings in `ARCADEDB_SETTINGS` and extra JVM flags in `JAVA_OPTS`. Docker **replaces** an environment variable rather than adding to it, so putting `-Darcadedb.*` settings in `JAVA_OPTS` would discard the garbage collector the image selects for you. Keeping the two apart means your container runs the same JVM as one started with no environment at all.

With `arcadedb.server.apiTokenRequireSecureTransport=true` the server refuses to mint an API token over a plain-HTTP connection that does not come from the loopback address. The setting is `false` by default (an unprotected mint is only logged at WARNING) and is scheduled to default to `true` from 27.1.1. Once it is on, a container reached through a published port (`-p 2480:2480`) usually sees the Docker network gateway as the peer, not loopback, so **Create Token** in Studio on `http://localhost:2480` is answered with a `412`, and opening Studio "from the server host itself" does not help. For a local development container, either enable HTTPS or leave the check off with `-Darcadedb.server.apiTokenRequireSecureTransport=false` in `ARCADEDB_SETTINGS`. Do not turn it off for a server reachable from other machines. Behind a TLS-terminating reverse proxy, list the proxy in `arcadedb.server.apiTokenTrustedProxies` instead; see Create an API token.

To run the console from the container started above, use:

```
$ docker exec -it my_arcadedb bin/console.sh
ArcadeDB Console v26.11.1 - Copyrights (c) 2021 Arcade Data (https://arcadedb.com)

>
```

The ArcadeDB image can also be used with Podman, just replace `docker` with `podman` in the examples.

The image declares the ports `2480` (HTTP API and Studio), `2434` (Raft replication), `8182` (Gremlin), `5432` (PostgreSQL), `6379` (Redis), `27017` (MongoDB), `7687` (Bolt) and `50051` (gRPC), so `docker run -P` publishes all of them and image inspection lists them. A wire protocol only listens once its plugin is enabled, and you can always publish a single port with `-p`.

2.4.1. Quick start with the OpenBeer database

You can run ArcadeDB server with a demo database in less than 1 minute. Run ArcadeDB server with docker specifying the database to import as a parameter in the docker command.

Example of running ArcadeDB Server with all the plugins enabled (Redis, Postgres, Mongo, Gremlin) that download and install OrientDB's `OpenBeer` dataset:

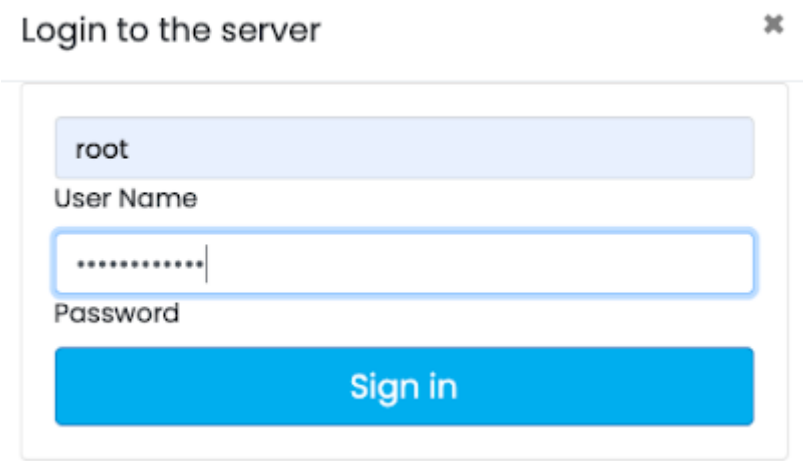
```
$ docker run --rm  -p 2480:2480 -p 6379:6379 -p 5432:5432 -p 8182:8182 --env
ARCADEDB_SETTINGS="\
   -Darcadedb.server.rootPassword=playwithdata \

-Darcadedb.server.defaultDatabases=Imported[root]{import:https://github.com/ArcadeData
/arcadedb-datasets/raw/main/orientdb/OpenBeer.gz} \
   -Darcadedb.server.plugins=Redis:com.arcadedb.redis.RedisProtocolPlugin, \
                             MongoDB:com.arcadedb.mongo.MongoDBProtocolPlugin, \
                             Postgres:com.arcadedb.postgres.PostgresProtocolPlugin, \

GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugin" \
         arcadedata/arcadedb:latest
```

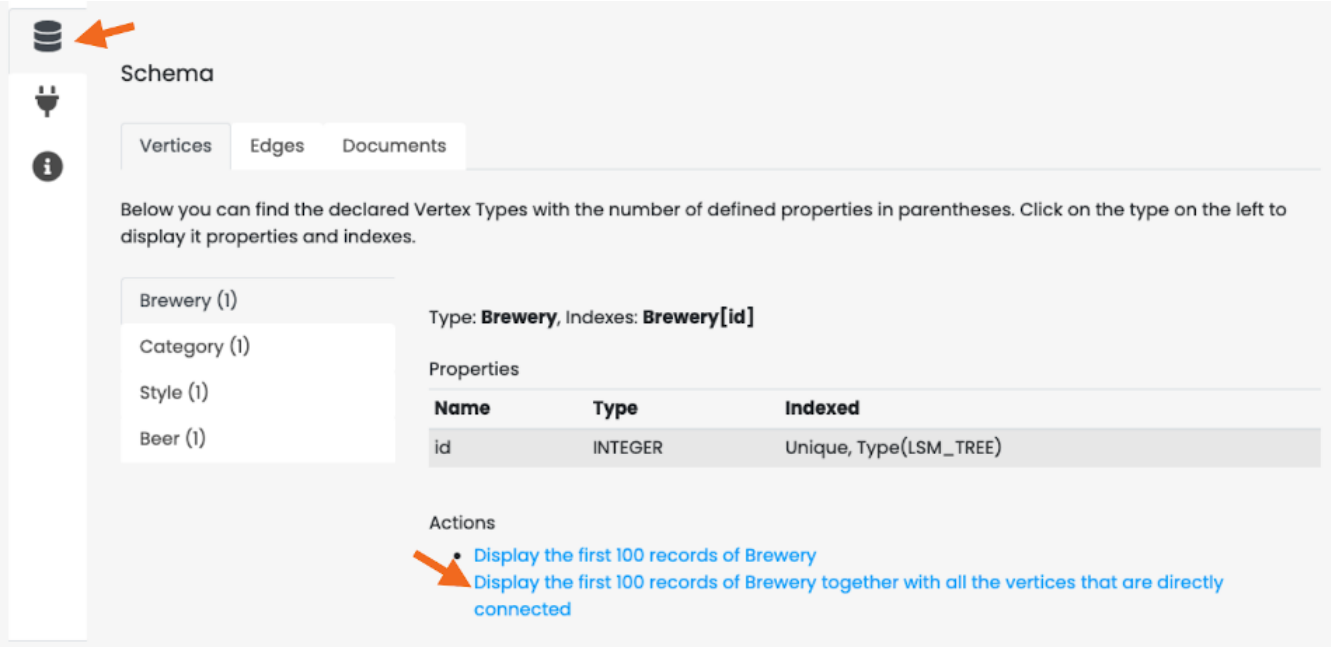
Now point your browser on **http://localhost:2480** and you'll see ArcadeDB Studio. Now enter "root" as a user and "playwithdata" as a password.

User and password are specified in the docker command above.

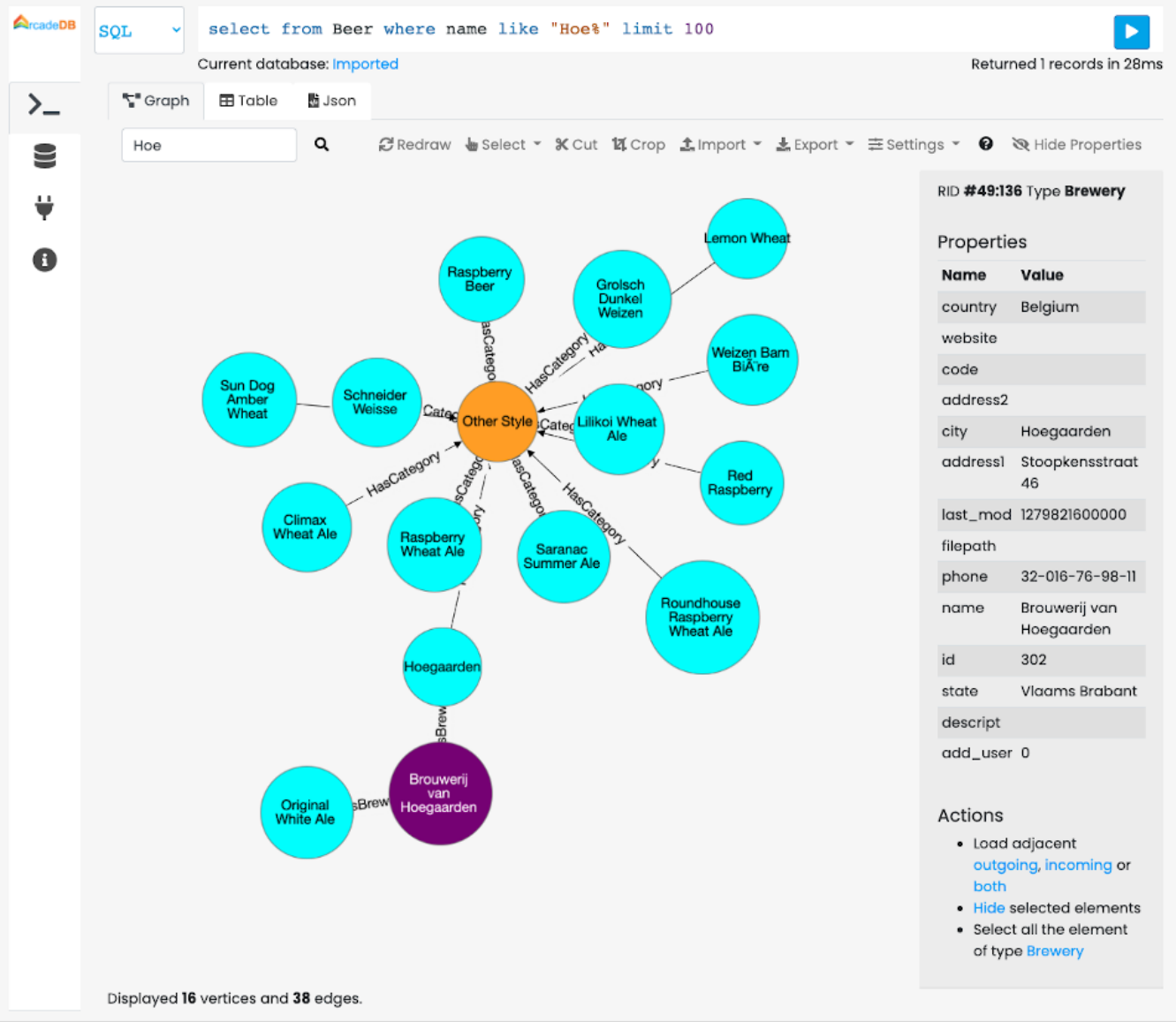


Now click on the "Database" icon on the toolbar on the left. This is the database schema. Click on

"OpenBeer" vertex type and then on the action "Display the first 100 records of Beer together with all the vertices that are directly connected".



You should see the first 100 beers in the database and all their connections.



Copyright 2021 Arcade Data Ltd

2.4.2. Persistence

By default, data created in a Docker container is lost when the container stops. To preserve your ArcadeDB databases beyond the container's lifecycle, you need to configure persistent storage by mounting volumes.

Understanding ArcadeDB's Database Directory

ArcadeDB stores its databases in the `/home/arcadedb/databases` directory by default. This is the path you need to mount to persist your data. You can customize this location using the `-Darcadedb.server.databaseDirectory` setting if needed.

Docker Volumes vs. Bind Mounts

Docker offers two approaches for persistent storage:

1. **Docker volumes** - Docker-managed storage sections on the host filesystem. Docker handles the storage location, permissions, and lifecycle. This is the recommended approach for most use cases as it's more portable and easier to manage.

2. **Bind mounts** - Direct mapping to a specific path on your host machine. This gives you full control over the exact location of your data files and allows direct access to them from the host. Useful when you need to inspect, backup, or manipulate the database files directly.

See the Docker storage documentation for detailed setup instructions.

Persisting the Database Directory

The default path `/home/arcadedb/databases` contains all your database files. Mount this directory to make your data persistent. Here's an example using a bind mount:

```
$ docker run --rm -p 2480:2480 --name my_arcadedb \
    -v /path/on/host/databases:/home/arcadedb/databases \
    --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata" \
    --hostname my_arcadedb arcadedata/arcadedb:26.11.1
```

Replace `/path/on/host/databases` with your desired directory path. All database read and write operations will occur on this mounted volume.

If you prefer using Docker volumes instead of bind mounts, replace `-v /path/on/host/databases:/home/arcadedb/databases` with `-v arcadedb-data:/home/arcadedb/databases` where `arcadedb-data` is your chosen volume name.

Alternative: Custom Database Directory

If you need to change where ArcadeDB stores databases, use the `-Darcadedb.server.databaseDirectory` setting and mount your volume to match the new path:

```
$ docker run --rm -p 2480:2480 --name my_arcadedb \
    -v /path/on/host/databases:/mydatabases \
    --env ARCADEDB_SETTINGS="-Darcadedb.server.databaseDirectory=/mydatabases \
                    -Darcadedb.server.rootPassword=playwithdata" \
    --hostname my_arcadedb arcadedata/arcadedb:26.11.1
```

This example changes the database directory to `/mydatabases` and mounts the volume to that location.

Alternative: Backup-Only Persistence

For better performance, you can keep databases in container storage and only persist backups. All database operations occur in fast container storage, but you must perform regular backups:

```
$ docker run --rm -p 2480:2480 --name my_arcadedb \
    -v /path/on/host/backups:/home/arcadedb/backups \
    --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata" \
    --hostname my_arcadedb arcadedata/arcadedb:26.11.1
```

The default backup directory is `/home/arcadedb/backups`. You can customize it with `-Darcadedb.server.backupDirectory=/mybackup`. Remember to execute regular `BACKUP DATABASE` operations.

The optimal strategy depends on your infrastructure. "Local storage" performance characteristics vary between bare metal, virtual machines, and container orchestration platforms.

Comprehensive Persistence Example

Mount multiple directories for complete data, backup, log, and configuration persistence:

```
$ docker run --rm -p 2480:2480 --name my_arcadedb \
    -v /path/on/host/databases:/home/arcadedb/databases \
    -v /path/on/host/backups:/home/arcadedb/backups \
    -v /path/on/host/log:/home/arcadedb/log \
    -v /path/on/host/config:/home/arcadedb/config \
    --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata" \
    --hostname my_arcadedb arcadedata/arcadedb:26.11.1
```

Image user and volumes

(Since v26.11.1) The JVM image and both native images (`linux/amd64` and `linux/arm64`) run as `1000:1000`, and their `bin`, `config`, `databases`, `backups`, `replication` and `log` folders are owned by that user, so no `--user` is needed and a volume written by one image can be opened by another. Before, the amd64 native image ran as root and the arm64 native image as UID 65532 (`nonroot`). A bind mount keeps the ownership of the host directory and must be writable by UID 1000.

An existing volume or bind mount written by an earlier native image is owned by root (amd64) or 65532 (arm64), and the new image cannot write to it. Re-own it once before upgrading:

```
$ chown -R 1000:1000 <volume or bind-mount path>
```

The image declares five volumes: `config`, `databases`, `replication`, `backups` and `log`. Mount all five to keep the users, groups and API tokens the server writes to `config`, and the HA state in `replication`, across container recreation. The `docker-compose.yml` quickstart mounts a named volume on each of them (the `databases` volume keeps its name, `arcade_db_vol`, so existing databases stay where they are); before v26.11.1 it mounted a named volume on `databases` only, so users, groups and API tokens created at runtime were lost on `docker compose down -v`. To keep them, copy them out of the old container before recreating the stack, then into the new one, and restart the service:

```
$ docker cp <old container>:/home/arcadedb/config/. ./config-backup
$ docker cp ./config-backup/. <new container>:/home/arcadedb/config
```

A custom distribution image generated by `arcadedb-builder.sh` follows the same rules since v26.11.1: it runs as `1000:1000` and declares the same five volumes (before, it ran as `100:101` and

declared only `databases`, `backups` and `log`). A volume written by an earlier custom image must be re-owned once with `chown -R 1000:1000` on `config`, `databases`, `replication`, `backups` and `log`. In a custom image, a named volume mounted on `config` is seeded with the image's default configuration files on first use; a bind mount on `config` starts out empty and hides them. Once `config` holds state it is not refreshed from the image, so a default file added by a later release does not appear in an existing volume.

Relocating the configuration directory

The server keeps its configuration files in `/home/arcadedb/config`: `server-configuration.json`, the users, groups and API tokens (`server-users.jsonl`, `server-groups.json`, `server-api-tokens.json`), `backup.json`, `ai.json`, `mcp-config.json` and `gremlin-server.yaml`. You can move them with `arcadedb.server.configDirectory`, for example onto the volume that already holds the databases, so a single mount persists both:

```
$ docker run --rm -p 2480:2480 --name my_arcadedb \
    -v /path/on/host/arcadedb:/data \
    --env ARCADEDB_SETTINGS="-Darcadedb.server.databaseDirectory=/data/databases \
                    -Darcadedb.server.configDirectory=/data/config \
                    -Darcadedb.server.rootPassword=playwithdata" \
    --hostname my_arcadedb arcadedata/arcadedb:26.11.1
```

Keep in mind:

- Pass the setting as a `-D` property (here through `ARCADEDB_SETTINGS`). Setting it in `server-configuration.json` has no effect, because that file is itself read from the configuration directory.
- The server creates the directory if it does not exist. *(Since v26.11.1)* A directory that holds no file yet is filled with the shipped defaults the first time the server starts on it: `server-groups.json`, `gremlin-server.yaml`, `gremlin-server.groovy` and `gremlin-server.properties`. They come from the server jar, so this works on the native images and on a Kubernetes volume claim, which is never populated from the image. A directory that already holds a file is not touched: edits are never overwritten, a file deleted on purpose does not come back, and defaults that a later release adds do not reach a directory already in use, so merge them by hand after an upgrade. Before v26.11.1 the server copied nothing and you had to copy them over yourself.
- Logging is not affected: `arcadedb-log.properties` is still loaded through `-Djava.util.logging.config.file`.

2.4.3. Tuning

The image sizes the JVM heap as a share of the container's memory limit (`-XX:MaxRAMPercentage=75`), so in most cases there is nothing to tune: give the container the memory you want ArcadeDB to have and the heap follows. The image also selects generational ZGC through `ARCADEDB_OPTS_GC`. See JVM settings inside a fixed memory budget for how the heap share and the collector are chosen, and why a heap pinned with `-Xms` equal to `-Xmx` inflates the memory the container reports.

```
$ docker run -m 4g ... arcadedata/arcadedb:latest
```

The heap is sized from the container limit. Earlier images pinned it to 2 GB (`-Xms2G -Xmx2G`), which meant a container limited to less than about 2.3 GB was killed at startup before the server came up.

To set the heap explicitly instead, override `ARCADEDB_OPTS_MEMORY`:

```
$ docker ... -e ARCADEDB_OPTS_MEMORY="-Xms800M -Xmx800M" ...
```

To run ArcadeDB with RAM <800M, it's suggested to tune some settings. You can use the `low-ram` profile to use the least memory possible.

```
$ docker ... -e ARCADEDB_OPTS_MEMORY="-Xms800M -Xmx800M" -e ARCADEDB_SETTINGS="-
Darcadedb.profile=low-ram" ...
```

Chapter 3. Examples

3.1. Vector Search Tutorial

This tutorial walks you through building a semantic search system with ArcadeDB. You will create vector embeddings, index them, query by similarity, and combine vector search with graph traversal.

3.1.1. What You Will Build

A product catalog with semantic search: given a query like "portable computing device", find the most relevant products by embedding similarity rather than keyword matching.

3.1.2. Prerequisites

- ArcadeDB running (Docker or binary install)
- A way to send queries (Console, HTTP API, or a Python/JavaScript client)

3.1.3. Step 1: Create the Schema

Create a vertex type with a vector property:

```
CREATE VERTEX TYPE Product
CREATE PROPERTY Product.name STRING
CREATE PROPERTY Product.category STRING
CREATE PROPERTY Product.embedding LIST OF FLOAT
```

3.1.4. Step 2: Create a Vector Index

Create an LSM_VECTOR index on the embedding property. Specify the number of dimensions and the similarity metric:

```
CREATE INDEX ON Product (embedding) LSM_VECTOR METADATA {
  dimensions: 4,
  similarity: 'COSINE'
}
```

In production, embeddings are typically 384-1536 dimensions. This tutorial uses 4 dimensions for simplicity. For production workloads, add `quantization: 'INT8'` for significantly better search performance — see concepts/vector-search.pdf below.

3.1.5. Step 3: Insert Data with Embeddings

Insert products with pre-computed embedding vectors:

```
CREATE VERTEX Product SET name = 'Laptop',       category = 'Electronics', embedding =
[0.9, 0.1, 0.8, 0.2]
CREATE VERTEX Product SET name = 'Tablet',        category = 'Electronics', embedding
= [0.85, 0.15, 0.75, 0.25]
CREATE VERTEX Product SET name = 'Smartphone',    category = 'Electronics', embedding
= [0.8, 0.2, 0.7, 0.3]
CREATE VERTEX Product SET name = 'Headphones',    category = 'Electronics', embedding
= [0.6, 0.4, 0.5, 0.5]
CREATE VERTEX Product SET name = 'Novel',         category = 'Books',       embedding
= [0.1, 0.9, 0.2, 0.8]
CREATE VERTEX Product SET name = 'Textbook',      category = 'Books',       embedding
= [0.2, 0.8, 0.3, 0.7]
CREATE VERTEX Product SET name = 'Running Shoes', category = 'Sports',      embedding
= [0.3, 0.5, 0.9, 0.1]
CREATE VERTEX Product SET name = 'Yoga Mat',      category = 'Sports',      embedding
= [0.25, 0.55, 0.85, 0.15]
```

In a real application, you would generate embeddings using an external model such as OpenAI's `text-embedding-3-small` (1536 dimensions) or Sentence Transformers' `all-MiniLM-L6-v2` (384 dimensions).

3.1.6. Step 4: Query by Similarity

Find the 3 products most similar to a query vector:

```
SELECT name, category, distance FROM (
  SELECT expand(vector.neighbors('Product[embedding]', [0.88, 0.12, 0.78, 0.22], 3))
)
```

The `vector.neighbors()` function returns a list of results — `expand()` flattens it into individual rows so you can access properties like `name`, `category`, and `distance` directly. The query vector `[0.88, 0.12, 0.78, 0.22]` is close to the electronics cluster. The results should return Laptop, Tablet, and Smartphone — the three most similar items by cosine similarity.

3.1.7. Step 5: Add Graph Relationships

Make the example more interesting by adding edges between products:

```
CREATE EDGE TYPE FREQUENTLY_BOUGHT_WITH
CREATE EDGE TYPE SIMILAR_TO

CREATE EDGE FREQUENTLY_BOUGHT_WITH
  FROM (SELECT FROM Product WHERE name = 'Laptop')
  TO (SELECT FROM Product WHERE name = 'Headphones')

CREATE EDGE SIMILAR_TO
  FROM (SELECT FROM Product WHERE name = 'Laptop')
```

```
  TO (SELECT FROM Product WHERE name = 'Tablet')

CREATE EDGE FREQUENTLY_BOUGHT_WITH
  FROM (SELECT FROM Product WHERE name = 'Novel')
  TO (SELECT FROM Product WHERE name = 'Textbook')
```

3.1.8. Step 6: Combine Vector Search with Graph Traversal

Find similar products, then expand recommendations through graph relationships:

```
-- Step 1: Find top 3 by vector similarity
SELECT name, category, distance FROM (
  SELECT expand(vector.neighbors('Product[embedding]', [0.88, 0.12, 0.78, 0.22], 3))
)
```

Then traverse from those results to find related products:

```
-- Step 2: Get products frequently bought with the top match
SELECT friend.name, friend.category
FROM MATCH {type: Product, as: product, where: (name = 'Laptop')}
     -FREQUENTLY_BOUGHT_WITH-> {type: Product, as: friend}
```

This two-step pattern — vector search to find semantically similar items, then graph traversal to expand through relationships — is the foundation of the Graph RAG and Recommendation Engine patterns.

3.1.9. Step 7: Use Different Similarity Metrics

Create additional indexes with different metrics for comparison:

```
-- Euclidean distance (absolute distance in vector space)
CREATE PROPERTY Product.embedding_l2 LIST OF FLOAT
CREATE INDEX ON Product (embedding_l2) LSM_VECTOR METADATA {
  dimensions: 4,
  similarity: 'EUCLIDEAN'
}

-- Dot product (fastest for normalized vectors)
CREATE PROPERTY Product.embedding_dot LIST OF FLOAT
CREATE INDEX ON Product (embedding_dot) LSM_VECTOR METADATA {
  dimensions: 4,
  similarity: 'DOT_PRODUCT'
}
```

3.1.10. Step 8: Control Search Quality with efSearch

By default, ArcadeDB sizes the search beam from the graph and `k` (see concepts/vector-search.pdf). If recall matters, override it per-query by passing efSearch as the 4th argument:

```
-- Higher efSearch for better recall (useful for critical queries)
SELECT name, category, distance FROM (
  SELECT expand(vector.neighbors('Product[embedding]', [0.88, 0.12, 0.78, 0.22], 3,
200))
)
```

See Adaptive efSearch for details on how the default strategy works.

3.1.11. Step 9: Enable Quantization for Large Datasets

For production datasets with many vectors, enable INT8 quantization to reduce memory by 75%:

```
CREATE VERTEX TYPE LargeProduct
CREATE PROPERTY LargeProduct.embedding ARRAY_OF_FLOATS

CREATE INDEX ON LargeProduct (embedding) LSM_VECTOR METADATA {
  dimensions: 384,
  similarity: 'COSINE',
  quantization: 'INT8'
}
```

Queries work exactly the same — quantization is transparent:

```
SELECT name, distance FROM (
  SELECT expand(vector.neighbors('LargeProduct[embedding]', :queryVector, 10))
)
```

3.1.12. Next Steps

- Vector Search Concepts — Architecture, algorithms, and parameter tuning
- Vector Embeddings How-To — Production best practices
- Recommendation Engine — Full use case with vector + graph + time-series
- Graph RAG — Vector + graph for LLM retrieval augmentation

3.2. Time Series Tutorial

This tutorial walks you through ingesting and querying time series data with ArcadeDB. You will create a sensor monitoring system that ingests temperature readings and queries them with SQL aggregations, PromQL, and continuous aggregates.

3.2.1. Prerequisites

- ArcadeDB running (Docker or binary install)
- `curl` for HTTP requests

3.2.2. Step 1: Create a TimeSeries Type

Create a type to store sensor readings with temperature, humidity, and pressure fields:

```
CREATE TIMESERIES TYPE SensorReading
  TIMESTAMP ts PRECISION MILLISECOND
  TAGS (sensor_id STRING, location STRING)
  FIELDS (
    temperature DOUBLE,
    humidity DOUBLE,
    pressure DOUBLE
  )
  SHARDS 4
```

- **TIMESTAMP** — Every time series type needs a timestamp column
- **TAGS** — Low-cardinality dimensions used for filtering (indexed automatically)
- **FIELDS** — The measurement values
- **SHARDS** — Parallel write/read partitions (default: CPU count)

3.2.3. Step 2: Ingest Data via InfluxDB Line Protocol

The fastest remote ingestion method. Send sensor readings using `curl`:

```
curl -X POST "http://localhost:2480/api/v1/ts/mydb/write?precision=ms" \
  -u root:arcadedb \
  -H "Content-Type: text/plain" \
  --data-binary '
SensorReading,sensor_id=sensor-A,location=floor-1
temperature=22.5,humidity=65.0,pressure=1013.2 1708430400000
SensorReading,sensor_id=sensor-A,location=floor-1
temperature=22.7,humidity=64.5,pressure=1013.1 1708430460000
SensorReading,sensor_id=sensor-A,location=floor-1
temperature=23.1,humidity=63.0,pressure=1013.0 1708430520000
SensorReading,sensor_id=sensor-B,location=floor-2
temperature=19.8,humidity=70.0,pressure=1012.5 1708430400000
SensorReading,sensor_id=sensor-B,location=floor-2
temperature=19.5,humidity=71.0,pressure=1012.6 1708430460000
SensorReading,sensor_id=sensor-B,location=floor-2
temperature=19.2,humidity=72.0,pressure=1012.7 1708430520000
'
```

The format is: `<type>,<tag>=<value> <field>=<value> <timestamp>`

3.2.4. Step 3: Ingest Data via SQL

You can also insert using standard SQL. The timestamp is built with the `date()` function (an ISO-8601 string is not accepted by `INSERT`, while it is in `WHERE` conditions); an epoch value in milliseconds works too:

```
INSERT INTO SensorReading
  (ts, sensor_id, location, temperature, humidity, pressure)
  VALUES
    (date('2026-02-20 10:10:00', 'yyyy-MM-dd HH:mm:ss'), 'sensor-A', 'floor-1', 23.5,
62.0, 1012.9),
    (date('2026-02-20 10:10:00', 'yyyy-MM-dd HH:mm:ss'), 'sensor-B', 'floor-2', 19.0,
73.0, 1012.8)
```

3.2.5. Step 4: Query with Time Range Filters

Time range conditions on the timestamp column are pushed down to the storage engine for efficient scans:

```
SELECT ts, sensor_id, temperature, humidity
FROM SensorReading
WHERE ts BETWEEN '2026-02-20T10:00:00Z' AND '2026-02-20T11:00:00Z'
  AND sensor_id = 'sensor-A'
ORDER BY ts
```

3.2.6. Step 5: Aggregate with Time Bucketing

Use `ts.timeBucket()` to group data into time intervals:

```
SELECT ts.timeBucket('1h', ts) AS hour,
       sensor_id,
       avg(temperature) AS avg_temp,
       max(temperature) AS max_temp,
       min(temperature) AS min_temp,
       count(*) AS samples
FROM SensorReading
WHERE ts BETWEEN '2026-02-20' AND '2026-02-21'
GROUP BY hour, sensor_id
ORDER BY hour
```

3.2.7. Step 6: Calculate Rates and Percentiles

```
-- Rate of change per minute
SELECT ts.timeBucket('5m', ts) AS window,
```

```
       sensor_id,
       ts.rate(temperature, ts) AS temp_change_per_sec
FROM SensorReading
GROUP BY window, sensor_id

-- 99th percentile temperature per hour
SELECT ts.timeBucket('1h', ts) AS hour,
       ts.percentile(temperature, 0.99) AS p99_temp
FROM SensorReading
GROUP BY hour
```

3.2.8. Step 7: Fill Gaps with Interpolation

Sensor data often has gaps. Use `ts.interpolate()` to fill them:

```
SELECT ts.timeBucket('1m', ts) AS minute,
       ts.interpolate(temperature, 'linear', ts) AS temp
FROM SensorReading
WHERE sensor_id = 'sensor-A'
  AND ts BETWEEN '2026-02-20T10:00:00Z' AND '2026-02-20T11:00:00Z'
GROUP BY minute
```

Methods: `'zero'` (fill with 0), `'prev'` (forward fill), `'linear'` (interpolate), `'none'` (keep null).

3.2.9. Step 8: Query with PromQL

ArcadeDB includes a native PromQL evaluator. Query via HTTP:

```
# Instant query
curl
"http://localhost:2480/api/v1/ts/mydb/prom/api/v1/query?query=avg(SensorReading{sensor
_id='sensor-A'})" \
  -u root:arcadedb

# Range query with rate calculation
curl "http://localhost:2480/api/v1/ts/mydb/prom/api/v1/query_range?\
query=rate(SensorReading{sensor_id='sensor-A'}[5m])&\
start=1708430400&end=1708434000&step=60" \
  -u root:arcadedb
```

Or use PromQL from within SQL:

```
RETURN promql('avg(SensorReading{sensor_id="sensor-A"})')
```

3.2.10. Step 9: Create a Continuous Aggregate

Pre-compute hourly summaries that update automatically:

```
CREATE CONTINUOUS AGGREGATE hourly_temps AS
  SELECT ts.timeBucket('1h', ts) AS hour,
         sensor_id,
         avg(temperature) AS avg_temp,
         max(temperature) AS max_temp,
         count(*) AS cnt
  FROM SensorReading
  GROUP BY hour, sensor_id
```

Query the aggregate like any table:

```
SELECT * FROM hourly_temps ORDER BY hour DESC LIMIT 10
```

New inserts into `SensorReading` automatically trigger incremental updates to the aggregate.

3.2.11. Step 10: Set Up Retention and Downsampling

Keep raw data for 90 days, then downsample:

```
ALTER TIMESERIES TYPE SensorReading
  ADD DOWNSAMPLING POLICY
    AFTER 7 DAYS GRANULARITY 1 MINUTES
    AFTER 30 DAYS GRANULARITY 1 HOURS
```

Raw data is automatically downsampled by a background scheduler.

3.2.12. Next Steps

- Time Series Concepts — Architecture, compression, HA, and full reference
- Realtime Analytics Use Case — Complete IoT monitoring with Grafana
- Time Series Quick Reference — Syntax cheat sheet

3.3. Recommendation Engine

Build intelligent product and content recommendations in a single multi-model database — no external recommendation service required. Graph traversal powers collaborative filtering through relationship patterns like User-PURCHASED→Product and User-WATCHED→Show, vector similarity drives content-based recommendations using `vector.neighbors()` with LSM_VECTOR indexes on product and show embeddings, and time-series analysis detects trending items by aggregating interaction counts over time windows.

3.3.1. Architecture Overview

Vertices	`User`, `Product`, `Show` (each with embedding vectors)
Edges	`PURCHASED`, `WATCHED`, `INTERACTED`
Documents	`ProductInteraction` (productId, purchaseCount, timestamp)

Users connect to products and shows through purchase, watch, and interaction edges. Each item carries a 4-dimensional embedding vector for content-based similarity. A `ProductInteraction` document type tracks interaction counts over time for trending analysis.

3.3.2. Key Queries

Collaborative Filtering — Find products purchased by users who bought the same items:

```
MATCH (u:User)-[:PURCHASED]->(p:Product)<-[:PURCHASED]-(other:User)-[:PURCHASED]->
(rec:Product)
WHERE u.name = 'Alice' AND NOT (u)-[:PURCHASED]->(rec)
RETURN rec.name, count(other) AS score ORDER BY score DESC
```

Vector Similarity Search — Find products similar to a given product by embedding:

```
SELECT name, category, distance FROM (
  SELECT expand(vector.neighbors('Product[embedding]', [0.9, 0.1, 0.8, 0.2], 5))
) ORDER BY distance
```

Diversified Recommendations Across Categories — Avoid result-page collapse to a single category by asking for the best matching product per category in one call:

```
SELECT expand(`vector.neighbors`(
    'Product[embedding]',
    [0.9, 0.1, 0.8, 0.2],
    10,
    { groupBy: 'category', groupSize: 1 }
))
```

`groupBy` is integrated with the index lookup, so a 10-category-wide carousel does not require over-fetching 50 or 100 candidates at the application layer just to dedupe afterwards. `groupSize: 2` would return up to two products per category for a "browse more like this" surface.

Trending Detection — Identify trending products by recent interaction volume:

```
SELECT productId, SUM(purchaseCount) AS total
FROM ProductInteraction
WHERE timestamp > date('2024-01-14', 'yyyy-MM-dd')
```

```
GROUP BY productId ORDER BY total DESC
```

3.3.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/recommendation-engine
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: recommendation-engine on GitHub

3.3.4. Related Documentation

- Vector Search Concepts
- Graph Model
- Time Series

3.4. Knowledge Graphs

Build a unified academic research system that integrates researchers, papers, institutions, and topics in a single database. Graph traversal drives multi-hop queries across co-authorship and citation networks, vector similarity enables semantic paper search via `vector.neighbors()` on paper embeddings, full-text search supports keyword queries on abstracts using `SEARCH_INDEX()`, and time-series tracking monitors citation activity over time.

3.4.1. Architecture Overview

Vertices	`Researcher`, `Paper`, `Topic`, `Institution`
Edges	`CO_AUTHORED`, `CITES`, `COVERS`, `AFFILIATED_WITH`
Documents	`PaperActivity` (paperId, citationCount, timestamp)

Papers carry 4-dimensional embedding vectors for semantic search and full-text indexed abstracts for keyword search. Citation activity is tracked as time-series documents.

3.4.2. Key Queries

Co-authorship Network — Discover collaborations between researchers:

```
MATCH (r:Researcher)-[:CO_AUTHORED]->(p:Paper)<-[:CO_AUTHORED]-(coauthor:Researcher)
WHERE r.name = 'Dr. Smith'
RETURN coauthor.name, collect(p.title) AS papers
```

Semantic Paper Search — Find papers similar to a research topic by embedding:

```
SELECT title, year, distance FROM (
  SELECT expand(vector.neighbors('Paper[embedding]', [0.8, 0.3, 0.7, 0.1], 5))
) ORDER BY distance
```

Full-Text Abstract Search — Keyword search across paper abstracts:

```
SELECT title, abstract FROM Paper
WHERE SEARCH_INDEX('Paper[abstract]', 'machine learning graph')
```

Hybrid Semantic + Keyword Search, one query — Fuse the semantic-similarity ranking with the abstract keyword match into a single ranked list, then collapse to one paper per topic so the literature review surface does not over-represent a single research thread:

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Paper[embedding]', [0.8, 0.3, 0.7, 0.1], 50),
    (SELECT @rid, $score FROM Paper
     WHERE SEARCH_INDEX('Paper[abstract]', 'machine learning graph') = true),
    { fusion: 'RRF', groupBy: 'topic', groupSize: 1 }
)) LIMIT 10
```

The semantic side surfaces conceptually similar papers even when the abstract phrasing diverges; the full-text side anchors the result on papers that explicitly use the query terms; RRF balances the two without requiring score calibration. Replace `groupBy: 'topic'` with `'institution'`, `'venue'`, or `'year'` to diversify along a different axis. To dedupe across multiple authors of the same paper, the upstream sparse vector index can be added as a third source for term-level disambiguation.

3.4.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/knowledge-graphs
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: knowledge-graphs on GitHub

3.4.4. Related Documentation

- Vector Search Concepts
- Full-Text Search
- Graph Model

3.5. Graph RAG

Implement retrieval-augmented generation (RAG) that retrieves richer, more connected context for LLM queries — all within a single database. Graph traversal enables multi-hop entity bridging across knowledge graph relationships, vector similarity powers semantic chunk retrieval using `vector.neighbors()` with LSM_VECTOR indexes, sparse vectors add SPLADE / BM25-style learned sparse retrieval for exact-term and OOV matches that dense embeddings miss, server-side hybrid fusion combines all of the above in one query, full-text search provides keyword-based chunk lookup, and Neo4j Bolt protocol compatibility on port 7687 supports LangChain4j integration.

3.5.1. Architecture Overview

Vertices	`Chunk` (content, source, chunkIndex, embedding, tokens, weights), `Entity`, `Person`, `Concept`, `Organization`
Edges	`MENTIONS`, `RELATES_TO`, `WORKS_AT`, `AUTHORED`

Document chunks carry both a dense embedding (`embedding`) and a sparse vector (`tokens` + `weights`) so the same record participates in both retrieval modalities. Chunks link to extracted entities through `MENTIONS` edges, and entities connect via `RELATES_TO`, enabling multi-hop discovery that bridges chunks from different documents through shared entity mentions.

3.5.2. Key Queries

Vector Search — Find semantically similar chunks by dense embedding:

```
SELECT content, source, distance FROM (
  SELECT expand(vector.neighbors('Chunk[embedding]', [0.9, 0.1, 0.8, 0.2], 5))
)
```

Hybrid Retrieval (Dense + Sparse), one query — The canonical modern-RAG retrieval shape. Dense + sparse hybrid with Reciprocal Rank Fusion, deduped to one chunk per source document via `groupBy`:

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Chunk[embedding]', :denseQuery, 50),
    `vector.sparseNeighbors`('Chunk[tokens,weights]', :qIdx, :qVal, 50),
    { fusion: 'RRF', groupBy: 'source', groupSize: 1 }
)) LIMIT 10
```

Sparse retrieval fixes the case where dense embeddings collapse semantically distinct queries onto the same point: "freeze card after theft" and "unfreeze card after travel" share the same dense vector but separate cleanly under sparse term matching. `groupBy: 'source', groupSize: 1` ensures each source document contributes its single best matching chunk to the LLM context window rather than overwhelming the prompt with near-duplicate chunks from one document.

Multi-Hop Entity Bridge — Discover related entities across documents:

```
MATCH (c:Chunk)-[:MENTIONS]->(e:Entity)-[:RELATES_TO*1..2]-(related:Entity)<-
[:MENTIONS]-(other:Chunk)
WHERE c.source = 'quantum_computing.txt'
RETURN related.name, other.content, other.source
```

Composite Scoring — Combine vector distance with graph connectivity for ranked retrieval:

```
SELECT content, source,
  (1.0 / (1.0 + distance)) * 0.7 + (entityCount / 5.0) * 0.3 AS compositeScore
FROM ChunkScores ORDER BY compositeScore DESC
```

3.5.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/graph-rag
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: graph-rag on GitHub

3.5.4. Related Documentation

- Vector Search Concepts
- Graph Model
- Full-Text Search and BM25 Ranking

3.6. Fraud Detection

Detect financial fraud by unifying four detection signals in a single multi-model database: graph traversal exposes organized fraud rings through shared identifier patterns such as devices, phones, and addresses; vector similarity flags behavioral anomalies using `vector.cosineSimilarity()` on transaction embeddings; time-series analysis catches velocity attacks through temporal transaction patterns; and document queries resolve synthetic identities by detecting duplicate SSNs.

3.6.1. Architecture Overview

Vertices	`Account`, `Customer`, `Device`, `Phone`, `Address`, `Email`, `Beneficiary`, `Transaction`
Edges	`USES_DEVICE`, `HAS_PHONE`, `HAS_ADDRESS`, `HAS_EMAIL`, `TRANSFERRED_TO`, `BENEFICIARY_OF`

Accounts and customers connect through shared devices, phones, and addresses. Overlapping identifiers reveal fraud rings. Customers carry `profile_embedding` vectors and transactions carry `behavior_embedding` vectors for anomaly detection.

3.6.2. Key Queries

Fraud Ring Detection — Identify accounts sharing devices or phones:

```
MATCH (a1:Account)-[:USES_DEVICE]->(d:Device)<-[:USES_DEVICE]-(a2:Account)
WHERE a1 <> a2
RETURN a1.name, a2.name, d.deviceId AS sharedDevice
```

Synthetic Identity Detection — Find accounts sharing SSNs:

```
SELECT a1.name, a2.name, a1.ssn
FROM Account a1, Account a2
WHERE a1.ssn = a2.ssn AND a1 != a2
```

Behavioral Anomaly Detection — Compare transaction embeddings against baselines:

```
SELECT name, vector.cosineSimilarity(behavior_embedding, [0.1, 0.9, 0.1, 0.8]) AS
similarity
FROM Transaction
WHERE vector.cosineSimilarity(behavior_embedding, [0.1, 0.9, 0.1, 0.8]) < 0.7
```

3.6.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/fraud-detection
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: fraud-detection on GitHub

3.6.4. Related Documentation

- Graph Model
- Vector Search Concepts
- Time Series
- Cypher Query Language
- Louvain Community Detection — Identify fraud rings as communities
- Cycle Detection — Detect circular money flows
- Dijkstra Shortest Path — Trace shortest paths between suspicious accounts

3.7. Realtime Analytics

Replace a fragmented stack of specialized monitoring tools with a single database that integrates time-series sensor data, service metrics, and infrastructure topology. Time-series native functions — `ts.timeBucket()`, `ts.rate()`, `ts.percentile()`, and `ts.interpolate()` — power sensor and service metric analysis, while graph traversal maps building topology and service dependencies, enabling multi-model correlation that combines graph-based entity identification with time-series aggregation in unified queries.

3.7.1. Architecture Overview

Time Series	`SensorReading` (sensor_id, location, temperature, humidity, pressure), `ServiceMetrics` (service_id, server_id, request_count, error_count, latency_ms)
Vertices	`Building`, `Floor`, `Sensor`, `Server`, `Service`
Edges	`HAS_FLOOR`, `INSTALLED_IN`, `RUNS_ON`, `DEPENDS_ON`

Sensors are installed on floors within buildings; services run on servers and depend on other services. Time-series data links to graph topology, enabling queries like "show the temperature trend for all sensors on the 2nd floor of Building A."

3.7.2. Key Queries

Hourly Temperature Bucketing — Aggregate sensor readings by hour:

```
SELECT sensor_id, ts.timeBucket(timestamp, 'PT1H') AS hour,
  AVG(temperature) AS avg_temp, MAX(temperature) AS max_temp
FROM SensorReading
GROUP BY sensor_id, hour ORDER BY hour
```

Service Request Rate — Calculate request rate per service:

```
SELECT service_id, ts.rate(timestamp, request_count, 'PT5M') AS req_per_5min
FROM ServiceMetrics
WHERE service_id = 'api-gateway'
```

Service Impact Analysis — Trace service dependencies for outage impact:

```
MATCH (s:Service {name: 'database'})<-[:DEPENDS_ON*1..3]-(dependent:Service)
RETURN dependent.name AS affected_service, length(path) AS hops
```

3.7.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
```

```
cd arcadedb-usecases/realtime-analytics
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: realtime-analytics on GitHub

3.7.4. Related Documentation

- Time Series
- Graph Model

3.8. Social Network Analytics

Analyze social network dynamics — trending content, user influence, viral propagation, and engagement metrics — in a unified platform. Materialized views with three refresh modes (PERIODIC for trending dashboards, INCREMENTAL for post counts on commit, and MANUAL for on-demand influence scores) pre-compute key analytics, graph traversal powers viral spread chain analysis and community overlap detection, time-series tracking captures engagement metrics as historical snapshots, and polyglot query support lets you use SQL for views and aggregations alongside OpenCypher for graph traversals.

3.8.1. Architecture Overview

Vertices	`User`, `Post`, `Topic`, `Group`
Edges	`FOLLOWS`, `CREATED`, `LIKED`, `SHARED`, `TAGGED`, `MEMBER_OF`
Documents	`EngagementMetric` (postId, likes, shares, comments, timestamp)

Users create and interact with posts, follow each other, and belong to groups. Engagement metrics are captured as time-series snapshots. Materialized views pre-compute trending scores, post counts, and influence rankings.

3.8.2. Key Queries

Trending Content Dashboard — Aggregate engagement into trending scores (via PERIODIC materialized view):

```
SELECT postId, SUM(likes + shares * 2 + comments * 3) AS trendingScore
FROM EngagementMetric
WHERE timestamp > date('2024-01-15', 'yyyy-MM-dd')
GROUP BY postId ORDER BY trendingScore DESC
```

Viral Spread Chain — Trace how content propagates through the network:

```
MATCH path = (origin:User)-[:CREATED]->(p:Post)<-[:SHARED]-(sharer:User)-[:FOLLOWS]->
```

```
(reached:User)
WHERE origin.name = 'Alice'
RETURN origin.name, sharer.name, reached.name, p.title
```

Community Overlap — Find users active in multiple groups:

```
MATCH (u:User)-[:MEMBER_OF]->(g1:Group), (u)-[:MEMBER_OF]->(g2:Group)
WHERE g1 <> g2
RETURN u.name, g1.name, g2.name
```

3.8.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/social-network-analytics
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: social-network-analytics on GitHub

3.8.4. Related Documentation

- Graph Model
- Time Series
- PageRank — Rank user influence in the network
- Louvain Community Detection — Discover user communities
- Betweenness Centrality — Identify bridge users connecting communities
- Label Propagation — Fast community assignment

3.9. Supply Chain

Achieve multi-tier supply chain visibility in a single database by combining graph traversal for multi-tier supplier discovery, blast radius analysis, and end-to-end batch traceability via variable-length path queries, vector similarity for alternative supplier sourcing using capability embeddings with `vector.neighbors()`, time-series analysis for delivery disruption detection via metric aggregation, and the PostgreSQL wire protocol for JavaScript connectivity.

3.9.1. Architecture Overview

Vertices	`Supplier`, `Component`, `Product`, `Warehouse`, `Customer`, `ShippingRoute`, `RawMaterial`
Edges	`SUPPLIES`, `CONTAINS`, `STORED_AT`, `SHIPS_VIA`, `SHIPPED_TO`, `ALTERNATIVE_FOR`, `ASSEMBLED_FROM`

Documents	`DeliveryMetric` (supplierId, deliveryTime, qualityScore, timestamp)

Suppliers carry 4-dimensional capability vectors. Components flow through a multi-tier graph from raw materials through assembly to distribution. Delivery metrics track supplier performance over time.

3.9.2. Key Queries

Multi-Tier Supplier Discovery — Find all suppliers up to 3 tiers deep:

```
MATCH path = (p:Product)-[:CONTAINS*1..3]->(c:Component)<-[:SUPPLIES]-(s:Supplier)
WHERE p.name = 'SmartWatch Pro'
RETURN s.name, c.name, length(path) AS tier
```

Blast Radius Analysis — Assess impact of a supplier going offline:

```
MATCH (s:Supplier {name: 'ChipCorp'})-[:SUPPLIES]->(c:Component)<-[:CONTAINS]-(
p:Product)
OPTIONAL MATCH (alt:Supplier)-[:SUPPLIES]->(c)
WHERE alt <> s
RETURN p.name, c.name, collect(alt.name) AS alternatives
```

Alternative Supplier Sourcing — Find suppliers with similar capabilities by embedding:

```
SELECT name, region, distance FROM (
  SELECT expand(vector.neighbors('Supplier[capability_vec]', [0.9, 0.8, 0.7, 0.6], 5))
)
```

3.9.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/supply-chain
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: supply-chain on GitHub

3.9.4. Related Documentation

- Graph Model
- Vector Search Concepts
- Time Series
- PostgreSQL Protocol

3.10. Identity & Access Management

Manage identity access across complex permission hierarchies while detecting security anomalies and maintaining compliance. Graph traversal resolves permissions through nested group/role hierarchies via SQL MATCH queries, vector similarity detects behavioral anomalies using access pattern embeddings with `vector.neighbors()`, time-series tracking captures access audit logs for SOX/GDPR compliance reporting, and the PostgreSQL wire protocol provides Python connectivity for integration.

3.10.1. Architecture Overview

Vertices	`Identity`, `Group`, `Role`, `Permission`, `Resource`, `Policy`
Edges	`MEMBER_OF`, `HAS_ROLE`, `GRANTS`, `APPLIES_TO`, `GOVERNED_BY`
Documents	`AccessLog` (identityEmail, resourceName, action, source_ip, recordedAt)

Identities belong to nested groups that hold roles. Roles grant permissions on resources. Policies govern resources for compliance. Access logs track all actions for audit trails. Identities carry 8-dimensional access pattern vectors for behavioral analysis.

3.10.2. Key Queries

Permission Resolution — Resolve all permissions for a user through group/role hierarchy:

```
SELECT identity.email, role.name AS role, permission.action, resource.name AS resource
FROM MATCH {type: Identity, as: identity, where: (email = 'alice@corp.com')}
  -MEMBER_OF->  {type: Group, as: grp}
  -HAS_ROLE->   {type: Role, as: role}
  -GRANTS->     {type: Permission, as: permission}
  -APPLIES_TO-> {type: Resource, as: resource}
```

Shadow Admin Detection — Find users with admin-equivalent access through indirect paths:

```
SELECT identity.email, role.name
FROM MATCH {type: Identity, as: identity}
  -MEMBER_OF-> {type: Group, as: grp, while: (true)}
  -HAS_ROLE->  {type: Role, as: role, where: (name LIKE '%admin%')}
```

Behavioral Anomaly Detection — Identify users with unusual access patterns:

```
SELECT email, distance FROM (
  SELECT expand(vector.neighbors('Identity[access_pattern_vec]', [0.1,0.9,0.1,0.8,0.2
,0.7,0.3,0.6], 3))
) WHERE distance > 0.5
```

3.10.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/iam
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: iam on GitHub

3.10.4. Related Documentation

- Graph Model
- Vector Search Concepts
- PostgreSQL Protocol
- BFS — Traverse permission hierarchies level by level
- All Simple Paths — Enumerate all permission paths to a resource
- Strongly Connected Components — Detect circular permission dependencies

3.11. Customer 360

Build a unified customer view in a single database by leveraging graph traversal for identity resolution, churn risk scoring, and cross-sell recommendations, the document model for fuzzy deduplication via cross-matching on customer attributes, vector similarity with LSM_VECTOR embeddings for customer preference and product matching, full-text search for support ticket content indexing, and time-series analysis for customer journey event chains and conversion path tracking.

3.11.1. Architecture Overview

Vertices	`Customer`, `Household`, `Product`, `Device`, `Address`, `Ticket`, `Campaign`, `Session`, `Event`, `Identifier`
Edges	`BELONGS_TO`, `PURCHASED`, `USES`, `LIVES_AT`, `OPENED`, `TARGETED_BY`, `STARTED`, `TRIGGERED`, `HAS_IDENTIFIER`

Customers connect to households, devices, addresses, and support tickets. Sessions contain events that trace customer journeys. Customers carry `prefVector` embeddings for personalization. Support tickets are full-text indexed.

3.11.2. Key Queries

Identity Resolution — Discover linked identities through shared sessions (3-hop transitive):

```
SELECT c1.name, c2.name, session.id AS sharedSession
```

```
FROM MATCH {type: Customer, as: c1}
  -STARTED->      {type: Session, as: session}
  <-STARTED-      {type: Customer, as: c2}
WHERE c1 <> c2
```

Churn Risk Scoring — Calculate churn risk based on churned neighbors in the social network:

```
SELECT customer.name,
  COUNT(CASE WHEN neighbor.status = 'churned' THEN 1 END) * 1.0 / COUNT(neighbor) AS
churnRatio
FROM MATCH {type: Customer, as: customer}
  -BELONGS_TO-> {type: Household}
  <-BELONGS_TO- {type: Customer, as: neighbor}
GROUP BY customer.name ORDER BY churnRatio DESC
```

Hybrid Preference + Ticket Search — Match a customer's preference vector against products while keeping support-ticket keyword evidence in the same ranked list. Useful for "next best action" surfaces that fold both behavioural similarity and recent issue context into one ranking, deduped to one product per category:

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Product[embedding]', :customerPrefVector, 50),
    (SELECT @rid, $score FROM Ticket WHERE SEARCH_INDEX('Ticket[content]',
:recentIssueTerms) = true),
    { fusion: 'RRF', groupBy: 'category', groupSize: 1 }
)) LIMIT 10
```

Customer Journey Analysis — Trace conversion paths from ad click to purchase:

```
MATCH (c:Customer)-[:STARTED]->(s:Session)-[:TRIGGERED]->(e1:Event)-[:TRIGGERED]->(
e2:Event)-[:TRIGGERED]->(e3:Event)
WHERE e1.type = 'ad_click' AND e3.type = 'purchase'
RETURN c.name, e1.type, e2.type, e3.type, s.channel
```

3.11.3. Try It Yourself

```
git clone https://github.com/ArcadeData/arcadedb-usecases.git
cd arcadedb-usecases/customer-360
docker compose up -d
./setup.sh
./queries/queries.sh
```

Full source: customer-360 on GitHub

3.11.4. Related Documentation

- Graph Model
- Document Model
- Vector Search Concepts
- Full-Text Search

3.12. Semantic Search (Planned)

This use case is in development. Want to contribute? See the arcadedb-usecases repository.

Standalone vector search for e-commerce product discovery and document retrieval. Demonstrates ArcadeDB's vector capabilities without requiring graph traversal — pure semantic search with filtering, faceting, and hybrid dense + sparse + keyword ranking.

3.12.1. Planned Features

- **Vector Similarity** — Product and document embeddings with LSM_VECTOR, HNSW, and DiskANN indexes
- **Sparse Vector Retrieval** — SPLADE / BGE-M3 learned-sparse embeddings via `LSM_SPARSE_VECTOR`
- **Server-side Hybrid Fusion** — `vector.fuse` combining dense + sparse + full-text in one query with RRF / DBSF / LINEAR strategies
- **BM25 Full-Text Search** — Lucene-backed keyword index with native BM25 ranking, plugged in as a `SEARCH_INDEX(⋯)` source of `vector.fuse` (see Hybrid Retrieval)
- **Document Model** — Faceted filtering on product attributes
- **Python** — Primary implementation language targeting data science and AI workflows

3.12.2. Reference Hybrid Query

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Product[dense]', :queryVec, 50),
    `vector.sparseNeighbors`('Product[tokens,weights]', :qIdx, :qVal, 50),
    (SELECT @rid, $score FROM Product WHERE SEARCH_INDEX('Product[name]', :keywords) =
true),
    { fusion: 'RRF', groupBy: 'category', groupSize: 1 }
)) LIMIT 20
```

This is the typical e-commerce shape: semantic similarity (dense embeddings) + exact-term matching (sparse) + keyword search (full-text), fused server-side, then diversified across product categories so the result page does not collapse to one category.

3.12.3. Target Scenarios

- E-commerce product search with natural language queries
- Document retrieval with semantic understanding
- Hybrid search combining keyword matching and dense+sparse vector similarity
- Multi-vector search (title embeddings + description embeddings + sparse term embeddings)

3.13. Geospatial Analytics (Planned)

This use case is in development. Want to contribute? See the arcadedb-usecases repository.

Fleet tracking and location-based services combining geospatial indexes with graph topology and time-series movement data. Demonstrates spatial queries, route optimization, and real-time vehicle tracking.

3.13.1. Planned Features

- **Geospatial** — Spatial indexes for point-in-polygon, nearest-neighbor, and distance queries
- **Graph Traversal** — Road network and delivery route graph modeling
- **Time Series** — Vehicle position tracking and movement analytics
- **Python and JavaScript** — Dual implementation for backend analytics and frontend visualization

3.13.2. Target Scenarios

- Fleet tracking with real-time position updates
- Nearest point-of-interest discovery
- Geofence-based alerting
- Route optimization through road network graphs

3.14. Content Management (Planned)

This use case is in development. Want to contribute? See the arcadedb-usecases repository.

A content management system combining document storage, full-text search, and graph-based taxonomies. Demonstrates how ArcadeDB replaces separate CMS databases with a single multi-model store for content, metadata, and navigation hierarchies.

3.14.1. Planned Features

- **Document Model** — Content storage with flexible schemas for articles, pages, and media

- **Full-Text Search** — Content indexing and search across all document types
- **Graph Traversal** — Category trees, tag taxonomies, and content relationship graphs
- **JavaScript/TypeScript** — Primary implementation for web application integration

3.14.2. Target Scenarios

- Hierarchical content organization with category graphs
- Full-text search across articles and documents
- Related content discovery via graph relationships
- Content versioning and publishing workflows

3.15. Network Monitoring (Planned)

This use case is in development. Want to contribute? See the arcadedb-usecases repository.

Network topology modeling with time-series metrics and Grafana dashboards. Demonstrates how ArcadeDB unifies network graph analysis with performance monitoring, replacing separate topology databases and time-series stores.

3.15.1. Planned Features

- **Graph Traversal** — Network topology modeling (routers, switches, links, VLANs) with path analysis
- **Time Series** — Interface metrics (bandwidth, latency, packet loss) with PromQL queries
- **Vector Similarity** — Anomaly detection on traffic pattern embeddings
- **Python** — Primary implementation for network operations tooling

3.15.2. Target Scenarios

- Network topology visualization and path analysis
- Interface utilization monitoring with Grafana
- Anomaly detection on traffic patterns
- Impact analysis for planned maintenance windows

3.16. Data Lineage (Planned)

This use case is in development. Want to contribute? See the arcadedb-usecases repository.

Data pipeline lineage tracking with graph traversal for impact analysis, compliance reporting, and debugging data quality issues. Demonstrates modeling ETL/ELT pipelines as directed graphs with transformation metadata.

3.16.1. Planned Features

- **Graph Traversal** — Pipeline DAG modeling (sources, transformations, destinations) with upstream/downstream analysis
- **Document Model** — Schema snapshots and transformation metadata storage
- **Time Series** — Pipeline execution history and data quality metrics
- **Python** — Primary implementation for data engineering workflows

3.16.2. Target Scenarios

- Upstream/downstream impact analysis for schema changes
- Data quality issue root cause tracing
- Compliance reporting (which data flows where)
- Pipeline execution monitoring and alerting

Chapter 4. Languages & Drivers

4.1. Languages & Drivers

Running Python in the same process as the database? See Python — Embedded Bindings.

4.2. ArcadeDB Java Tutorial

The Java API is the primary interface to ArcadeDB. Use it in **embedded** mode to run the database inside your JVM (no separate server process), or in **remote** mode to talk to a running ArcadeDB Server over HTTP.

Connection options for Java

From Java you can also reach an ArcadeDB Server over the wire protocol of your choice:

- **Neo4j BOLT** using the official `neo4j-java-driver` — run Cypher queries.
- **PostgreSQL wire protocol** using any JDBC PostgreSQL driver — run SQL.
- **HTTP/JSON API** using `java.net.http.HttpClient` — no driver required.

The tutorial below covers the embedded / native Java API. Per-protocol Java examples are coming soon.

4.2.1. Embedded

Add the following dependency to your Maven `pom.xml` (or the equivalent in Gradle / SBT):

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-engine</artifactId>
    <version>26.11.1</version>
</dependency>
```

Most of the API works in both synchronous and asynchronous modes. Synchronous calls execute on the current thread and return when finished — the easiest model for procedural code. Asynchronous calls schedule jobs onto a pool of worker threads (sized to the available cores by default), so a single-threaded application can still saturate every core. The asynchronous API is reached via `db.async()` — see the Async Database API page.

You can create a new database from scratch or open an existent one.

To start from scratch, let's create a new database. The entry point it's the `DatabaseFactory` class that allows to create and open a database.

```
DatabaseFactory databaseFactory = new DatabaseFactory("/databases/mydb");
```

Pass the path in the file system where you want the database to be stored. In this case a new directory 'mydb' will be created under the path `/databases/` of your file system. You can also use a relative path like `databases/mydb`.

A `DatabaseFactory` object doesn't hold the `Database` instances. It's up to you to close them once you have finished.

Create a new database

To create a new database from scratch, use the `.create()` method in `DatabaseFactory` class. If the database already exists, an exception is thrown.

Syntax:

```
DatabaseFactory databaseFactory = new DatabaseFactory("/databases/mydb");
try( Database db = databaseFactory.create(); ){
  // YOUR CODE
}
```

The database instance `db` is ready to be used inside the try block. The `Database` instance extends Java7 `AutoClosable` interface, that means the database is closed automatically when the Database variable reaches out of the scope.

Open an existent database

If you want to open an existent database, use the `open()` method instead:

```
DatabaseFactory databaseFactory = new DatabaseFactory("/databases/mydb");
try( Database db = databaseFactory.open(); ){
  // YOUR CODE
}
```

By default a database is open in `READ_WRITE` mode, but you can open it in `READ_ONLY` in this way:

```
databaseFactory.open(PaginatedFile.MODE.READ_ONLY);
```

Using `READ_ONLY` denys any changes to the database. This is the suggested method if you're going to execute reads and queries only. Or if you are opening a database from a read-only file system like a DVD or a shared read-only directory. By letting know to ArcadeDB that you're not changing the database, a lot of optimizations will be used, like in a distributed high-available configuration a REPLICA server could be used instead of the busy MASTER.

If you open a database in READ_ONLY mode, no lock file is created, so the same database could be

opened in READ_ONLY mode by another process at the same time. A database opened in READ_ONLY mode owns nothing of its directory, and writes nothing to it, not even when it is closed: closing it leaves `statistics.json`, `last-tx-id.bin`, the schema, the `.wal` files and the index files as they were, so closing it after another process committed cannot rewind that process's record counters *(since v26.11.1)*.

Write your first transaction

Either if you create or open a database, in order to use it, you have to execute your code inside a transaction, in this way:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( (tx) -> {
    // YOUR CODE HERE
  });
}
```

Using the database's auto-close and the `transaction()` method allows to forget to manage begin/commit/rollback/close operations like you would do with a normal DBMS. Anyway, you can control the transaction with explicit methods if you prefer. This code block is equivalent to the previous one:

```
Database db = databaseFactory.open();
try {
  db.begin();

  // YOUR CHANGES HERE

  db.commit();

} catch (Exception e) {
  db.rollback();
} finally {
  db.close();
}
```

Remember that every change in the database must be executed inside a transaction. ArcadeDB is a fully transactional DBMS, ACID compliant. The usage of transactions is like with a Relational DBMS: `.begin()` starts a new transaction and `.commit()` commits all the changes in the database unless there is an error (like a conflict on updating the same record), then the entire transaction will be automatically rollbacked and none of your changes will be in the database. In case you want to manually rollback the transaction at a certain point (like when you have an error in your application code), you can call `.rollback()`.

Once you have your database instance (in this tutorial the variable `db` is used), you can create/update/delete records and execute queries.

Write your first document object

Let's start now populating the database by creating our first document of type "Customer". What is a document? A Document is like a map of entries. They can be nested and entries can have different types of values, such as Strings, Integers, Floats, etc. You can think to a document like a JSON Document but it's stored in a binary form in the database. By the way, if you use JSON in your application, ArcadeDB provides easy API to convert a document to and from JSON.

In ArcadeDB it's mandatory to specify a type when you want tot create a document, a vertex or an edge.

Let's create the new document type "Customer" without any properties:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    // CREATE THE CUSTOMER TYPE
    db.getSchema().createDocumentType("Customer");
  });
}
```

Once the "Customer" type has been created, we can create our first document:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    // CREATE A CUSTOMER INSTANCE
    MutableDocument customer = db.newDocument("Customer");
    customer.set("name", "Jay");
    customer.set("surname", "Miner");
    customer.save(); // THE DOCUMENT IS SAVED IN THE DATABASE ONLY WHEN `.save()` IS
CALLED
  });
}
```

You can create types and records in the same transaction.

Execute a Query

Once we have our database populated, how to extract data from it? Simple, with a query. Example of executing a prepared query:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    ResultSet result = db.query("SQL", "select from V where age > ? and city = ?", 18,
"Melbourne");
    while (result.hasNext()) {
      Result record = result.next();
      System.out.println( "Found record, name = " + record.getProperty("name"));
    }
```

```
  });
}
```

The first parameter of the query method is the language to be used. In this case the common "SQL" is used. You can also use Gremlin or other language that will be supported in the future.

The prepared statement is cached in the database, so further executions will be faster than the first one. With prepared statements, the parameters can be passed in positional way, like in this case, or with a `Map<String,Object>` where the keys are the parameter names and the values the parameter values. A `Map` passed as the only positional argument is bound to the first `?` placeholder. To pass named parameters use the `query(language, text, Map)` overload. Example:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    Map<String,Object> parameters = new HashMap<>();
    parameters.put( "age", 18 );
    parameters.put( "city", "Melbourne" );

    ResultSet result = db.query("SQL", "select from V where age > :age and city =
:city", parameters);
    while (result.hasNext()) {
      Result record = result.next();
      System.out.println( "Found record, name = " + record.getProperty("name"));
    }
  });
}
```

By using a map, parameters are referenced by name (`:age` and `:city` in this example).

Create a Graph

Now that we're familiar with the most basic operations, let's see how to work with graphs. Before creating our vertices and edges, we have to create both vertex and edge types beforehand. In our example, we're going to create a minimal social network with "User" type for vertices and "IsFriendOf" to map the friendship relationship:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    // CREATE THE ACCOUNT TYPE
    db.getSchema().createVertexType("User");
    db.getSchema().createEdgeType("IsFriendOf");
  });
}
```

Now let's create two "Profile" vertices and let's connect them with the friendship relationship "IsFriendOf", like in the chart below:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    MutableVertex albert = db.newVertex("User").set("name", "Albert").set("lastName",
"Einstein").save();
    MutableVertex michelle = db.newVertex("User").set("name", "Michelle").set
("lastName", "Besso").save();
    albert.newEdge("IsFriendOf", michelle, true, "since", 2010);
  });
}
```

In the code snipped above, we have just created our first graph, made of 2 vertices and one edge that connects them. Vertices and documents are not persistent until you call the `save()` method. Note the 3rd parameter in the `newEdge()` method. It's telling to the Graph engine that we want a bidirectional edge. In this way, even if the direction is still from the "Albert" vertex to the "Michelle" vertex, we can traverse the edge from both sides. Use always bidirectional unless you want to avoid creating super-nodes when it's necessary to traverse only from one side. Note also that we stored a property "since = 2010" in the edge. That's right, edges can have properties like vertices.

Traverse the Graph

What do you do with a brand new graph? Traversing, of course!

You have basically three ways to do that (Java API, SQL, Apache Gremlin and Open Cypher) each one with its pros/cons:

	JVM Embedded API	SQL	Apache Gremlin	Cypher
Speed	* * *	* *	* *	* *
Flexibility	* * *	*	* *	* *
Embedded mode	Yes	Yes	Yes	Yes
Remote mode	Yes (via `RemoteImmutableVertex`)	Yes	Yes (through the Gremlin Server plugin)	Yes (through the Gremlin Server plugin)

When using the API, when the SQL and Apache Gremlin? The API is the very code based. You have total control on the query/traversal. With the SQL, you can combine the `SELECT` with the `MATCH` statement to create powerful traversals in a just few lines. You could use Apache Gremlin if you're coming from another GraphDB that supports this language.

Traverse via API

In order to start traversing a graph, you need your root vertex (in some cases you want to start from multiple root vertices). You can load your root vertex by its RID (Record ID), via the indexes properties or via a SQL query.

Loading a record by its RID it's the fastest way and the execution time remains constants with the growing of the database (algorithm complexity: `O(1)`). Example of lookup by RID:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    // #10:232 in our example is Albert Einstein's RID
    Vertex albert = db.lookupByRID( new RID(db, "#10:232"), true );
  });
}
```

In order to have a quick lookup, it's always suggested to create an index against one or multiple properties. In our case, we could index the properties "name" and "lastName" with 2 separate indexes, or indeed, creating a composite index with both properties. In this case the algorithm complexity is `O(LogN)`). Example:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    db.getSchema().createTypeIndex(SchemaImpl.INDEX_TYPE.LSM_TREE, false, "Profile",
new String[] { "name", "lastName" });
  });
}
```

Now we're able to load Michelle's vertex in a flash by using this:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    Vertex michelle = db.lookupByKey( "Profile", new String[]{"name", "lastName"}, new
String[]{"Michelle", "Besso"} );
  });
}
```

Remember that loading a record by its RID is always faster than looking up from an index. What about the query approach? ArcadeDB supports SQL, so try this:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    ResultSet result = db.query( "SQL", "select from Profile where name = ? and
lastName = ?", "Michelle", "Besso" );
    Vertex michelle = result.next();
  });
}
```

With the query approach, if an existent index is available, then it's automatically used, otherwise a scan is executed.

Now that we have loaded the root vertex in memory, we're ready to do some traversal. Before looking at the API, it's important to understand every edge has a direction: from vertex A to vertex B. In the example above, the direction of the friendship is from "Albert" to "Michelle". While in most of the cases the direction is important, sometimes, like with the friendship, it doesn't really matter

the direction because if A is friend with B, it's true also the opposite.

In our example, the relationship is `Albert ---Friend--> Michelle`. This means that if I want to retrieve all Albert's friends, I could start from the vertex "Albert" and traverse all the **outgoing** edges of type "IsFriendOf".

Instead, if I want to retrieve all Michelle's friends, I could start from Michelle as root vertex and traverse all the **incoming** edges.

In case the direction doesn't really matters (like with friendship), I could consider **both** outgoing and incoming.

So the basic traversal operations from one or more vertices, are:

- outgoing, expressed as `OUT`
- incoming, expressed as `IN`
- both, expressed as `BOTH`

In order to load Michelle's friends, this is the example by using API:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    Vertex michelle; // ALREADY LOADED VIA RID, KEYS OR SQL
    Iterable<Vertex> friends = michelle.getVertices(DIRECTION.IN, "IsFriendOf" );
  });
}
```

Instead, if I start from Albert's vertex, it would be:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    Vertex albert; // ALREADY LOADED VIA RID, KEYS OR SQL
    Iterable<Vertex> friends = albert.getVertices(DIRECTION.OUT, "IsFriendOf");
  });
}
```

The iterators returned by `getEdges()` and `getVertices()` support `Iterator.remove()`: it deletes the edge of the element the last `next()` returned, exactly like `edge.delete()` (the edge is disconnected from both vertices, and a record-backed edge's record is deleted with it). For `getVertices()` that is the edge leading to the vertex just returned; the vertex itself is not deleted.

Since 26.10.2, `remove()` behaves the same on every edge-list iterator, with or without an edge type filter, and reports a failure to the caller:

- a delete the current user is not allowed to perform throws `SecurityException` and leaves the edge in place. Earlier versions logged a WARNING and unlinked the edge from the vertex being iterated anyway;

- on a filtered iterator, such as `getEdges(DIRECTION.OUT, "IsFriendOf")`, earlier versions only unlinked the entry from the iterated vertex, leaving the edge record and the other vertex's link behind;
- calling `remove()` twice without a `next()` in between throws `IllegalStateException`, as the `Iterator` contract requires.

The iterators over a super-node's striped edge list do not support `remove()` and throw `UnsupportedOperationException`: delete the edge with `edge.delete()` instead.

Traverse via API (Remote)

When working with a `RemoteDatabase`, graph traversal uses the same `getVertices()` and `getEdges()` methods on the vertex object returned from a lookup. Each call translates to a SQL query executed over HTTP — there is no per-vertex round trip.

```
RemoteDatabase db = new RemoteDatabase("localhost", 2480, "school", "root",
"playwithdata");

// lookupByRID returns a RemoteImmutableVertex when connected remotely
Vertex albert = db.lookupByRID(new RID(db, "#1:0"), true);

// iterate all outgoing "IsFriendOf" neighbors -- automatically paginates
for (Vertex friend : albert.getVertices(Vertex.DIRECTION.OUT, "IsFriendOf")) {
    System.out.println(friend.getString("name"));
}
```

`getVertices()` and `getEdges()` on a remote vertex automatically paginate results in batches of 2,000 records per HTTP call (`RemoteImmutableVertex.DEFAULT_PAGE_SIZE`), so they work correctly even when the vertex has tens of thousands of neighbors without hitting the server result limit.

For a vertex with an extremely large number of edges (a *supernode*), you can tune the page size by casting to `RemoteImmutableVertex` and calling `getVerticesPaged()`:

```
RemoteImmutableVertex albert = (RemoteImmutableVertex) db.lookupByRID(new RID(db,
"#1:0"), true);

// fetch 5,000 neighbors per HTTP call instead of the default 2,000
for (Vertex friend : albert.getVerticesPaged(Vertex.DIRECTION.OUT, 5_000,
"IsFriendOf")) {
    System.out.println(friend.getString("name"));
}

// manual offset-based page (e.g. records 1001-2000)
for (Vertex friend : albert.getVertices(Vertex.DIRECTION.OUT, 1_000, 1_000,
"IsFriendOf")) {
    System.out.println(friend.getString("name"));
```

```
}
```

Each page uses a `SKIP/LIMIT` SQL clause on the server. For supernodes with millions of edges the edge list is re-scanned from the beginning on every page ($O(n^2)$ total). In that case, prefer a targeted SQL query with a server-side `WHERE` filter to narrow the result set before returning it.

Traverse via SQL

By using SQL, you can do the traversal by using SELECT:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    ResultSet friends = db.query( "SQL", "SELECT expand( out('IsFriendOf') ) FROM
Profile WHERE name = ? AND lastName = ?", "Michelle", "Besso" );
  });
}
```

Or with the more powerful MATCH statement:

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    ResultSet friends = db.query( "SQL", "MATCH {type: Profile, as: Profile, where:
(name = ? and lastName = ?)}.out('IsFriendOf') {as: Friend} RETURN Friend",
"Michelle", "Besso" );
  });
}
```

Traverse via Apache Gremlin

Since ArcadeDB is 100% compliant with Gremlin 3.7.x, you can run this query against the Apache Gremlin Server configured with ArcadeDB:

```
g.V().has('name','Michelle').has('lastName','Besso').out('IsFriendOf');
```

For more information about Apache Gremlin see: Gremlin API support

Traverse via Open Cypher

ArcadeDB supports also Open Cypher. The same query would be the following:

```
MATCH (me)-[:IsFriendOf]-(friend)
WHERE me.name = 'Michelle' and me.lastName = 'Besso'
RETURN friend.name, friend.lastName
```

For more information about Cypher see: Cypher support

4.2.2. Remote

When ArcadeDB runs as a separate server process, your Java application can connect over HTTP/JSON via the `RemoteDatabase` driver, which wraps the same `Database` API you use in embedded mode.

Add the network module alongside the engine:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-engine</artifactId>
    <version>26.11.1</version>
</dependency>
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-network</artifactId>
    <version>26.11.1</version>
</dependency>
```

Create the database on the server, then open a `RemoteDatabase` and run a query:

```
RemoteServer server = new RemoteServer("localhost", 2480, "root", "playwithdata");
if (!server.exists("school"))
    server.create("school");

RemoteDatabase db = new RemoteDatabase("localhost", 2480, "school", "root",
"playwithdata");

db.command("sqlscript", """
    create document type Class if not exists;
    create property Class.name if not exists string;
    """);

db.command("sql",
    "insert into Class set name = :name, location = :location",
    Map.of("name", "English", "location", "3rd floor"));

ResultSet rs = db.query("sql", "select from Class");
rs.forEachRemaining(r -> System.out.println(r.toJSON()));
```

`RemoteDatabase` is **not** thread-safe — it carries a backend session, and a thread closing a transaction also closes the session. Don't share an instance across threads; instead create one inside the method scope where you use it.

For the full reference covering connection pooling, async calls, gRPC, and every server endpoint, see the Remote API and HTTP/JSON API pages.

4.3. Database API

The two core classes of the embedded Java API are:

- `DatabaseFactory` — entry point that creates and opens databases.
- `Database` — main handle for transactions, queries, schema, and CRUD operations.

4.3.1. `DatabaseFactory` Class

It's the entry point class that allows to create and open a database. A `DatabaseFactory` object doesn't keep any state and its only goal is creating a `Database` instance.

Never open the same database directory from two processes at once (for example, an embedded tool and a running ArcadeDB Server). Doing so can corrupt the database. If a server already has a database open, connect to it remotely instead of opening it with `DatabaseFactory`.

Example:

```
DatabaseFactory factory = new DatabaseFactory("/databases/mydb");
```

4.3.2. close()

Close a database factory. This method frees some resources, but it's not necessary to call it to unlock the databases.

Syntax:

```
void close()
```

4.3.3. exists()

Returns `true` if the database already exists, otherwise `false`.

Syntax:

```
boolean exists()
```

4.3.4. Database create()

Creates a new database. If the database already exists, an exception is thrown.

Example:

```
DatabaseFactory arcade = new DatabaseFactory("/databases/mydb");
Database db = arcade.create();
```

4.3.5. Database open()

Opens an existent database in READ_WRITE mode. If the database does not exist, an exception is thrown.

Example:

```
DatabaseFactory arcade = new DatabaseFactory("/databases/mydb");
try( Database db = arcade.open(); ) {
  // YOUR CODE
}
```

4.3.6. Database open(MODE mode)

Opens an existent database by specifying a mode between READ_WRITE and READ_ONLY mode. If the database does not exist, an exception is thrown. In READ_ONLY mode, any attempt to modify the database throws an exception.

Example:

```
DatabaseFactory arcade = new DatabaseFactory("/databases/mydb");
Database db = arcade.open(MODE.READ_ONLY);
try {
  // YOUR CODE
} finally {
  db.close();
}
```

4.3.7. `Database` Interface

It's the main class to operate with ArcadeDB. To obtain an instance of Database, use the class `DatabaseFactory`.

4.3.8. Methods

Transaction	Lifecycle	Query	Records	Misc
transaction() default	close()	query() positional parameters	newDocument()	async()
transaction() with retries	drop()	query() (parameter map)	newVertex()	command() positional parameters
begin()	isOpen()	lookupByKey()	newEdgeByKeys()	command() (parameter map)
commit()		lookupByRID()	deleteRecord()	getSchema()
rollback()		iterateType()		getSize()
		iterateBucket()		
		scanBucket()		
		scanType()		
		select()		

async()

It returns an instance of `DatabaseAsyncExecutor` to execute asynchronous calls.

Syntax:

```
DatabaseAsyncExecutor async()
```

Example:

Execute an asynchronous query:

```
db.async().query("sql", "select from User", new AsyncResultsetCallback() {
  @Override
  public void onComplete(ResultSet resultSet) {
    while (resultSet.hasNext())
      System.out.println( "Name = " + resultSet.next().getProperty("name"));
  }

  @Override
  public void onError(Exception exception) {
    System.err.println("Error on executing the query: " + exception );
  }
});
```

begin()

Starts a transaction on the current thread. Each thread can have only one active transaction. All the modification to the database become persistent only at pending changes in the transaction are made persistent only when the `commit()` method is called. ArcadeDB supports ACID transactions. Before the commit, no other thread/client can see any of the changes contained in the current transaction.

Syntax:

```
begin()
```

Example:

```
db.begin();  // <--- AT THIS POINT THE TRANSACTION IS STARTED AND ALL THE CHANGES ARE
COLLECTED TILL THE COMMIT (SEE BELOW)
try{
  // YOUR CODE HERE
  db.commit();
} catch( Exception e ){
  db.rollback();
}
```

---

close()

Closes a database. This method should be called at the end of the application. By using Java7+ AutoClosed statement, the `close()` method is executed automatically at the end of the scope of the database variable.

Syntax:

```
void close()
```

Example:

```
Database db = new DatabaseFactory("/temp/mydb").open();
try{
  // YOUR CODE HERE
} finally {
  db.close();
}
```

The suggested method is using Java7+ AutoClosed statement, to avoid the explicit `close()` calling:

```
try( Database db = new DatabaseFactory("/temp/mydb").open(); ) {
  // YOUR CODE
}
```

drop()

Drops a database. The database will be completely removed from the filesystem.

Syntax:

```
void drop()
```

Example:

```
new DatabaseFactory("/temp/mydb").open().drop();
```

getSchema()

Returns the Schema instance for the database.

Syntax:

```
Schema getSchema()
```

Example:

```
db.getSchema().createVertexType("Song");
```

getSize()

Returns the size of the database in bytes.

Syntax:

```
Schema getSize()
```

Example:

```
db.getSize();
```

isOpen()

Returns `true` if the database is open, otherwise `false`.

Syntax:

```
boolean isOpen()
```

Example:

```
if( db.isOpen() ){
  // YOUR CODE HERE
}
```

query(language, command, positionalParameters)

Executes a query, with optional positional parameters. This method only executes idempotent statements, namely `SELECT` and `MATCH`, that cannot change the database. The execution of any other commands will throw a `QueryNotIdempotentException` exception. A write nested anywhere in the statement makes it a write, as described for query(language, command, positionalParameters).

A statement is classified by everything it runs, not only by its first keyword. A write nested in a `LET`, a projection, a `FROM` subquery, a `CASE`, a `RETURN` or a script block makes the whole statement a write, so `SELECT FROM (INSERT INTO T SET a = 1)` or `SELECT *, (DELETE FROM T) AS d FROM V` is refused here: run it with `command()`. `PROFILE` takes the classification of the statement it wraps, while `EXPLAIN` does not execute its statement and stays a read. *(Since v26.10.2)*

Syntax:

```
Resultset query( String language, String command, Object... positionalParameters )
```

Where:

- `language` is the language to use. Only "SQL" language is supported for now, but in the future multiple languages could be used
- `command` is the command to execute. If the language supports prepared statements (SQL does), you can specify parameters by using `?` for positional replacement
- `positionalParameters` optional variable array of parameters to execute with the query

It returns a `Resultset` object where the result can be iterated.

With SQL, the number of positional values must match the placeholders of the statement: each `?`, and each `:name` bound positionally, takes exactly one value (pass `null` explicitly to bind a null). In a `sqlscript` the placeholders of all its statements are counted together. A Postgres-style `$N` reads the value at position `N` and may leave lower positions unread. A call that does not match is refused before the statement runs with a `CommandSQLParsingException` whose message names both counts. A list meant for one placeholder, as in `WHERE @rid IN ?`, is passed as one value: a `List` or a Java array (`RID[]`, `int[]`, ...), which `IN ?` reads as the list of its elements. A parameter map (named parameters) is not counted. The same rule applies to `command()` and to the remote Java client.

```
db.query("sql", "select from V where @rid in ?", List.of(rid1, rid2)); // one
placeholder, one value
db.query("sql", "select from V where @rid in ?", rid1, rid2);          // refused: 1
placeholder, 2 values
```

(since v26.11.1) Before, a surplus value was silently dropped (so `where @rid in ?` with two RIDs spread as two values answered one row) and a missing value was bound to `null`.

Examples:

Simple query:

```
ResultSet resultset = db.query("sql", "select from V");
while (resultset.hasNext()) {
  Result record = resultset.next();
  System.out.println( "Found record, name = " + record.getProperty("name"));
}
```

Query passing positional parameters:

```
ResultSet resultset = db.query("sql", "select from V where age > ? and city = ?", 18,
"Melbourne");
while (resultset.hasNext()) {
  Result record = resultset.next();
  System.out.println( "Found record, name = " + record.getProperty("name"));
}
```

To tell a property that is missing from one stored as `null`, use `Result.getPropertyIfPresent(name, absentValue)` (or `Document.getIfPresent(name, absentValue)` on a record). It returns the value, `null` included, when the property exists, and `absentValue` otherwise. It does in one lookup what `hasProperty()` followed by `getProperty()` does in two.

query(language, command, parameterMap)

Executes a query taking a map for parameters. This method only executes idempotent statements, namely `SELECT` and `MATCH`, that cannot change the database. The execution of any other commands will throw a `QueryNotIdempotentException` exception.

Syntax:

```
Resultset query( String language, String command, Map<String,Object> parameterMap )
```

Where:

- `language` is the language to use. Only "SQL" language is supported for now, but in the future multiple languages could be used
- `command` is the command to execute. If the language supports prepared statements (SQL does), you can specify parameters by name by using `:<arg-name>`
- `parameterMap` this map is used to extract the named parameters

It returns a `Resultset` object where the result can be iterated.

Examples:

```
Map<String,Object> parameters = new HashMap<>();
parameters.put("age", 18);
parameters.put("city", "Melbourne");

ResultSet resultset = db.query("sql", "select from V where age > :age and city =
:city", parameters);
while (resultset.hasNext()) {
  Result record = resultset.next();
  System.out.println( "Found record, name = " + record.getProperty("name"));
}
```

---

command(language, command, positionalParameters)

Executes a command that could change the database. This is the equivalent to `query()`, but allows the command to modify the database. Only "SQL" language is supported, but in the future multiple languages could be used.

Syntax:

```
Resultset command( String language, String command, Object... positionalParameters )
```

Where:

- `language` is the language to use. Only "SQL" is supported
- `command` is the command to execute. If the language supports prepared statements (SQL does), you can specify parameters by using `?` for positional replacement or by name by using `:<arg-name>`
- `positionalParameters` optional variable array of parameters to execute with the query. With SQL their number must match the placeholders of the statement, see positional parameter count *(since v26.11.1)*

It returns a `Resultset` object where the result can be iterated.

Examples:

Create a new record:

```
db.command("sql", "insert into V set name = 'Jay', surname = 'Miner'");
```

Create a new record by passing position parameters:

```
db.command("sql", "insert into V set name = ?, surname = ?", "Jay", "Miner");
```

---

command(language, command, parameterMap)

Executes a command that could change the database. This is the equivalent to `query()`, but allows the command to modify the database. Only "SQL" language is supported, but in the future multiple languages could be used.

Syntax:

```
Resultset command( String language, String command, Map<String,Object> parameterMap )
```

Where:

- `language` is the language to use. Only "SQL" is supported
- `command` is the command to execute. If the language supports prepared statements (SQL does), you can specify parameters by using `?` for positional replacement or by name by using `:<arg-name>`
- `parameterMap` this map is used to extract the named parameters

It returns a `Resultset` object where the result can be iterated.

Examples:

Create a new record by passing a map of parameters:

```
Map<String,Object> parameters = new HashMap<>();
parameters.put("name", "Jay");
parameters.put("surname", "Miner");

db.command("sql", "insert into V set name = :name, surname = :surname", parameters);
```

---

commit()

Commits the thread's active transaction. All the pending changes in the transaction are made persistent. A transaction must be begun by calling the `begin()` method. Rolled back transactions cannot be committed. ArcadeDB supports ACID transactions. Before the commit, no other thread/client can see any of the changes contained in the current transaction. ArcadeDB uses a WAL (Write Ahead Log) as journal in case a crash happens at commit time. In this way, at the next restart, the database can be rollbacked at the previous state. If the commit operation succeed, the changes are immediately visible to the other threads/clients and further transactions of the current thread.

Syntax:

```
commit()
```

Example:

```
db.begin();
try{
  // YOUR CODE HERE
  db.commit();  // <--- COMMIT ALL THE CHANGES "ALL OR NOTHING" IN PERSISTENT WAY
} catch( Exception e ){
  db.rollback();
}
```

---

deleteRecord(record)

Deleted a record. The record will be persistently deleted only at commit time.

(Since v26.11.1) Creating an edge from or to a vertex deleted earlier in the same transaction fails with `VertexNotFoundException`, through the Java API, SQL, Cypher and Gremlin alike, and nothing is stored. Before, the edge record was committed without the deleted vertex (issue #10196).

Syntax:

```
void deleteRecord( Record record )
```

Examples:

```
db.deleteRecord( customer );
```

---

iterateBucket(bucketName)

Iterates all the records contained in a bucket. To scan a type (with all its buckets), use the method iterateType() instead. The result are not accumulated in RAM, but rather this method returns an `Iterator<Record>` that fetches the records only when `.next()` is called.

Syntax:

```
Iterator<Record> iterateBucket( String bucketName )
```

Example:

Aggregate the records by age. This is equivalent to a SQL query with a "group by age":

```
Map<String, AtomicInteger> aggregate = new HashMap<>();

Iterator<Record> result = db.iterateType("V", true );
while( result.hasNext() ){
  Record record = result.next();

  String age = (String) record.get("age");
  AtomicInteger counter = aggregate.get(age);
  if (counter == null) {
    counter = new AtomicInteger(1);
    aggregate.put(age, counter);
  } else
    counter.incrementAndGet();
}
```

Example:

Prints all the records in the bucket "Customer" with age major or equals to 21.

```
Iterator<Record> result = db.iterateBucket("Customer");
while( result.hasNext() ){
  Record record = result.next();

  final Document customer = record.asDocument();
  final Integer age = customer.getInteger("age");
  if (age != null && age >= 21 )
    System.out.println("Found customer: " + customer.getString("name") );
```

```
}
```

---

iterateType(className, polymorphic)

Iterates all the records contained in the buckets relative to a type. If `polymorphic` is `true`, then also the sub-types buckets are considered. To iterate one bucket only check out the iterateBucket() method. The result are not accumulated in RAM, but rather this method returns an `Iterator<Record>` that fetches the records only when `.next()` is called.

Syntax:

```
Iterator<Record> iterateType( String typeName, boolean polymorphic )
```

Example:

Aggregate the records by age. This is equivalent to a SQL query with a "group by age":

```
Map<String, AtomicInteger> aggregate = new HashMap<>();

Iterator<Record> result = db.iterateType("V", true );
while( result.hasNext() ){
  Record record = result.next();

  String age = (String) record.get("age");
  AtomicInteger counter = aggregate.get(age);
  if (counter == null) {
    counter = new AtomicInteger(1);
    aggregate.put(age, counter);
  } else
    counter.incrementAndGet();
}
```

---

lookupByKey(type, properties, keys)

Look ups for one or more records (document, vertex or edge) that match one or more indexed keys.

Syntax:

```
Cursor<RID> lookupByKey( String type, String[] properties, Object[] keys )
```

Where:

- `type` type name

- `properties` array of property names to match
- `keys` array of keys

It returns a `Cursor<RID>` (like an iterator).

The properties must be covered by a key index (`LSM_TREE` or `HASH`). A full-text index matches words rather than whole values, so it cannot answer an exact lookup: `lookupByKey()` and `newEdgeByKeys()` throw an `IllegalArgumentException` when the only index on the properties is `FULL_TEXT`, as they do when there is no index at all.

Examples:

Look up for an author with name "Jay" and surname "Miner". This requires an index on the type "Author", properties "name" and "surname".

```
Cursor<RID> jayMiner = database.lookupByKey("Author", new String[] { "name", "surname"
}, new Object[] { "Jay", "Miner" });
while( jayMiner.hasNext() ){
  System.out.println( "Found Jay! " + jayMiner.next().getProperty("name"));
}
```

---

lookupByRID(rid, loadContent)

Look ups for a record (document, vertex or edge) by its RID (Record Identifier).

Syntax:

```
Record lookupByRID( RID rid, boolean loadContent )
```

Where:

- `rid` is the record identifier
- `loadContent` forces the load of the content too. If the content is not loaded will be lazy loaded at the first access. Use `true` if you are going to access to the record content for sure, otherwise, use `false`

It returns a `Record` implementation (document, vertex or edge).

Examples:

Load the vertex by RID and its content:

```
Vertex v = (Vertex) db.lookupByRID(new RID("#3:47"));
```

---

newDocument(typeName)

Creates a new document of a certain type. The type must be of type "document" and must be created beforehand. In order to be saved, the method `MutableDocument.save()` must be called.

Syntax:

```
MutableDocument newDocument( typeName )
```

Where:

- `typeName` type name

It returns a `MutableDocument` instance.

Examples:

Create a new document of type "Customer":

```
MutableDocument doc = db.newDocument("Customer");
doc.set("name", "Jay");
doc.set("surname", "Miner");
doc.save();  // THE DOCUMENT IS SAVED IN THE DATABASE ONLY WHEN `.save()` IS CALLED
```

Inside a transaction the record is written at the commit, as it is then. A property assigned after the last `save()` and before the commit is therefore stored with the record and, since v26.12.1, also reaches the indexes, and a UNIQUE index refuses it at the commit when another record holds that value. Earlier versions kept the indexes on the values of the last `save()`, so a query by the new value did not find the record (issue #11751). A list, map or embedded document changed in place is not tracked: call `save()` again after changing it.

When an index refuses a `save()` (a vector of the wrong dimension, or with a NaN or an Infinity, a duplicate of a UNIQUE key, the change of the primary key of a partitioned type) the exception can be handled and the transaction goes on. Since v26.12.1, if the record was already saved earlier in the same transaction, the refused `save()` puts its indexed properties back to the values of its last accepted `save()`, so the transaction still commits with all its other writes (issue #13130). The non-indexed properties assigned before the refused `save()` stay on the record. When an indexed value cannot be put back (an index on a nested path of an embedded document or map), the commit is refused with the reason. Earlier versions left the refused value on the record, so the commit failed on it and the whole transaction was rolled back.

The metadata names `@rid`, `@type`, `@cat`, `@in`, `@out`, `@props` and `@this` are not properties. Since v26.10.2 the positional setters (`set(name, value)`, `set(n1, v1, n2, v2)`, `set(Object…)`) refuse them with an `IllegalArgumentException`, before assigning

> anything; earlier versions stored them as properties no reader could return. Another name starting with `@` (for example `@special`) is an ordinary property. The map-based setters (`set(Map)`, `fromMap()`, `fromJSON()`, `merge()`) skip every key starting with `@` as metadata, so the output of `toMap(true)` can be fed back. On an edge, `@in` and `@out` re-point the edge through every one of these setters (earlier versions did it only through `set(name, value)` and `merge()`).

---

newVertex(typeName)

Creates a new vertex of a certain type. The type must be of type "vertex" and must be created beforehand. In order to be saved, the method `MutableVertex.save()` must be called.

Syntax:

```
MutableVertex newVertex( typeName )
```

Where:

- `typeName` type name

It returns a `MutableVertex` instance.

Examples:

Create a new document of type "Customer":

```
MutableVertex v = db.newVertex("Customer");
v.set("name", "Jay");
v.set("surname", "Miner");
v.save();
```

Removing edges while iterating a vertex

The iterators returned by `Vertex.getEdges()` and `Vertex.getVertices()`, with or without edge types, support `Iterator.remove()`. It deletes, through the database, the edge of the entry the last `next()` returned (for `getVertices()`, the edge leading to the vertex just returned; the vertex itself stays): the `DELETE_RECORD` permission is checked, the indexes on the edge are updated, the record events fire and the edge is unlinked from the vertex at the other end too, exactly as `edge.delete()` does. If the delete is refused (a security error, a before-delete listener vetoing it), the exception reaches the caller and the entry stays in place. A second `remove()` without a `next()` in between throws `IllegalStateException`.

```
Iterator<Edge> it = v.getEdges(Vertex.DIRECTION.OUT, "IsFriendOf").iterator();
while (it.hasNext())
  if (it.next().getInteger("since") < 2000)
```

```
    it.remove();
```

(Since v26.10.2)

---

newEdgeByKeys(sourceVertexType, sourceVertexKey, sourceVertexValue, destinationVertexType, destinationVertexKey, destinationVertexValue, createVertexIfNotExist, edgeType, bidirectional, properties)

Creates a new edge between two vertices found by their keys.

Syntax:

```
Edge newEdgeByKeys( String sourceVertexType, String[] sourceVertexKey,
                    Object[] sourceVertexValue,
                    String destinationVertexType, String[] destinationVertexKey,
                    Object[] destinationVertexValue,
                    boolean createVertexIfNotExist, String edgeType, boolean
bidirectional,
                    Object... properties )
```

Where:

- `sourceVertexType` source vertex type name
- `sourceVertexKey` source vertex key properties
- `sourceVertexValue` source vertex key values
- `destinationVertexType` destination vertex type name
- `destinationVertexKey` destination vertex key properties
- `destinationVertexValue` destination vertex key values
- `createVertexIfNotExist` creates source and/or destination vertices if not exist
- `edgeType` edge type name
- `bidirectional` `true` if the edge must be bidirectional, otherwise `false`
- `properties` optional property array with pairs of name (as string) and value

It returns a `MutableEdge` instance.

Examples:

Create a new document of type "Customer":

```
Edge likes = db.newEdgeByKeys( "Account",
                               new String[] {"id"}, new Object[] {322323},
                              "Song",
                               new String[] {"title"},
```

```
                                 new Object[] {"Chasing Cars"},
                                false, "Likes", true);
likes.save();
```

rollback()

Aborts the thread's active transaction by rolling back all the pending changes. Usually the transaction rollback is executed in case of errors. If an exception happens during the call `commit()`, the transaction is roll backed automatically. Once rolled backed, the transaction cannot be committed anymore but it has to be re-started by calling the `begin()` method. With no active transaction (never begun, already committed or already rolled back) `rollback()` does nothing and does not throw, so it is safe in a `finally` block that also runs after a successful commit.

> *(Since v26.11.1)* this holds for the remote drivers too (`RemoteDatabase` over HTTP and `RemoteGrpcDatabase`). Before, they threw `TransactionException("Transaction not begun")` in that state, so a `finally { db.rollback(); }` reported a unit of work that had committed as failed. The same holds for an embedded database opened `READ_ONLY`, where `rollback()` and `commit()` also end a read transaction instead of throwing `DatabaseIsReadOnlyException` (writes are still refused). A remote database still does not support nested transactions: `begin()` while a transaction is already active throws a `TransactionException`.

Syntax:

```
rollback()
```

Example:

```
db.begin();
try{
  // YOUR CODE HERE
  db.commit();
} catch( Exception e ){
  db.rollback(); // <--- ROLLBACK IN CASE OF EXCEPTION
}
```

Or, equivalently, rolling back whatever the block left open:

```
db.begin();
try{
  // YOUR CODE HERE
  db.commit();
} finally {
  db.rollback(); // <--- NO-OP AFTER A SUCCESSFUL COMMIT
```

```
}
```

---

scanBucket(bucketName, callback)

Scans all the records contained in a buckets. For each record found, the callback is called passing the current record. To scan a type (with all its buckets), use the method scanType() instead. The callback method must return `true` to continue the scan, otherwise `false`. Look also at the iterateBucket() method if you want to use an iterator approach instead of callback.

Syntax:

```
void scanBucket(String bucketName, RecordCallback callback);
```

Example:

Prints all the records in the bucket "Customer" with age major or equals to 21.

```
db.scanBucket("Customer", (record) -> {
  final Document customer = record.asDocument();
  final Integer age = customer.getInteger("age");
  if (age != null && age >= 21 )
    System.out.println("Found customer: " + customer.getString("name") );
  return true;
});
```

---

scanType(className, polymorphic, callback)

Scans all the records contained in all the buckets relative to a type. If `polymorphic` is `true`, then also the sub-types buckets are considered. For each record found, the callback is called passing the current record. To scan one bucket only check out the scanBucket() method. The callback method must return `true` to continue the scan, otherwise `false`. Look also at the iterateType() method if you want to use an iterator approach instead of callback.

Syntax:

```
scanType( String className, boolean polymorphic, DocumentCallback callback )
```

Example:

Aggregate the records by age. This is equivalent to a SQL query with a "group by age":

```
Map<String, AtomicInteger> aggregate = new HashMap<>();
```

```
db.scanType("V", true, (record) -> {
  String age = (String) record.get("age");
  AtomicInteger counter = aggregate.get(age);
  if (counter == null) {
    counter = new AtomicInteger(1);
    aggregate.put(age, counter);
  } else
    counter.incrementAndGet();

  return true;
});
```

---

select()

Returns a `Select` builder to construct and execute queries using the Native Select API. This provides a zero-parsing, JIT-friendly fluent query builder as an alternative to SQL or Cypher.

Syntax:

```
Select select()
```

It returns a `Select` instance that supports fluent chaining for building queries.

Examples:

Simple query:

```
Vertex v = db.select().fromType("Person")
    .where().property("name").eq().value("John")
    .vertices().nextOrNull();
```

With count:

```
long count = db.select().fromType("Person")
    .where().property("active").eq().value(true)
    .count();
```

With stream:

```
List<String> names = db.select().fromType("Person")
    .where().property("age").gt().value(18)
    .stream()
    .map(d -> d.getString("name"))
    .collect(Collectors.toList());
```

See the Native Select API documentation for the full reference including operators, graph traversal, and vector search.

---

transaction(txBlock)

This methods wraps a call to the method transaction with retries by using the default retries specified in the database setting `arcadedb.mvccRetries`.

---

transaction(txBlock, retries)

Executes a transaction block as a callback or a clojure. Before calling the callback in `TransactionScope`, the transaction is begun and after the end of the callback, the transaction is committed. In case of any exceptions, the transaction is rolled back. In case a `NeedRetryException` exceptions is thrown (a `ConcurrentModificationException`, a `LockTimeoutException` from the commit's file locks, and the rest of that family), the transaction is repeated up to `retries` times. A `DuplicatedKeyException` is retried once: a genuine duplicate fails the same way on every attempt, and only a concurrent insert that committed between two attempts, as in an upsert that lost the race, can turn it into a success. The same rules apply to `async().transaction()` and to SQL `COMMIT RETRY` *(since v26.11.1)*.

(Since v26.10.2) A `NeedRetryException` whose `isRetryableOnThisNode()` returns `false` is rethrown on the first attempt. That is a refusal that would repeat on every attempt against this node: a `ServerIsNotTheLeaderException` raised by an HA follower that refuses leader-only work (a schema change, a security mutation) and names the leader to send it to. A refusal raised during an election, which names no leader, is still retried.

The retry is skipped if the block already committed part of its work before failing. This happens when the block runs a statement with an explicit batch boundary, such as `UPDATE … BATCH n`, `DELETE … BATCH n` or `MOVE VERTEX … BATCH n`: those commit every `n` records by design, so the rollback can no longer undo the records already committed and repeating the block would apply them a second time. In that case the exception is propagated to the caller, who is the only one who knows how to compensate for the part that already went through. `TRUNCATE TYPE` and `REBUILD TYPE` are not affected, because they skip their internal batching when they run inside a transaction opened by the caller.

Syntax:

```
void transaction( TransactionScope txBlock )
```

Examples:

Example by using Java21+ syntax:

```
db.transaction( () -> {
  final MutableVertex v = database.newVertex("Author");
  v.set("name", "Jay");
  v.set("surname", "Miner");
  v.save();
});
```

Example by using Java7 syntax:

```
db.transaction( new Database.TransactionScope() {
  @Override
  public void execute() {
    final MutableVertex v = db.newVertex("Author");
    v.set("name", "Jay");
    v.set("surname", "Miner");
    v.save();
  }
});
```

4.4. `DatabaseAsyncExecutor` Interface

This is the class to manage asynchronous operations. To obtain an instance of DatabaseAsyncExecutor, use the method `.async()` in `Database`.

The Asynchronous API schedule the operation to be executed as soon as possible, but by a different thread. ArcadeDB optimizes the usage of asynchronous threads to be equals to the number of cores found in the machine (but it is still configurable). Use Asynchronous API if the response of the operation can be managed in asynchronous way and if you want to avoid developing Multi-Threads application by yourself.

query(language, command, callback, positionalParameters)

Executes a query in asynchronous way, with optional positional parameters. This method returns immediately. This method only executes idempotent statements, namely `SELECT` and `MATCH`, that cannot change the database. The execution of any other commands will throw a `IllegalArgumentException` exception.

Syntax:

```
void query( String language, String command, AsyncResultsetCallback callback,
            Object... positionalParameters )
```

Where:

- `language` is the language to use
- `command` is the command to execute If the language supports prepared statements (SQL does),

you can specify parameters by using `?` for positional replacement If the language supports prepared statements (SQL does), you can specify parameters by name by using `:<arg-name>`

- `callback` optional, is the callback that will be used for the whole lifecycle of the result set:
  - `onStart()` executed when the query is parsed and the first result ready
  - `onNext()` executed foreach result in the result set Return `true` to continue browsing the result set, otherwise `false` to interrupt fetching the result set
  - `onComplete()` executed when the entire result set is browsed, or the `onNext()` returned false to interrupt the browsing
  - `onError()` in case of any exception while executing the query
- `positionalParameters` optional variable array of parameters to execute with the query

To iterate the result, use the callback.

Examples:

Simple query:

```
db.async().query("sql", "select from V", new AsyncResultsetCallback() {
  @Override
  public boolean onNext(Result result) {
    System.out.println( "Found record, name = " + result.getProperty("name"));
    return true;
  }

  @Override
  public void onError(Exception exception) {
    System.err.println("Error on executing query: " + exception );
  }
});
```

Query passing positional parameters:

```
db.async().query("sql", "select from V where age > ? and city = ?", new
AsyncResultsetCallback(){
  @Override
  public boolean onNext(Result result) {
    System.out.println( "Found record, name = " + result.getProperty("name"));
    return true;
  }
}, 18, "Melbourne");
```

When you have multiple independent queries that could be executed in parallel, you could use the asynchronous interface and a `CountDownLatch`. The following example executes 3 queries in parallel and wait for all of them to have finished.

```
CountDownLatch counter = new CountDownLatch(3);

final ResultSet[] resultSets = new ResultSet[3];
database.async().query("sql", "select from X", resultset -> { resultSets[0] =
resultset; counter.countDown(); });
database.async().query("sql", "select from Y", resultset -> { resultSets[1] =
resultset; counter.countDown(); });
database.async().query("sql", "select from Z", resultset -> { resultSets[2] =
resultset; counter.countDown(); });

// WAIT INDEFINITELY
counter.await();

// OR YOU CAN SPECIFY A TIMEOUT, LIKE 10 SECONDS TOP
// counter.await(10, TimeUnit.SECONDS);
```

---

query(language, command, callback, parameterMap)

Executes a query in asynchronous way, with optional named parameter. This method returns immediately. This method only executes idempotent statements, namely `SELECT` and `MATCH`, that cannot change the database. The execution of any other commands will throw a `IllegalArgumentException` exception.

Syntax:

```
void query( String language, String command, AsyncResultsetCallback callback,
            Map<String, Object> parameterMap )
```

Where:

- `language` is the language to use
- `command` is the command to execute If the language supports prepared statements (SQL does), you can specify parameters by name by using `:<arg-name>`
- `callback` optional, is the callback that will be used for the whole lifecycle of the result set:
  - `onStart()` executed when the query is parsed and the first result ready
  - `onNext()` executed foreach result in the result set Return `true` to continue browsing the result set, otherwise `false` to interrupt fetching the result set
  - `onComplete()` executed when the entire result set is browsed, or the `onNext()` returned false to interrupt the browsing
  - `onError()` in case of any exception while executing the query
- `parameterMap` this map is used to extract the named parameters

To iterate the result, use the callback.

Examples:

```
Map<String,Object> parameters = Map.of("age", 18, "city", "Melbourne");

db.async().query("sql", "select from V where age > :age and city = :city", new
AsyncResultsetCallback(){
  @Override
  public boolean onNext(Result result) {
    System.out.println( "Found record, name = " + result.getProperty("name"));
    return true;
  }
}, parameters);
```

---

command(language, command, callback, positionalParameters)

Executes any command in asynchronous way, with optional positional parameters and callback. This method returns immediately.

Syntax:

```
void command( String language, String command,
            AsyncResultsetCallback callback,
            Object... positionalParameters )
```

Where:

- `language` is the language to use
- `command` is the command to execute If the language supports prepared statements (SQL does), you can specify parameters by using `?` for positional replacement or by name by using `:<arg-name>` If the language supports prepared statements (SQL does), you can specify parameters by name by using `:<arg-name>`
- `callback` optional, is the callback that will be used for the whole lifecycle of the result set:
  - `onStart()` executed when the query is parsed and the first result ready
  - `onNext()` executed foreach result in the result set Return `true` to continue browsing the result set, otherwise `false` to interrupt fetching the result set
  - `onComplete()` executed when the entire result set is browsed, or the `onNext()` returned false to interrupt the browsing
  - `onError()` in case of any exception while executing the query
- `positionalParameters` optional variable array of parameters to execute with the query

To iterate the result, use the callback.

Examples:

Create a new record:

```
db.async().command("sql", "insert into V set name = 'Jay', surname = 'Miner'", new
AsyncResultsetCallback() {
  @Override
  public boolean onNext(Result result) {
    System.out.println("Created new record: " + result.toJSON() );
    return true;
  }

  @Override
  public void onError(Exception exception) {
    System.err.println("Error on creating new record: " + exception );
  }
});
```

Create a new record by passing position parameters:

```
db.async().command("sql", "insert into V set name = ? surname = ?", new
AsyncResultsetCallback() {
  @Override
  public boolean onNext(Result result) {
    System.out.println("Created new record: " + result.toJSON() );
    return true;
  }
}, "Jay", "Miner");
```

---

command(language, command, callback, parameterMap)

Executes any command in asynchronous way, with optional parameters passed in map format. This method returns immediately. It takes named parameters. Each parameter name (in the SQL query) is prefixed by a semicolon (`:`).

Syntax:

```
void command( String language, String command,
              AsyncResultsetCallback callback,
              Map<String, Object> parameterMap )
```

Where:

- `language` is the language to use
- `command` is the command to execute If the language supports prepared statements (SQL does), you can specify parameters by using `?` for positional replacement or by name by using `:<arg-name>` If the language supports prepared statements (SQL does), you can specify parameters by

name by using `:<arg-name>`

- `callback` optional, is the callback that will be used for the whole lifecycle of the result set:
  - `onStart()` executed when the query is parsed and the first result ready
  - `onNext()` executed foreach result in the result set Return `true` to continue browsing the result set, otherwise `false` to interrupt fetching the result set
  - `onComplete()` executed when the entire result set is browsed, or the `onNext()` returned false to interrupt the browsing
  - `onError()` in case of any exception while executing the query
- `parameterMap` this map is used to extract the named parameters

To iterate the result, use the callback.

Examples:

Create a new record by passing a map of parameters:

```
Map<String,Object> parameters = Map.of("name", "Jay", "surname", "Miner");

db.async().command("sql", "insert into V set name = :name, surname = :surname", new
AsyncResultsetCallback() {
  @Override
  public boolean onNext(Result result) {
    System.out.println("Created new record: " + result.toJSON() );
    return true;
  }

  @Override
  public void onError(Exception exception) {
    System.err.println("Error on creating new record: " + exception );
  }
}, parameters);
```

transaction()

Executes a transaction in async mode. The transaction is executed in a separated thread. Example of use:

```
database.async().transaction( database -> {
  // EXECUTE OPERATIONS WITHIN TRANSACTION ASYNCHRONOUSLY
  // I.E.: database.command("sql", "update client set bonus = true where type = 'gold'
and bonus is null");
  // I.E.: database.command("sql", "insert into notification set type = 'bonus',
status = 'waiting', date = sysdate() ");
});
```

The overload `transaction(block, retries, okCallback, errorCallback)` reports the outcome:

`okCallback` once the transaction committed, `errorCallback` with what made it fail after the transaction was rolled back. That includes an `Error` thrown inside the block, such as an `OutOfMemoryError` or an `AssertionError` *(since v26.11.1; before, an* `Error` *was rolled back without calling* `errorCallback`*, so the submitter could not tell the transaction had failed).*

A failed attempt is retried up to `retries` times on the same rules as `db.transaction()`: the whole `NeedRetryException` family (a `ConcurrentModificationException`, a `LockTimeoutException` from the commit's file locks, and so on), plus a `DuplicatedKeyException` once. A refusal bound to this node, such as an HA follower naming the leader, is not retried locally. Between two attempts the worker waits the same jittered exponential backoff as `db.transaction()` (`arcadedb.txRetryDelay` / `arcadedb.txRetryDelayBase`). *(Since v26.11.1; before, only a* `ConcurrentModificationException` *was retried, with no wait between attempts.)*

---

waitCompletion()

Waits until the tasks submitted before the call have completed, together with the tasks they schedule in turn, and commits the workers' pending batches. Tasks that other threads submit while it waits are not waited for. *(Since v26.11.1; before, it also waited for those, so another thread that kept the async queues busy could keep* `waitCompletion()`*, a parallel* `GraphBatch` *flush or an index build from ever returning.)*

The tasks those other threads' tasks schedule are not waited for either, so `waitCompletion()` returns once your own work is done, whatever anybody else keeps submitting. A task scheduled from the global `onOk()` or `onError()` callback counts as submitted by another thread: a `waitCompletion()` already in progress does not wait for it. *(Since v26.11.1; before, either of these could keep* `waitCompletion()`*, and with it an index build, from ever returning.)*

When no other thread and no global callback submits work, the executor is idle by the time `waitCompletion()` returns: `isProcessing()` asked right after it answers `false`.

```
database.async().waitCompletion();
```

---

setParallelLevel(parallelLevel)

Changes the number of worker threads used for asynchronous operations. It is recommended to keep this number equal to or lower than the cores available on the machine. The initial value comes from the `arcadedb.asyncWorkerThreads` setting (see Settings).

Syntax:

```
void setParallelLevel( int parallelLevel )
```

Example:

```
database.async().setParallelLevel( 8 );
```

This call is **safe to use while other threads have asynchronous work in flight** on the same database. Growing the pool only adds workers and leaves the existing ones untouched. Shrinking it stops routing new work to the workers it retires and lets them finish the tasks already queued on them, so nothing already submitted is dropped and no unrelated caller is told the executor has shut down.

Because a shrink waits for the retired workers to finish, `setParallelLevel()` **blocks** until they have drained, up to an internal bound of ten seconds. If a worker is still busy inside user code when that bound expires, the call returns anyway, logs a warning, and leaves that worker finishing in the background - where it is still counted by `waitCompletion()` and `isProcessing()`. Treat this as an administrative call rather than something to invoke on a hot path.

`async().command()` hands commands to the workers in turn, so with a type that has a single bucket every worker writes the same bucket and their batch commits often conflict. A conflicting batch is replayed (up to `arcadedb.txRetries` times, waiting the `arcadedb.txRetryDelay` backoff between attempts), and a batch that still cannot commit is reported through `onError`. For heavy parallel inserts, create the type with as many buckets as the parallel level, for example `CREATE DOCUMENT TYPE Item BUCKETS 4`.

---

setTransactionUseWAL(useWAL) and setTransactionSync(flushType)

Set the durability policy the asynchronous executor applies to the transactions it opens. Turning the write-ahead log off makes bulk loading substantially faster at the cost of losing crash recovery for those transactions, which is why bulk-loading APIs such as `GraphBatch` use it.

Syntax:

```
void setTransactionUseWAL( boolean transactionUseWAL )
void setTransactionSync( WALFile.FlushType transactionSync )
```

Example:

```
database.async().setTransactionUseWAL( false );
database.async().setTransactionSync( WALFile.FlushType.NO );
```

Both take effect on the **next task** the executor runs, and apply to every task it runs, including DDL dispatched through `command()` with no wait for the response. A change made while a batch transaction is already open closes that transaction first, so records accumulated under the previous policy are always committed under the policy they were written with, never under the new one.

Batch commit conflicts

Each async worker thread batches its writes and commits them together every `arcadedb.asyncTxBatchSize` operations (see Settings). If that periodic commit conflicts with another worker writing the same data, it is now retried automatically (up to `arcadedb.txRetries` attempts) instead of being silently dropped. *Since v26.11.1* the retry covers the whole `NeedRetryException` family, such as a `LockTimeoutException` from the commit's file locks, and a `DuplicatedKeyException` once, like `db.transaction()`; before, only a `ConcurrentModificationException` was retried. A refusal bound to this node (an HA follower naming the leader) is not retried. If every retry fails, the `onError()` callback of every command in that batch is invoked, so a genuine failure is always reported rather than lost.

A statement that commits part-way through on its own - `UPDATE`/`DELETE`/`MOVE VERTEX … BATCH n`, `TRUNCATE TYPE`, `REBUILD INDEX` - makes everything batched before it durable at that moment. The executor recognises this, so a later conflict on the same worker never re-applies those writes and never reports them as lost.

---

createRecord(record, newRecordCallback [,errorCallback])

Create a record in async way. Once the record is applied to the worker's current batch, the callback will be executed. This is not yet durable: if a later operation in the same batch fails, the whole batch rolls back, and the record with it, even though the callback already fired. For a signal that fires only once the batch has committed, register `onOk()` on the executor or call `waitCompletion()`. This method returns immediately. The result can be managed in the NewRecordCallback callback and errors in ErrorCallback callback.

Syntax:

```
void createRecord(final MutableDocument record, final NewRecordCallback
newRecordCallback,
                  final ErrorCallback errorCallback)
```

Where:

- `record` is the mutable record to insert
- `newRecordCallback` is the callback to handle the result after the record has been inserted
- `errorCallback` (optional) is the callback to handle any error raised during insertion

Example on inserting a vertex asynchronously.

```
final MutableVertex vertex = database.newVertex("Customer").set("name", "Albert");
database.async().createRecord(vertex,
                              v -> { System.out.println("Record " + v.toJSON() + "
created") });
```

---

updateRecord(record, updateRecordCallback [,errorCallback])

Updates a record in async way. Once the update is applied to the worker's current batch, the callback will be executed. This is not yet durable: if a later operation in the same batch fails, the whole batch rolls back, and the update with it, even though the callback already fired. For a signal that fires only once the batch has committed, register `onOk()` on the executor or call `waitCompletion()`. This method returns immediately. The result can be managed in the UpdatedRecordCallback callback and errors in ErrorCallback callback.

Syntax:

```
void updateRecord(final MutableDocument record, final UpdatedRecordCallback
updateRecordCallback,
                  final ErrorCallback errorCallback)
```

Where:

- `record` is the mutable record to update
- `updateRecordCallback` is the callback to handle the result after the record has been updated
- `errorCallback` (optional) is the callback to handle any error raised during update]

Example on inserting a vertex asynchronously.

```
database.async().updateRecord(vertex,
                              v -> { System.out.println("Record " + v.toJSON() + "
updated") });
```

---

deleteRecord(record, deleteRecordCallback [,errorCallback])

Deletes a record in async way. Once the delete is applied to the worker's current batch, the callback will be executed. This is not yet durable: if a later operation in the same batch fails, the whole batch rolls back, and the delete with it, even though the callback already fired. For a signal that fires only once the batch has committed, register `onOk()` on the executor or call `waitCompletion()`. This method returns immediately. The result can be managed in the DeletedRecordCallback callback and errors in ErrorCallback callback.

Syntax:

```
void deleteRecord(final Record record, final DeletedRecordCallback
deleteRecordCallback,
                  final ErrorCallback errorCallback)
```

Where:

- `record` is the record to delete
- `updateRecordCallback` is the callback to handle the result after the record has been deleted
- `errorCallback` (optional) is the callback to handle any error raised during deletion

Example on inserting a vertex asynchronously.

```
database.async().deleteRecord(vertex,
                                v -> { System.out.println("Record " + v.toJSON() + "
updated") });
```

---

onOk(okCallback)

Registers a global callback that fires every time a worker's batch transaction has committed: at each `setCommitEvery()` boundary, when a durability setting (`setTransactionUseWAL()` and `setTransactionSync()`) changes mid-batch, and when `waitCompletion()` commits the pending batches. This is the durability signal for `createRecord()`, `updateRecord()` and `deleteRecord()`: their own callbacks fire as soon as the write is applied to the worker's current batch, which a later failure in the same batch can still roll back.

Syntax:

```
void onOk(final OkCallback callback)
```

Example:

```
final AtomicLong commits = new AtomicLong();
database.async().onOk(() -> commits.incrementAndGet());

database.async().createRecord(vertex,
                                v -> { System.out.println("Record " + v.toJSON() + "
applied, not yet durable"); });
database.async().waitCompletion();
```

The callback is not tied to one record: it tells you a batch has committed, not which records were in it. Each worker commits on its own, so the callback runs on the committing worker's thread, possibly from several threads at once. It must be thread-safe and cheap, because a slow callback slows that worker down. A worker also calls it on shutdown even when it had nothing left to commit, so a call at shutdown is not proof that a particular write became durable at that moment. A task scheduled from this callback is not waited for by a `waitCompletion()` already in progress.

"Committed" here means durable in the write-ahead log and safe to replay on the next open. With the default `setTransactionSync(NO)` there is no `fsync` at that point, so the write can still be lost to a power loss or an operating system crash (not to a JVM crash or restart). Set `setTransactionSync(YES_NOMETADATA)` or `YES_FULL` if you

> need that guarantee; the timing of this callback does not change with it.

4.5. Java Schema API

ArcadeDB can work in schema-less mode (like most of NoSQL DBMS), schema-full (like with RDBMS) or hybrid. The main API to manage the schema is the Schema interface you can obtain by calling the API `db.getSchema()`:

```
Schema schema = db.getSchema();
```

Before creating any record it's mandatory to define a type. If you're going to create a new Document, then you need a Document Type. The same applies for Vertex → Vertex Type and Edge → Edge Type.

The specific API to manage document types in the Schema interface are:

```
DocumentType createDocumentType(String typeName);
DocumentType createDocumentType(String typeName, int buckets);
DocumentType createDocumentType(String typeName, int buckets, int pageSize);
```

Where:

- `typeName` is the name of the type
- `buckets` is the number of buckets to create. A bucket is like a file. If not specified, the default is taken from the `arcadedb.typeDefaultBuckets` global setting, which is `1` by default
- `pageSize` is the page size for the file. If not specified is 65K. Pay attention to this value. In case of large objects to store, you need to increase the page size or the record won't be stored, throwing an exception.

To manage vertex types, the API are similar as for the document types:

```
VertexType createVertexType(String typeName);
VertexType createVertexType(String typeName, int buckets);
VertexType createVertexType(String typeName, int buckets, int pageSize);
```

And the same for edge types:

```
EdgeType createEdgeType(String typeName);
EdgeType createEdgeType(String typeName, int buckets);
EdgeType createEdgeType(String typeName, int buckets, int pageSize);
```

In order to retrieve and removing a type, API common to any record type are provided:

```
Collection<DocumentType> getTypes();
```

```
DocumentType            getType(String typeName);
void                    dropType(String typeName);
String                  getTypeNameByBucketId(int bucketId);
DocumentType            getTypeByBucketId(int bucketId);
boolean                 existsType(String typeName);
```

4.5.1. Creating many types at once

Every schema change is written to disk before the call returns, which costs a durable file write per statement. To create many types, properties and indexes at once, wrap them in `bulkChange()`: the schema is written once, at the end, and under HA the whole batch is replicated as one entry.

```
db.getSchema().bulkChange(() -> {
  for (String name : List.of("Customer", "Order", "Product")) {
    db.getSchema().createDocumentType(name).createProperty("id", Type.LONG);
    db.getSchema().createTypeIndex(Schema.INDEX_TYPE.LSM_TREE, true, name, "id");
  }
});
```

There is no rollback. If a call fails, the changes made before it are kept and published (under HA, as one replicated entry, so every node holds the same partial schema), and the exception is rethrown. This is the same rule a schema-only SQL script follows; see reference/sql/sql-script.pdf.

The database write lock is held for the whole callback, so use it for schema definition, not for loading data. On a remote database (`RemoteDatabase`) `bulkChange()` simply runs the callback: each call is already its own HTTP request, so it is safe to call but batches nothing. See reference/storage.pdf for the other ways to batch schema changes.

4.5.2. Working with buckets

A bucket is like a file. A type can rely on one or multiple buckets. Why using multiple buckets? Because ArcadeDB could lock a bucket for certain operations. Having multiple buckets allows to go in parallel with a multi-cpus and multi-cores architecture.

The specific API to manage buckets are:

```
Bucket              createBucket(String bucketName);
boolean             existsBucket(String bucketName);
Bucket              getBucketById(int id);
Bucket              getBucketByName(String name);
Collection<Bucket>  getBuckets();
```

4.5.3. Working with indexes

Like any other DBMS, ArcadeDB has indexes. Even if indexes are not used to manage relationships (because ArcadeDB has a native GraphDB engine based on links), indexes are fundamental for a quick lookup of records by one or multiple properties.

By default null values are not indexed, so any query that is looking for null values will not use the index but instead result in a full scan.

ArcadeDB provides automatic and manual indexes:

- `automatic` that are updated automatically when you work with records
- `manual` are detached from a type and the user is totally responsible to insert and remove entries into and from the index

The `Schema` API to manage indexes are:

```
TypeIndex createTypeIndex(SchemaImpl.INDEX_TYPE indexType, boolean unique, String
typeName, String[] propertyNames);

TypeIndex createTypeIndex(SchemaImpl.INDEX_TYPE indexType, boolean unique, String
typeName, String[] propertyNames, int pageSize);

TypeIndex createTypeIndex(EmbeddedSchema.INDEX_TYPE indexType, boolean unique, String
typeName, String[] propertyNames, int pageSize, Index.BuildIndexCallback callback);

TypeIndex createTypeIndex(EmbeddedSchema.INDEX_TYPE indexType, boolean unique, String
typeName, String[] propertyNames, int pageSize, LSMTreeIndexAbstract.NULL_STRATEGY
nullStrategy, Index.BuildIndexCallback callback);

boolean existsIndex(String indexName);

Index[] getIndexes();

Index   getIndexByName(String indexName);
```

Where:

- `indexName` is the name of the index
- `indexType` can be:
  - `LSM_TREE`, implemented as a Log Structured Merge tree. Best for range scans and ordered iteration.
  - `HASH`, implemented using extendable hashing. Provides O(1) equality lookups — best for primary keys, JOINs, and edge traversal where ordering is not needed. Does not support range queries.
  - `FULL_TEXT`, that uses Lucene's Analyzers for tokenizing, stemming and categorize words inside a text. Internally it's managed as a LSM_TREE
- `unique` tells if the entries in the index must be unique or they can be repeated
- `typeName` is the name of the type (document, vertex or edge) where the index must be applied
- `propertyNames` is the array of property names to index. In case of more than one property is used, the index is composed. Property names may also reference a document type's

polymorphic properties

- `pageSize` is the page size. If not specified, the default of 2MB is used
- `nullStrategy` can be:
  - `SKIP` ignores all null values when creating and reading an index, this is the default.
  - `ERROR` throws an exception when creating or reading a null value indexed property.
- `callback` is a 2 argument Functional interface called when each `Document` is added to this index. *This is most likely only useful for internal use cases.* The callback receives the `Document` just indexed, and a running count of the total number of documents indexed in the bucket.

The API above sometimes returns `TypeIndex` vs `Index`. The more general `getIndexes` and `getIndexByName` methods return `Index` because the index might be a bucket index or a type index. The `TypeIndex` should be useful in most cases, but sometimes it may be useful to work at the bucket level.

In addition to the `createTypeIndex` methods above, for each overload there is a matching `getOrCreateTypeIndex` method which returns an existing index rather than throwing an exception if a matching index is found. There are also matching index API creation methods available via the `DocumentType` for creating an index directly from a document type, rather than against the general `Schema` object. When using the `DocumentType` convenience variants `typeName` is provided automatically.

An index is unique to a set of property names per document type. Any attempt to create a second index on a non-unique set of [`typeName`, `propertyNames`] will throw an exception using the `createTypeIndex` method variants.

A special mention goes for the method `createManualIndex()` that creates indexes not attached to any type (manual):

```
Index createManualIndex(SchemaImpl.INDEX_TYPE indexType, boolean unique, String
indexName, com.arcadedb.schema.Type[] keyTypes, int pageSize);
```

While by default indexes are updated automatically when you work with records, in this case, the user is totally responsible to insert and remove entries into and from the index.

4.5.4. Indexing Edges

Like any other document type, indexes may be defined for `Edge` types as well. If the property name for the index is either `@out` or `@in`, the index property will be a `LINK` type on the adjacently referenced `Vertex`.

The `LINK` type represents @RIDs (like #13:222). Usually creating `LINK` indexes is meant for indexing incoming/outgoing edges in order to prevent multigraphs (i.e. duplicates edges between the same vertex pairs).

4.5.5. Database Configuration

ArcadeDB stores the database configuration into the schema and allows to change things like the timezone, the format of dates and the encoding:

```
ZoneId   getZoneId();
void     setZoneId(ZoneId zoneId);
TimeZone getTimeZone();
void     setTimeZone(TimeZone timeZone);
String   getDateFormat();
void     setDateFormat(String dateFormat);
String   getDateTimeFormat();
void     setDateTimeFormat(String dateTimeFormat);
String   getEncoding();
void     setEncoding(String encoding);
```

`setZoneId()`, `setDateFormat()` and `setDateTimeFormat()` are saved with the schema and need the `updateDatabaseSettings` permission (see Groups); setting the value a setting already has changes nothing. `setZoneId()` is the setter to use for the time zone: `setTimeZone()` changes only the value held in memory by the open database, and the zone saved with the schema is the one `setZoneId()` sets. `setEncoding()` is the charset used to read a legacy schema file on this open; it is not saved, and the schema is always written as UTF-8.

4.5.6. Schema Changes in a High Availability Cluster

(Since v26.10.2) every programmatic schema change is saved and replicated through the same schema session as the SQL DDL, so under HA it must run on the leader. On a follower these calls throw a `ServerIsNotTheLeaderException` (`Changes to the schema must be executed on the leader server`) instead of changing only the local `schema.json`:

- the `Property` setters (`setMandatory()`, `setNotNull()`, `setReadonly()`, `setHidden()`, `setMin()`, `setMax()`, `setRegexp()`, `setOfType()`, `setExternal()`, `setCompression()`, `setCustomValue()`, ...);
- `Schema.setExtension()`;
- the date settings `setZoneId()`, `setDateFormat()` and `setDateTimeFormat()`;
- `setLightweight()` and `setUnique()` on an edge type (`LocalEdgeType`, what `ALTER TYPE … WITH lightweight` runs);
- `GraphAnalyticalViewBuilder.build()` for a named view, which saves the view definition with the schema.

A statement sent over HTTP to a follower (`ALTER PROPERTY`, `ALTER DATABASE`, ...) is forwarded to the leader.

4.6. Native Select API

The Native Select API is a zero-parsing, JIT-friendly fluent query builder for embedded Java applications. It provides a type-safe alternative to SQL or Cypher for common query patterns, with

automatic index detection and significantly lower overhead.

4.6.1. Quick Start

```
// Simple query
List<Vertex> results = database.select().fromType("Person")
    .where().property("age").gt().value(18)
    .vertices().toList();

// Compiled query with parameters (reusable)
SelectCompiled query = database.select().fromType("Person")
    .where().property("name").eq().parameter("name")
    .compile();

Vertex john = query.parameter("name", "John").vertices().nextOrNull();
Vertex jane = query.parameter("name", "Jane").vertices().nextOrNull();
```

4.6.2. Building Queries

From Type or Buckets

Every query starts with `database.select()` followed by the source:

```
// Query from a type (polymorphic by default)
database.select().fromType("Person")...

// Disable polymorphic query (only the exact type, not subtypes)
database.select().fromType("Person").polymorphic(false)...

// Query from specific buckets
database.select().fromBuckets("person_bucket_0", "person_bucket_1")...
```

WHERE Conditions

The `where()` method starts the fluent condition builder. Each condition follows the pattern: `property(name)` → `operator()` → `value(val)`.

```
database.select().fromType("Person")
    .where().property("age").gt().value(18)
    .and().property("active").eq().value(true)
    .vertices().toList();
```

Comparison Operators

Method	Description	Example
`eq()`	Equals	`.property("name").eq().value("John")`
`neq()`	Not equals	`.property("status").neq().value("inactive")`
`lt()`	Less than	`.property("age").lt().value(30)`
`le()`	Less than or equal	`.property("age").le().value(30)`
`gt()`	Greater than	`.property("age").gt().value(18)`
`ge()`	Greater than or equal	`.property("age").ge().value(18)`
`like()`	Pattern match (case-sensitive)	`.property("name").like().value("J%")`
`ilike()`	Pattern match (case-insensitive)	`.property("name").ilike().value("j%")`
`in()`	Value in collection	`.property("status").in().value(List.of("A", "B"))`
`between()`	Value in range (inclusive)	`.property("age").between().values(18, 65)`
`isNull()`	Property is null (unary)	`.property("email").isNull()`
`isNotNull()`	Property is not null (unary)	`.property("email").isNotNull()`

`like()` and `ilike()` follow exactly the same rules as SQL `LIKE`/`ILIKE`: a record without the property, or a `null` pattern, does not match; a non-string value is matched by its text form; a list property matches when any of its items does.

A comparison against a missing or `null` value is never true. `lt()`, `le()`, `gt()`, `ge()` and `between()` all skip a record that does not carry the property, or carries it as `null`, exactly as SQL does - so `age < 30` never matches a record without an `age`. Use `isNull()` / `isNotNull()` to ask about absent values.

IN Operator

The `in()` operator checks if a property value is contained in a collection. When the property is indexed, ArcadeDB performs multiple point lookups on the index.

```
// Find persons with specific statuses
List<Vertex> results = database.select().fromType("Person")
    .where().property("status").in().value(List.of("active", "pending"))
    .vertices().toList();

// IN with parameterized values
SelectCompiled query = database.select().fromType("Order")
    .where().property("status").in().parameter("statuses")
    .compile();

List<Document> orders = query.parameter("statuses", List.of("shipped", "delivered"))
```

```
    .documents().toList();
```

BETWEEN Operator

The `between()` operator checks if a value falls within an inclusive range. When the property is indexed, ArcadeDB uses a range scan on the index.

```
// Find orders with amount between 100 and 500
List<Document> results = database.select().fromType("Order")
    .where().property("amount").between().values(100, 500)
    .documents().toList();
```

IS NULL / IS NOT NULL

These are unary operators — they don't require a right-hand value.

```
// Find records where email is not set
List<Document> results = database.select().fromType("Person")
    .where().property("email").isNull()
    .documents().toList();

// Combine with other conditions
List<Vertex> active = database.select().fromType("Person")
    .where().property("email").isNotNull()
    .and().property("age").gt().value(18)
    .vertices().toList();
```

AND / OR Logic

Conditions can be combined with `and()` and `or()`. AND has higher precedence than OR (same as SQL).

```
// AND: both conditions must match
database.select().fromType("Person")
    .where().property("age").gt().value(18)
    .and().property("active").eq().value(true)
    .vertices();

// OR: either condition matches
database.select().fromType("Person")
    .where().property("role").eq().value("admin")
    .or().property("role").eq().value("manager")
    .vertices();

// Mixed: AND binds tighter than OR
// Equivalent to: (age > 18 AND active = true) OR role = 'admin'
database.select().fromType("Person")
```

```
    .where().property("age").gt().value(18)
    .and().property("active").eq().value(true)
    .or().property("role").eq().value("admin")
    .vertices();
```

4.6.3. Terminal Operations

Terminal methods execute the query and return results.

Iterator-Based Results

```
// Returns SelectIterator<Vertex>
SelectIterator<Vertex> iter = database.select().fromType("Person")
    .where().property("active").eq().value(true)
    .vertices();

// Returns SelectIterator<Edge>
SelectIterator<Edge> iter = database.select().fromType("Follows")
    .where().property("since").gt().value(2020)
    .edges();

// Returns SelectIterator<Document>
SelectIterator<Document> iter = database.select().fromType("Log")
    .where().property("level").eq().value("ERROR")
    .documents();
```

`SelectIterator` extends `Iterator` and adds:

- `toList()` — collects all results into a `List`
- `nextOrNull()` — returns the next element or `null` if empty
- `stream()` — returns a Java `Stream` for functional-style processing
- `getMetrics()` — returns execution metrics (evaluated records, used indexes)
- `outVertices()` / `inVertices()` / `bothVertices()` — traverse to adjacent vertices (see Graph Traversal)
- `outEdges()` / `inEdges()` / `bothEdges()` — traverse to adjacent edges (see Graph Traversal)

count()

Returns the number of matching records without materializing them. Respects `skip()` and `limit()`.

```
long total = database.select().fromType("Person")
    .where().property("active").eq().value(true)
    .count();

// With limit
long capped = database.select().fromType("Person")
```

```
    .where().property("active").eq().value(true)
    .limit(100).count();

// From compiled query
SelectCompiled query = database.select().fromType("Order")
    .where().property("status").eq().parameter("status")
    .compile();

long pending = query.parameter("status", "pending").count();
```

exists()

Returns `true` if at least one matching record exists. Short-circuits immediately on the first match (does not scan further).

```
boolean hasAdmins = database.select().fromType("Person")
    .where().property("role").eq().value("admin")
    .exists();
```

stream()

Returns a standard Java `Stream<Document>` for functional-style processing.

```
List<String> names = database.select().fromType("Person")
    .where().property("age").gt().value(18)
    .stream()
    .map(d -> d.getString("name"))
    .collect(Collectors.toList());

// From iterator (preserves the generic type)
Optional<Vertex> first = database.select().fromType("Person")
    .where().property("name").eq().value("Alice")
    .vertices().stream()
    .findFirst();
```

4.6.4. Pagination and Ordering

Skip and Limit

```
// Skip the first 20, return the next 10
List<Vertex> page = database.select().fromType("Person")
    .where().property("active").eq().value(true)
    .skip(20).limit(10)
    .vertices().toList();
```

Order By

```
// Sort ascending by name
List<Vertex> sorted = database.select().fromType("Person")
    .where().property("active").eq().value(true)
    .orderBy("name", true)  // true = ascending
    .vertices().toList();

// Sort descending by age
List<Vertex> sorted = database.select().fromType("Person")
    .orderBy("age", false)  // false = descending
    .vertices().toList();
```

When the ORDER BY property matches an indexed property used in the WHERE condition, ArcadeDB reads results directly from the index in sorted order, avoiding an in-memory sort. This also applies when ORDER BY targets the property immediately following a matched composite-index prefix (see Index Detection): the index is scanned as a directional range in that case, so it already returns rows in the requested order.

4.6.5. Compiled Queries

Compiled queries separate query definition from execution. The query structure is built once, then reused with different parameter values.

```
// Compile once
SelectCompiled query = database.select().fromType("Person")
    .where().property("name").eq().parameter("name")
    .and().property("city").eq().parameter("city")
    .compile();

// Execute many times with different parameters
Vertex v1 = query.parameter("name", "John").parameter("city", "NYC").vertices
().nextOrNull();
Vertex v2 = query.parameter("name", "Jane").parameter("city", "LA").vertices
().nextOrNull();

// All terminal methods are available on compiled queries
long count = query.parameter("name", "John").parameter("city", "NYC").count();
boolean exists = query.parameter("name", "John").parameter("city", "NYC").exists();
Stream<Document> stream = query.parameter("name", "John").parameter("city", "NYC"
).stream();
```

JSON Serialization

Compiled queries can be serialized to/from JSON for persistence or network transfer:

```
SelectCompiled original = database.select().fromType("Person")
    .where().property("name").eq().parameter("name")
```

```
    .compile();

// Serialize to JSON
JSONObject json = original.json();

// Deserialize from JSON
SelectCompiled restored = database.select().json(json).compile();

Vertex v = restored.parameter("name", "John").vertices().nextOrNull();
```

`json()` and `json(JSONObject)` are inverse: reading a serialized query back and serializing it again produces the same document.

The `where` entry is a condition, and a condition is one of:

Shape	Meaning
`[left, "<operator>", right]`	A binary condition, e.g. `[":name", "=", "John"]`.
`[left, "<operator>"]`	A unary condition: `not`, `is null` and `is not null` are the only operators that take no right operand.
`[condition]`	A single condition, the shape a query with one `where` leaf serializes to.

An operand is `":property"` for a property, `"#parameter"` for a named parameter, a nested condition array, or a literal value. A nested array **is a parenthesis**: it is evaluated as one unit, so `[[":a", "=", 2], "and", [[":b", "=", 1], "or", [":b", "=", 3]]]` means `a = 2 AND (b = 1 OR b = 3)` and never the re-associated `(a = 2 AND b = 1) OR b = 3`. This is the only way to hand the native API an explicitly grouped condition: the fluent builder appends conditions left to right and applies operator precedence.

A literal value that itself starts with `:` or `#` is escaped by doubling that leading character (e.g. the literal `":alice"` is written as `"::alice"`), so it is never confused with a property or parameter reference. This happens automatically on serialization; a hand-written literal starting with `:` or `#` must be doubled the same way.

The right operand of `between` and `in` is a plain array of values — `[":age", "between", [18, 65]]` — which is told apart from a nested condition by its shape: a condition's second element names an operator.

4.6.6. Parallel Execution

For large datasets, enable parallel execution to distribute bucket scanning across multiple threads:

```
SelectCompiled query = database.select().fromType("LargeTable")
    .where().property("status").eq().value("active")
    .compile()
    .parallel();
```

```
List<Document> results = query.documents().toList();
```

Parallel queries cannot use the current transaction context because they execute in separate threads. Use them for read-only queries outside transactions.

4.6.7. Timeout

Set a maximum execution time for queries:

```
// Timeout after 5 seconds, throw TimeoutException
database.select().fromType("Person")
    .where().property("name").like().value("%pattern%")
    .timeout(5, TimeUnit.SECONDS, true)
    .documents().toList();

// Timeout after 5 seconds, silently return partial results
database.select().fromType("Person")
    .where().property("name").like().value("%pattern%")
    .timeout(5, TimeUnit.SECONDS, false)
    .documents().toList();
```

4.6.8. Index Detection

The Select API automatically detects and leverages available indexes:

- **Equality (`eq`)** — single-key index lookup
- **Range (`gt`, `ge`, `lt`, `le`)** — range scan on the index
- **IN (`in`)** — multiple point lookups on the index; with an ascending `orderBy()` on the same property the rows come back in key order
- **BETWEEN (`between`)** — range scan with inclusive bounds
- **AND logic** — uses the best available index (prefers unique indexes)
- **OR logic** — uses both indexes when both sides are indexed, even when they are on properties of different types; falls back to full scan when either side is not indexed
- **Composite indexes** — an AND chain of equality conditions covering a leading prefix of a composite (multi-property) index's properties uses that index too, even when not every property is bound; if the query's `orderBy()` matches the trailing indexed property, the scan also satisfies the ordering without a separate in-memory sort

Operators that cannot use indexes (`neq`, `like`, `ilike`, `isNull`, `isNotNull`) always require a full scan (but can still benefit from indexes on other conditions in an AND chain).

Use `getMetrics()` on the iterator to verify index usage:

```
SelectIterator<Vertex> iter = database.select().fromType("Person")
    .where().property("id").eq().value(42)
```

```
    .vertices();

Vertex result = iter.nextOrNull();

Map<String, Object> metrics = iter.getMetrics();
long evaluatedRecords = (long) metrics.get("evaluatedRecords");
int usedIndexes = (int) metrics.get("usedIndexes");
```

4.6.9. Graph Traversal

The Select API supports graph traversal directly from query results. ArcadeDB is optimized to skip edges when you only need vertices — use `outVertices()` / `inVertices()` / `bothVertices()` for vertex-to-vertex traversal, and `outEdges()` / `inEdges()` / `bothEdges()` when you need the edge records themselves.

Vertex Traversal

```
// Single hop: find all friends of Alice
List<Vertex> friends = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .outVertices("FRIENDS")
    .toList();

// Multi-hop: friends of friends
Set<String> friendsOfFriends = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .outVertices("FRIENDS")
    .thenOutVertices("FRIENDS")
    .stream()
    .map(v -> v.getString("name"))
    .collect(Collectors.toSet());

// Incoming direction
List<Vertex> followers = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .inVertices("FOLLOWS")
    .toList();

// Both directions
List<Vertex> connections = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .bothVertices("KNOWS")
    .toList();

// No edge type filter (all edge types)
```

```
List<Vertex> allNeighbors = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .outVertices()
    .toList();
```

`SelectVertexTraversal` implements `Iterator<Vertex>` and provides:

- `thenOutVertices(edgeTypes…)` — chain another outgoing vertex traversal
- `thenInVertices(edgeTypes…)` — chain another incoming vertex traversal
- `thenBothVertices(edgeTypes…)` — chain another bidirectional vertex traversal
- `thenOutEdges(edgeTypes…)` — switch to edge traversal (outgoing)
- `thenInEdges(edgeTypes…)` — switch to edge traversal (incoming)
- `thenBothEdges(edgeTypes…)` — switch to edge traversal (both directions)
- `toList()` — collect results into a `List<Vertex>`
- `stream()` — return a Java `Stream<Vertex>`

Edge Traversal

Use edge traversal when you need the edge records (e.g., to read edge properties):

```
// Get outgoing FRIENDS edges
List<Edge> edges = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .outEdges("FRIENDS")
    .toList();

// Access edge properties
for (Edge edge : edges) {
    System.out.println(edge.getString("since"));
}

// Vertex-to-edge chaining: find friends, then get their WORKS_AT edges
List<Edge> worksAt = database.select().fromType("User")
    .where().property("name").eq().value("Alice")
    .vertices()
    .outVertices("FRIENDS")
    .thenOutEdges("WORKS_AT")
    .toList();
```

`SelectEdgeTraversal` implements `Iterator<Edge>` and provides:

- `toList()` — collect results into a `List<Edge>`
- `stream()` — return a Java `Stream<Edge>`

4.6.10. Vector k-NN Search

The Select API provides a fluent interface for vector similarity search. It requires a vector index on the target property.

```
float[] queryVector = getEmbedding("search query");

// Basic k-NN search: find 10 nearest neighbors
List<SelectVectorResult<Vertex>> results = database.select()
    .fromType("Product")
    .nearestTo("embedding", queryVector, 10)
    .vertices();

// Access results
for (SelectVectorResult<Vertex> result : results) {
    Vertex doc = result.getDocument();     // the matched vertex
    float distance = result.getDistance();  // distance from query vector
}
```

Results are returned as a `List` (not an iterator) sorted by ascending distance, because vector search is inherently bounded by `k` and requires cross-bucket merging.

The search covers the records of the type passed to `fromType()` and of its sub-types, even when the vector index is defined on a parent type. With `polymorphic(false)` it covers that type only. A record deleted by a concurrent transaction while the search runs is skipped, and the search widens to still return `k` results when enough records exist.

Approximate Search

For large datasets, enable approximate nearest neighbor search using product quantization:

```
List<SelectVectorResult<Vertex>> results = database.select()
    .fromType("Product")
    .nearestTo("embedding", queryVector, 10)
    .approximate(true)
    .vertices();
```

Post-Filtering

Apply WHERE conditions to filter vector search results:

```
// Find 50 nearest, then filter to only "electronics" category
List<SelectVectorResult<Vertex>> results = database.select()
    .fromType("Product")
    .nearestTo("embedding", queryVector, 50)
    .where().property("category").eq().value("electronics")
    .vectorVertices();
```

Post-filtering is applied after the k-NN search. To ensure enough results after filtering, request more candidates than you need (e.g., request 50 to get ~10 after filtering).

4.6.11. Complete Example

```
try (DatabaseFactory factory = new DatabaseFactory("./mydb")) {
  Database database;
  if (factory.exists())
    database = factory.open();
  else
    database = factory.create();

  // Create schema
  VertexType person = database.getSchema().createVertexType("Person");
  person.createProperty("name", Type.STRING)
      .createIndex(Schema.INDEX_TYPE.LSM_TREE, true);
  person.createProperty("age", Type.INTEGER);
  database.getSchema().createEdgeType("FRIENDS");

  // Insert data
  database.transaction(() -> {
    Vertex alice = database.newVertex("Person").set("name", "Alice", "age", 30).
save();
    Vertex bob = database.newVertex("Person").set("name", "Bob", "age", 25).save();
    alice.newEdge("FRIENDS", bob).save();
  });

  // Query with the Select API
  // Count
  long adults = database.select().fromType("Person")
      .where().property("age").ge().value(18).count();

  // Check existence
  boolean hasAlice = database.select().fromType("Person")
      .where().property("name").eq().value("Alice").exists();

  // Stream processing
  List<String> names = database.select().fromType("Person")
      .where().property("age").gt().value(20)
      .stream()
      .map(d -> d.getString("name"))
      .sorted()
      .collect(Collectors.toList());

  // Graph traversal
  List<Vertex> aliceFriends = database.select().fromType("Person")
      .where().property("name").eq().value("Alice")
      .vertices()
```

```
      .outVertices("FRIENDS")
      .toList();

  database.close();
}
```

4.7. Embedded Documents

ArcadeDB is a Multi-Model database with a full support for documents. The nice thing about documents (and Document Databases) is that they can have embedded documents. This feature is very powerful. In some cases is preferable to embed documents instead of connect them by using a graph.

```
{
  "firstName": "Jay",
  "lastName": "Miner",
  "worksAt": {
    "companyName": "Commodore",
    "since": "1982"
  }
}
```

Below you can find the code to create such document by using the Java API. Note the creation of the types at the beginning:

```
db.transaction( () -> {
  // CREATE THE SCHEMA (NEED ONLY ONCE BEFORE CREATING RECORDS)
  DocumentType employee = db.getSchema().createDocumentType("Employee");
  DocumentType company = db.getSchema().createDocumentType("Company");

  // CREATE DOCUMENTS
  MutableDocument jay = db.newDocument("Employee").set("firstName", "Jay", "lastName",
"Miner");

  jay.newEmbeddedDocument("Company", "worksAt").set("companyName", "Commodore",
"since", 2010);

  // MAKE THE MAIN DOCUMENT PERSISTENT
  jay.save();
});
```

Modeling with a graph it would be something like this:

And this would be the code to create the types and the graph.

```
db.transaction( () -> {
  // CREATE THE SCHEMA (NEED ONLY ONCE BEFORE CREATING RECORDS)
```

```
  VertexType employee = db.getSchema().createVertexType("Employee");
  VertexType company = db.getSchema().createVertexType("Company");
  EdgeType worksAt = db.getSchema().createEdgeType("WorksAt");

  // CREATE THE GRAPH
  MutableVertex jay = db.newVertex("Employee").set("firstName", "Jay", "lastName",
"Miner").save();
  MutableVertex commodore = db.newVertex("Company").set("name", "Commodore").save();
  jay.newEdge("WorksAt", commodore, "since", 2010);
});
```

With ArcadeDB Multi-Model DBMS you can have vertices with embedded documents linked to other vertices through edges. Check out this example that uses a graph to connect Employee with Company, but keeps the addresses as embedded documents.

```
db.transaction( () -> {
  // CREATE THE SCHEMA (NEED ONLY ONCE BEFORE CREATING RECORDS)
  VertexType employee = db.getSchema().createVertexType("Employee");
  VertexType company = db.getSchema().createVertexType("Company");
  EdgeType worksAt = db.getSchema().createEdgeType("WorksAt");
  DocumentType address = db.getSchema().createDocumentType("Address");

  // CREATE THE GRAPH + EMBEDDED DOCUMENTS
  MutableVertex jay = db.newVertex("Employee").set("firstName", "Jay", "lastName",
"Miner").save();
  jay.newEmbeddedDocument("Address", "residenceAddress").set("city", "San Francisco",
"state", "CA", "country", "USA");

  MutableVertex commodore = db.newVertex("Company").set("name", "Commodore").save();
  commodore.newEmbeddedDocument("Address", "hqAddress").set("city", "Palo Alto",
"state", "CA", "country", "USA");
  commodore.newEmbeddedDocument("Address", "ukAddress").set("city", "London", "state",
"London", "country", "UK");

  jay.newEdge("WorksAt", commodore, "since", 2010);
});
```

To retrieve embedded documents, you can retrieve as any other properties. Example:

```
db.transaction( () -> {
  ResultSet result = db.query( "SQL", "select from Employee where firstName = ? and
lastName = ?", "Jay", "Miner" );
  Vertex jay = result.next().getVertex().get();

  EmbeddedDocument residenceAddress = (EmbeddedDocument) jay.get("residenceAddress");
  System.out.println( "Jay's lives in " + residenceAddress.getString("city") );
});
```

4.8. Events

ArcadeDB allows hooking listener to the following events on records (vertices, edges, documents):

- **before** is **created**, by registering the interface `BeforeRecordCreateListener`
- **after** is **created**, by registering the interface `AfterRecordCreateListener`
- **before** is **read**, by registering the interface `BeforeRecordReadListener`
- **after** is **read**, by registering the interface `AfterRecordReadListener`
- **before** is **updates**, by registering the interface `BeforeRecordUpdateListener`
- **after** is **updated**, by registering the interface `AfterRecordUpdateListener`
- **before** is **deleted**, by registering the interface `BeforeRecordDeleteListener`
- **after** is **deleted**, by registering the interface `AfterRecordDeleteListener`

The listeners above can be installed and removed at database by using:

```
database.getEvents().registerListener()
```

And at specific type level by using:

```
database.getSchema().getType(<type-name>).registerListener()
```

All the interface listeners that work **before** a record is created, read, updated or deleted, require to return a boolean value. If the callback returns `true`, the listener chain continues and all the following listeners are invoked. If `false`, the chain of calls is interrupted and the operation is skipped with no errors. In case an error is requested, the callback can throw an exception instead of returning false.

A write a before listener skipped is not counted as applied anywhere a count is replied: the count of a SQL `DELETE` or `UPDATE` (and so the Postgres wire and the HTTP API reply), the records an `INSERT` returns, the `n`/`nModified`/`upserted` of a MongoDB `insert`, `update` or `delete`, the integer of a Redis `HSET` or `HDEL`, and the `inserted`/`updated` counters of a gRPC insert, where a vetoed row is counted as `ignored`. A Java caller that counts its own writes learns the answer from `MutableDocument.saveReportingVeto()`, which returns `false` for a skipped create or update, where `save()` returns the document whatever happened. The lower-level `DatabaseInternal.createRecordReportingVeto()`, `updateRecordReportingVeto()` and `deleteRecordReportingVeto()` answer the same for each operation *(since v26.10.2)*.

Over the remote Java drivers (HTTP `RemoteDatabase` and gRPC `RemoteGrpcDatabase`) a record cannot be skipped silently, because the caller is not the process that registered the listener: `save()` of a document, a vertex or an edge a server side listener refused raises `RecordVetoedException`, for a create and for an update alike, and `saveReportingVeto()` answers `false` instead. The driver statistics (`createRecord`,

`updateRecord`) count only the writes the server applied. Before v26.11.1 a vetoed update was reported over gRPC as a success, over HTTP as a `RecordNotFoundException`, and a vetoed create over HTTP as a `NoSuchElementException`; the update of a remote edge that no longer exists, which used to succeed silently, now raises `RecordNotFoundException` as a document's does *(since v26.11.1)*. `vertex.newEdge()` of a remote vertex raises `RecordVetoedException` too, with the message the embedded engine builds (the edge type and both endpoints), where it used to fail with a bare `java.util.NoSuchElementException` because the server answers no row for a refused `CREATE EDGE` *(since v26.11.1)*. `vertex.moveToType()` and `vertex.moveToBucket()` of a remote vertex now go to the server (the immutable vertex used to call itself and the mutable one reached for an embedded database it has none of), and raise `RecordNotFoundException` for a vertex that no longer exists, as `countEdges()` does.

An edge is the one write that has to return what it created, so `Vertex.newEdge()` cannot answer "skipped": when a `BeforeRecordCreateListener` refuses the edge it raises `RecordVetoedException`, a `ValidationException` that every protocol reports as the request being refused and not as a server fault, and nothing is stored or linked. The same refusal fails an openCypher `CREATE` and a Gremlin `addEdge()`, while SQL `CREATE EDGE`, like SQL `INSERT`, skips the refused edge and returns no row for it. A Java caller that counts its own writes uses `GraphEngine.newEdgeReportingVeto()` (through `DatabaseInternal.getGraphEngine()`), which returns `null` for a refused edge *(since v26.11.1)*. A lightweight edge owns no record, so no record listener is ever consulted for it. An asynchronous create (`database.async().createRecord()`, `newEdge()`) of a record a listener refused fires neither its success callback nor its error callback, because nothing failed and nothing was created: an application that waits on the callback to learn the outcome must expect that silence for a refused record.

A full scan of a type or bucket (`iterateType()`, `iterateBucket()`, or a query that scans a type) fires the read listeners when it reads each record, once per record, even for records your code then never reads. A record a before-read or after-read listener filters out is not returned by the scan.

A read listener runs while the database holds its shared lock, and a schema change (creating or altering a type, a property or an index) needs the exclusive one, which a thread holding the shared lock can never be granted. A schema change issued from a before-read or after-read listener therefore fails with a `DatabaseOperationException` ("a read lock cannot be upgraded"), and the read fails with it. Before v26.10.2 the thread waited for itself forever. Run the schema change outside the listener.

An after-read listener may return other content than the stored one (a decryption, for instance). A record read that way keeps the stored content it was read from, and a later update or delete of it, or of its `modify()` copy, is checked against that stored content, so it is refused with the retryable `ConcurrentModificationException`

> only when another transaction changed the record after the read. The index entries and the external property values of a deleted record are removed by their stored values, and the index changes of an update are computed from them, even when the listener rewrote an indexed property. Before v26.12.1 every update and delete of such a record was refused as a concurrent modification, retry after retry *(since v26.12.1)*.

The typical use cases for the listeners are:

- listen before a create or update to enhance the record with additional properties
- listen before a create or update to validate properties and in case the record is not valid, returning false or an exception to avoid the operation is executed
- execute cascade operations. This is the typical use case for `AfterRecordDeleteListener` where a cascade deletion of multiple connected records can be executed
- listen to after create, read, update and delete operations to propagate changes to the external or the webapp via web-socket. This allows to have a reactive application that doesn't poll the database for changes, but rather listens and receives updates as soon as they occur
- implement custom profiling on changes to the database (by implementing "before" listeners)

Example of `before-record-create` listener where vertices with "validated" field equal to false cannot be saved (callback returns false):

```
database.getEvents().registerListener((BeforeRecordCreateListener)
  record -> record instanceof Vertex && record.asVertex().getBoolean("validated"));
```

The same by only for vertex type "Client":

```
database.getSchema().getType("Client").getEvents().registerListener((BeforeRecordCreat
eListener)
  record -> record.asVertex().getBoolean("validated"));
```

4.9. Graph Batch

The `GraphBatch` is a high-performance bulk graph loader designed for importing large volumes of edges into an ArcadeDB graph. It achieves **8-12x speedup** over the standard edge creation API by buffering edges in memory, sorting them by vertex for sequential I/O, parallel edge connection, and deferring incoming edge connection to a single optimized pass.

4.9.1. Quick Start

```
Database db = new DatabaseFactory("mydb").create();
db.transaction(() -> {
  db.getSchema().createVertexType("Person");
  db.getSchema().createEdgeType("KNOWS");
```

```
});

try (GraphBatch batch = db.batch()
    .withExpectedEdgeCount(1_000_000)
    .build()) {

  // Phase 1: Create vertices in bulk
  RID[] vertices = batch.createVertices("Person", 100_000);

  // Phase 2: Buffer edges (auto-flushes when batch is full)
  for (int i = 0; i < edgeCount; i++)
    batch.newEdge(vertices[src[i]], "KNOWS", vertices[dst[i]]);

} // close() flushes remaining edges + connects all incoming edges
```

All edges (both outgoing and incoming) are fully connected when `close()` returns.

4.9.2. How It Works

The importer operates in phases for maximum throughput:

1. **Buffer**: Edges are stored in flat primitive arrays (minimal GC pressure)
2. **Sort**: On flush, outgoing edges are sorted by source vertex, converting random I/O into sequential page access
3. **Connect OUT**: Outgoing edges are connected to source vertices in sorted order, using vectorized segment writes
4. **Accumulate IN**: Incoming edges are deferred to an in-memory buffer across flushes
5. **Connect IN at close()**: All deferred incoming edges are sorted by destination vertex and connected in a single pass
6. **Batch vertex update**: All vertex head-chunk pointers are updated in one final sorted pass

4.9.3. Edges with Properties

The importer supports edges with properties. Pass them as key-value pairs:

```
try (GraphBatch batch = db.batch()
    .withExpectedEdgeCount(500_000)
    .withLightEdges(false)  // edges have properties
    .build()) {

  RID[] vertices = batch.createVertices("Person", 50_000);

  for (int i = 0; i < edgeCount; i++)
    batch.newEdge(vertices[src[i]], "KNOWS", vertices[dst[i]],
        "weight", weights[i], "timestamp", timestamps[i]);
}
```

Property edges use a bulk record creation path that bypasses per-record overhead, achieving nearly the same throughput as light edges. Edges of a type with declared properties still get what `Vertex.newEdge()` gives them: values converted to the declared type, defaults applied and constraints (mandatory, min/max, regexp...) validated. A violation is raised when the buffered edges are flushed, not at the `newEdge()` call. Properties can be passed as name/value pairs or as a single `Map`. With `withLightEdges(true)`, an edge without properties is still stored as a regular edge if its type declares a property with a default or a mandatory property.

(Since v26.10.2) An edge type with an `EXTERNAL` property, declared on the type or inherited from a super type, does not use the bulk record creation path: its values are stored in the paired external bucket, so those edges are saved one at a time and the import is slower than for other edge types. Before 26.10.2 the bulk path kept those values inside the edge record instead of the external bucket.

4.9.4. Builder Options

Method	Default	Description
`withExpectedEdgeCount(int)`	—	Auto-tunes the batch size to `edgeCount` (clamped to 100K-5M). A single flush is optimal for most workloads.
`withBatchSize(int)`	100,000	Maximum edges buffered before auto-flush. Overrides auto-tuning if both are set.
`withLightEdges(boolean)`	`false`	If true, edges without properties are created as light edges (no record stored, only connectivity pointers). Deprecated: the edge type must declare `LIGHTWEIGHT`, otherwise an `IllegalArgumentException` is raised.
`withBidirectional(boolean)`	`true`	If true, incoming edges are connected for the edge types declared bidirectional; an edge type declared `UNIDIRECTIONAL` never gets them, whatever this flag says. Set to false to load only unidirectional edge types: `newEdge()` then refuses an edge of a bidirectional type with an `IllegalArgumentException`.
`withWAL(boolean)`	`false`	Write-Ahead Logging during import. Disable for maximum speed; enable if crash recovery is needed.
`withCommitEvery(int)`	0 (WAL off) / 50,000 (WAL on)	Edges per transaction commit within a flush. Set to 0 to commit once per flush. Auto-set to 0 when WAL is disabled.
`withEdgeListInitialSize(int)`	2048	Initial size in bytes for edge segments. Larger values reduce segment splits for high-degree vertices.

Method	Default	Description
`withPreAllocateEdgeChunks(boolean)`	`true`	Pre-allocate empty edge segments at vertex creation time, eliminating lazy allocation cost. Each segment takes `withEdgeListInitialSize` bytes on disk even if the vertex never gets an edge, so turn it off when loading many vertices whose edges come later: the edges then get segments of exactly the size they need.
`withMaxVertices(long)`	0 (no bound)	Upper bound on the distinct vertices the batch's edges can touch. The batch keeps a small entry per touched vertex until `close()`; the bound stops that table from growing past what it needs. Nothing is allocated up front *(since 26.10.2)*.
`withParallelFlush(boolean)`	`true`	Parallel edge connection partitioned by bucket. Uses the database async executor to connect edges across buckets concurrently.

4.9.5. Auto-Tuning Batch Size

The `withExpectedEdgeCount()` method automatically selects the optimal batch size based on benchmarks. A single flush (batch size = edge count) delivers the best throughput, so the batch size is set to `edgeCount`, clamped between 100K and 5M:

Expected Edges	Auto-tuned Batch	Approx. RAM
< 100K	100K (minimum)	~200MB
100K - 5M	= edge count (single flush)	~300MB-1GB
> 5M	5M (maximum)	~1GB

You can always override with `withBatchSize()` if you know your workload characteristics.

4.9.6. Key Methods

Method	Description
`createVertices(typeName, count)`	Creates vertices in bulk within a single transaction. Returns an array of RIDs for use with `newEdge()`.
`newEdge(srcRID, edgeType, dstRID, properties…)`	Buffers an edge for batch processing. Automatically triggers a flush when the batch size is reached.

Method	Description
`newEdgeWithHints(srcRID, edgeType, dstRID, hints, properties…)`	Same as `newEdge()`, for a caller that knows an endpoint has no edges yet in that direction: `GraphBatch.SOURCE_HAS_NO_OUT_EDGES` and/or `GraphBatch.DESTINATION_HAS_NO_IN_EDGES`, or 0. The batch then starts the vertex's edge list without reading the vertex, a saving of one random disk read per distinct vertex and flush when the graph is larger than the memory. Pass the hint with the first edge of the vertex you hand to a batch, never later: a wrong hint is not detected in production and leaves the edges the vertex already had unreachable from it (with assertions enabled, `-ea`, the batch checks every hint and fails instead). `GraphImporter` uses it for the vertices it creates. *Since v26.11.1*
`getVertexHeadReads()`	How many vertices the flushes read to find the edge list they append to: one per distinct vertex and flush that the batch neither created nor was given a hint for. *Since v26.11.1*
`getSegmentHeadReads()`	How many edge-list segments the flushes read back to append to. A segment the batch wrote itself is never read again: the batch sizes it to its edges, so the next edges of that vertex go to a new segment linked to it. Only segments the batch found on disk are counted. *Since v26.11.1*
`flush()`	Manually flushes all buffered edges. Outgoing edges are sorted and connected; incoming edges are deferred.
`close()`	Flushes remaining edges, connects all deferred incoming edges in a single sorted pass, and batch-updates vertex head-chunk pointers.

4.9.7. Transactions

The batch manages its own transactions: every flush is several durable steps, committed by the batch itself. For this reason `createVertices()`, `flush()`, `close()` and any `newEdge()` call that fills the buffer refuse to run inside a transaction you opened with `begin()` or `transaction()`. They throw an `IllegalStateException` and leave your transaction untouched.

Commit or roll back your own transaction first. If `close()` is refused, the batch stays open with its pending work, and calling `close()` again outside the transaction writes it.

The batch's relaxed WAL settings apply only to the batch's own transactions, never to the ones you run between batch calls.

4.9.8. One Batch at a Time

A database grants one batch at a time, because two overlapping batches silently lose edges. Building a second batch while another one is open throws a `DatabaseOperationException` and nothing of the second load runs: build it again once the first batch is closed. *Since v26.12.1* the exception is the

subtype `GraphBatchInProgressException`, so code that catches `DatabaseOperationException` keeps working. Over HTTP the same refusal answers `409`, see HTTP Batch Import.

4.9.9. Recovering an Interrupted Batch

Each flush commits the edge records it created, but the vertices' head-chunk pointers are only written at `close()`. If the process dies before `close()` (`kill -9`, an OOM kill, a power loss), the edges committed by the earlier flushes are on disk but not reachable by traversal: `CHECK DATABASE` reports them as `ORPHAN RECORD` and counts them under `unreachableEdgeRecords`.

Since v26.11.1 the supported recovery is `CHECK DATABASE FIX`. It relinks each of those edge records into the outgoing list of its `@out` vertex and, for a bidirectional edge type, into the incoming list of its `@in` vertex, so the committed edges become traversable without reloading them (`relinkedOrphanEdgeRecords` in the result). Edges still buffered in memory when the process died were never written and must be loaded again. Use `CHECK DATABASE FIX DELETE ORPHANS` instead to delete the committed edges and reload them all. Before v26.11.1 a plain `FIX` left those records untouched, and deleting them was the only repair.

4.9.10. Performance

Benchmark results on a single machine (500K edges, 50K vertices):

Method	Time (ms)	Edges/sec	Speedup
Standard API (tx batches of 1000)	11,933	42K	1.0x
GraphBatch (light edges)	**1,028**	**486K**	**11.6x**
Standard API + properties (int + long)	13,385	37K	1.0x
GraphBatch + properties	**1,584**	**316K**	**8.5x**

4.9.11. Importing into an Existing Graph

The importer can also connect edges to vertices that already exist in the database (not just vertices created by `createVertices()`). Simply pass existing vertex RIDs to `newEdge()`:

```
try (GraphBatch batch = db.batch()
    .withExpectedEdgeCount(100_000)
    .build()) {

  // Use existing vertex RIDs from queries or lookups
  for (int i = 0; i < edgeCount; i++)
    batch.newEdge(existingVertexRIDs[src[i]], "FOLLOWS", existingVertexRIDs[dst[i]]);
}
```

4.9.12. HTTP Batch Endpoint

GraphBatch is also available over HTTP for non-Java clients. The POST /api/v1/batch endpoint

accepts JSONL or CSV input and uses GraphBatch under the hood, with all builder options exposed as query parameters. See HTTP Batch Import for full details and examples in curl, Python, and JavaScript.

4.9.13. Relationship with GraphImporter

The GraphImporter is a declarative, JSON-driven graph importer built on top of `GraphBatch`. It adds file parsing (XML, CSV, JSONL), schema auto-creation, foreign key resolution, and post-import commands.

Use `GraphBatch` directly when you need full programmatic control over vertex and edge creation. Use `GraphImporter` when you want to import graph data from files using a JSON configuration.

4.10. Working with Vector Embeddings

ArcadeDB provides robust support for vector embeddings through the LSMVectorIndex, a persistent vector index built on ArcadeDB's LSM Tree architecture and powered by the JVector 4.0.0 library. LSMVectorIndex offers efficient storage, retrieval, and similarity search for vector embeddings with full transaction support and automatic persistence.

4.10.1. Key Features

The LSMVectorIndex implementation provides:

- **Persistent Storage**: Vector indexes are stored on disk with automatic page management and compaction
- **Transaction Support**: Full ACID compliance with automatic persistence on transaction commit
- **Multiple Similarity Functions**: Supports COSINE (default), DOT_PRODUCT, and EUCLIDEAN distance metrics
- **SQL Integration**: Create and query vector indexes using SQL commands
- **Automatic Compaction**: Efficiently reclaims disk space through automatic compaction of immutable pages
- **High Performance**: Leverages LSM Tree benefits for write efficiency and space optimization at scale
- **Configurable Parameters**: Tune maxConnections and beamWidth for optimal ANN search performance
- **JVector Library**: Built on JVector 4.0.0 for state-of-the-art vector search capabilities

4.10.2. Creating Vector Indexes with SQL

The simplest way to create a vector index is through SQL. This approach is recommended for most use cases as it provides a declarative syntax and automatic schema management.

Basic Vector Index Creation

Create a basic LSMVectorIndex for similarity search:

```
-- Create vertex type and property
CREATE VERTEX TYPE Document;
CREATE PROPERTY Document.embedding LIST OF FLOAT;

-- Create vector index with 384 dimensions using COSINE similarity and INT8
quantization
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE",
  "quantization": "INT8"
};
```

INT8 quantization is recommended for production use. It stores vectors in compact index pages instead of full documents, providing 4x lower memory usage with negligible accuracy loss and usually lower search latency; the latency gain depends on the data and the recall target. See concepts/vector-search.pdf.

Configuring Similarity Functions

Choose the appropriate similarity function for your use case:

COSINE Similarity (default) - Best for normalized vectors, commonly used with text embeddings:

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE"
};
```

DOT_PRODUCT - Efficient for normalized vectors, faster than cosine:

```
CREATE INDEX ON Image (featureVector) LSM_VECTOR METADATA {
  "dimensions": 512,
  "similarity": "DOT_PRODUCT"
};
```

EUCLIDEAN - Measures absolute distance, useful for spatial data:

```
CREATE INDEX ON Product (attributes) LSM_VECTOR METADATA {
  "dimensions": 256,
  "similarity": "EUCLIDEAN"
};
```

Performance Tuning Parameters

For large-scale deployments, tune performance parameters:

```
CREATE INDEX ON VectorVertex (embedding) LSM_VECTOR METADATA {
  "dimensions": 1024,
  "similarity": "COSINE",
  "quantization": "INT8",           -- 4x less memory, usually lower latency
(recommended)
  "maxConnections": 32,             -- Higher values improve recall but increase
memory
  "beamWidth": 200,                 -- Higher values improve accuracy but reduce
speed
  "neighborOverflowFactor": 1.2,    -- Candidate neighbor pool multiplier (default:
1.2)
  "alphaDiversityRelaxation": 1.2,  -- Diversity vs distance trade-off (default: 1.2)
  "addHierarchy": true              -- Enable HNSW hierarchical layers (default:
false)
};
```

Parameter Guidelines:

- `dimensions`: Must match your embedding model's output dimension (required)
- `similarity`: Choose based on your embedding model and use case (default: COSINE)
- `maxConnections`: Graph connections per node, per layer — the Vamana degree (default: 32). Not doubled at the base layer like hnswlib's `M`, so set `maxConnections = 2 * M` to match an hnswlib config. Increase to 48-64 for better recall in large datasets
- `beamWidth`: Search depth during index construction (default: 100). Increase to 200-400 for more accurate searches
- `neighborOverflowFactor`: Controls how many extra candidate neighbors are considered during graph building (default: 1.2, range: 1.0-1.5). Higher values improve graph quality but increase build time
- `alphaDiversityRelaxation`: Trade-off between strict distance ordering and diversity in graph connections (default: 1.2, range: 1.0-1.5). Higher values prioritize diversity, improving recall for complex queries
- `addHierarchy`: Controls whether to build HNSW hierarchical layers on top of the base Vamana index (default: false). See Hierarchical vs Flat Index Structure for detailed guidance
- `buildGraphNow`: When creating the index via SQL, controls whether the HNSW graph is built immediately (default: `true`). Set to `false` to defer graph construction to the first search. This is a creation-time directive only — it is not persisted as index metadata.

When using the Java API directly, you can call `buildVectorGraphNow()` on any `LSMVectorIndex` instance to trigger an immediate graph build at any time:

```
LSMVectorIndex vectorIndex = (LSMVectorIndex) database.getSchema().getIndexByName
("MyType[embedding]");
vectorIndex.buildVectorGraphNow();
```

This is lightweight compared to `REBUILD INDEX` — it only rebuilds the HNSW graph topology from already-indexed vectors, without dropping or re-scanning the underlying index data.

Searches keep running while `buildVectorGraphNow()` rebuilds an index that already has a graph in memory: they answer from that graph plus the vectors written since, and switch to the new graph the moment it is published, exactly as during an automatic background rebuild. Only an index with no graph in memory yet (for example one not searched since the database was opened) makes its searches wait for the build, since there is nothing else to answer from. Before 26.11.1, an explicit rebuild made every search wait for the whole build when nothing wrote to the index during it.

Hierarchical vs Flat Index Structure

ArcadeDB's LSMVectorIndex uses JVector, which combines the hierarchical structure from HNSW (Hierarchical Navigable Small World) with the Vamana algorithm from DiskANN. The `addHierarchy` parameter determines whether to build a multi-layer hierarchical structure or use a flat single-layer graph.

How It Works

When `addHierarchy` is enabled:

- **Multiple Layers**: Nodes are assigned to different hierarchy levels using HNSW's exponential decay sampling strategy
- **Top-Down Search**: Searches start from higher abstraction levels and progressively narrow down to specific neighbors
- **Fast Seeding**: Upper layers use in-memory adjacency lists for rapid navigation without disk I/O
- **Base Layer**: The bottom layer remains disk-based with inline vector representations

When `addHierarchy` is disabled (default):

- **Single Layer**: All nodes exist at level 0 in a flat Vamana graph structure
- **Direct Search**: Searches operate directly on the base layer without hierarchical navigation
- **Simpler Structure**: Reduced complexity with fewer memory allocations during construction

When to Enable Hierarchy (addHierarchy=true)

Enable hierarchical layers when:

1. **Challenging Search Scenarios**: Complex datasets with diverse cluster distributions or high-dimensional embeddings (1536+ dimensions)
2. **Robustness Over Speed**: Applications where consistent search quality across different query patterns is critical
3. **Large-Scale Deployments**: Datasets with 100K+ vectors where hierarchical navigation provides better scaling
4. **Diverse Query Patterns**: Workloads with varying query characteristics that benefit from adaptive search depth

```
-- Recommended for large, complex datasets
CREATE INDEX ON ComplexDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "maxConnections": 16,
  "beamWidth": 100,
  "addHierarchy": true  -- Enable for robustness
};
```

When to Keep Flat Structure (addHierarchy=false)

Use the default flat structure when:

1. **Small to Medium Datasets**: Less than 100K vectors where flat search is efficient
2. **Simple Distributions**: Well-clustered data where flat Vamana performs optimally
3. **Memory Constraints**: Limited resources during index construction
4. **Faster Build Times**: When index creation speed is prioritized over search robustness

```
-- Default for most use cases
CREATE INDEX ON StandardDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 16,
  "beamWidth": 100
  -- addHierarchy defaults to false
};
```

Performance Trade-offs

Aspect	Flat (addHierarchy=false)	Hierarchical (addHierarchy=true)
Index Build Time	Faster (baseline)	10-20% slower
Memory During Build	Lower	15-25% higher
Index Size on Disk	Smaller (baseline)	5-15% larger
Search Robustness	Good for well-clustered data	**Excellent** for diverse patterns
Predictable Performance	May vary across query types	**More consistent** across queries
Recommended For	<100K vectors, simple distributions	100K+ vectors, complex patterns

Accuracy and Recall

Based on JVector documentation and benchmarks, the hierarchical structure provides improved

robustness rather than raw speed improvements. Recall@10 typically remains similar between flat and hierarchical structures for well-behaved datasets, but hierarchical indexing shows **significant advantages** in:

- **Edge Cases**: Queries far from cluster centers or in sparse regions
- **Consistency**: More stable recall across different query distributions
- **Challenging Scenarios**: High-dimensional spaces (1536+ dims) with complex manifold structures

Memory Impact Example

For 1 million vectors with 1536 dimensions:

- **Flat Structure**: ~6.1 GB base layer
- **Hierarchical Structure**: ~6.8 GB total (6.1 GB base + ~700 MB hierarchy layers)
- **Memory Overhead**: ~11% additional disk space

Build Time Impact

Example build times for 100K vectors (1536 dimensions, maxConnections=16, beamWidth=100):

- **Flat Structure**: ~45 seconds
- **Hierarchical Structure**: ~52 seconds (~15% slower)

Practical Guidelines

1. **Start with Default (false)**: For most applications under 100K vectors, the flat structure provides excellent performance with faster build times
2. **Enable for Scale**: When dataset exceeds 100K vectors or dimensionality exceeds 1024, consider enabling hierarchy for better robustness
3. **Test Both Configurations**: Benchmark search quality on your specific dataset and query patterns. Enable hierarchy if you observe:
   - Inconsistent recall across different query types
   - Poor performance on queries far from training data
   - Degraded search quality in sparse regions
4. **Combine with Other Optimizations**: Hierarchy works well with quantization and graph storage:

```
-- Optimized large-scale configuration
CREATE INDEX ON LargeDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "quantization": "INT8",           -- 4x memory reduction
  "maxConnections": 16,
  "beamWidth": 100,
  "addHierarchy": true,             -- Robustness for large scale
```

```
  "storeVectorsInGraph": true,       -- Fast co-located access
  "locationCacheSize": 200000,
  "mutationsBeforeRebuild": 1000
};
```

This configuration provides: - Hierarchical robustness for complex search patterns - 4x memory reduction through INT8 quantization - Efficient vector access through graph storage - Total overhead: ~20% larger index size with 4x memory savings

Related JVector Documentation

For deeper technical details about the hierarchical structure, see:

- JVector GitHub Repository
- GraphIndexBuilder Source Code

Advanced Tuning Profiles

Default Configuration (recommended for most use cases):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 16,
  "beamWidth": 100,
  "neighborOverflowFactor": 1.2,
  "alphaDiversityRelaxation": 1.2
};
```

High Recall Configuration (accuracy-critical applications):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 32,
  "beamWidth": 200,
  "neighborOverflowFactor": 1.4,
  "alphaDiversityRelaxation": 1.3,
  "addHierarchy": true
};
```

This configuration provides ~98% recall@10 with improved robustness across diverse query patterns, but requires 2-3x longer build time and ~50% more memory.

Fast Indexing Configuration (real-time ingestion, large-scale ETL):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
```

```
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 12,
  "beamWidth": 80,
  "neighborOverflowFactor": 1.1,
  "alphaDiversityRelaxation": 1.1
};
```

This configuration provides 2x faster indexing with 5-10% lower recall.

Memory Constrained Configuration (edge deployments, resource-limited servers):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 8,
  "beamWidth": 100,
  "neighborOverflowFactor": 1.2,
  "alphaDiversityRelaxation": 1.2
};
```

This configuration reduces memory usage while maintaining acceptable recall

4.10.3. Memory and Performance Tuning

ArcadeDB provides three global configuration settings to control memory consumption and performance of LSM Vector indexes. These settings can be configured globally via database settings or per-index through metadata.

Configuration Parameters

Location Cache Size (`locationCacheSize`)

Controls the maximum number of vector location metadata entries cached in memory. Each entry uses approximately 56 bytes.

- **Default**: `-1` (unlimited, backward compatible)
- **Per-index metadata**: `"locationCacheSize": 100000`
- **Global setting**: `arcadedb.vectorIndex.locationCacheSize`
- **Recommended**: 100,000 for datasets with 1M+ vectors (~5.6 MB RAM)

Graph Build Cache Size (`graphBuildCacheSize`)

Controls the maximum number of vectors cached during HNSW graph construction. RAM usage = `cacheSize x (dimensions x 4 + 64)` bytes.

- **Default**: `0` (auto-sized: the whole corpus, as long as it fits `arcadedb.vectorIndex.graphBuildCacheMaxHeapPercent` of the `-Xmx` ceiling)

- **Per-index metadata**: `"graphBuildCacheSize": 5000`
- **Global setting**: `arcadedb.vectorIndex.graphBuildCacheSize`
- **Recommended**: leave at `0` and let the heap percentage bound it. Pin an explicit value only to cap the build's footprint on a heap shared with other work; a cache smaller than the corpus makes the build re-read vectors it has already seen, which is by far the largest term in build time

Mutations Before Rebuild (`mutationsBeforeRebuild`)

Number of mutations (inserts/updates/deletes) before rebuilding the HNSW graph index. Higher values reduce rebuild cost but may return slightly stale results. Since v26.12.1 this also applies when a search finds a small graph (under 1,000 vectors in a bucket): it is no longer rebuilt after every write, and the vectors written meanwhile are found exactly through the delta buffer.

- **Default**: `100` (balanced)
- **Per-index metadata**: `"mutationsBeforeRebuild": 200`
- **Global setting**: `arcadedb.vectorIndex.mutationsBeforeRebuild`
- **Recommended**: 50-200 for read-heavy workloads, 200-500 for write-heavy workloads

Inactivity Rebuild Timeout (`inactivityRebuildTimeoutMs`)

When mutations exist but haven't reached the rebuild threshold, a timer starts after the last mutation. If no new mutations arrive within this window, the graph is rebuilt asynchronously. This ensures buffered vectors are flushed and persisted even during low-volume ingestion.

- **Default**: `15000` (15 seconds)
- **Per-index metadata**: `"inactivityRebuildTimeoutMs": 30000`
- **Global setting**: `arcadedb.vectorIndex.inactivityRebuildTimeoutMs`
- **Recommended**: 10000-30000 for low-volume ingestion, 0 to disable

Store Vectors in Graph (`storeVectorsInGraph`)

Controls whether vectors are stored inline within the JVector graph file (`.vecgraph`) alongside the HNSW topology. When enabled, vectors are co-located with graph data, eliminating expensive document lookups during search operations.

- **Default**: `false` (vectors fetched from documents)
- **Per-index metadata**: `"storeVectorsInGraph": true`
- **Global setting**: none, `storeVectorsInGraph` is set per index only
- **When to enable**: Large indexes (100K+ vectors), high search throughput requirements, RAM-constrained environments
- **When to disable**: Small indexes (<10K vectors), frequent vector updates, datasets where document cache is effective

The inline vectors are stored as full-precision `float32` whatever the index

> quantization is - quantization applies to the index data, not to this copy. Size the `.vecgraph` file accordingly: roughly `vectors x dimensions x 4 bytes`, plus the graph topology. A 1M x 768 index is therefore about 3 GB on disk even with `INT8` quantization enabled.

Performance Characteristics:

Index Size	storeVectorsInGraph=false	storeVectorsInGraph=true	Recommendation
Small (<10K vectors)	Fast (document cache effective)	Slower (I/O overhead dominates)	**Disable** - use document cache
Medium (10K-100K vectors)	Moderate (some cache misses)	Moderate (balanced I/O)	Test both configurations
Large (100K+ vectors)	Slow (cache thrashing, page faults)	Fast (co-located data, sequential reads)	**Enable** - reduces RAM pressure

Memory Trade-offs:

- **Disk Usage**: Enabling this option duplicates vectors on disk (stored in both documents and graph file)
- **RAM Usage**: Significantly reduces RAM consumption by eliminating document page cache thrashing during search
- **Search Latency**: Reduces latency for large indexes by avoiding RID lookups and document deserialization

Combining with Quantization:

When `storeVectorsInGraph=true` is combined with quantization, vectors are stored in their quantized form within the graph file:

- `quantization=INT8` + `storeVectorsInGraph=true`: Vectors stored as int8 (1 byte per dimension)
- `quantization=BINARY` + `storeVectorsInGraph=true`: Vectors stored as binary (1 bit per dimension)
- `quantization=NONE` + `storeVectorsInGraph=true`: Vectors stored as float32 (4 bytes per dimension)

This combination provides maximum benefit: reduced memory usage from quantization + fast access from co-location.

Metrics Tracking:

Three new metrics track vector fetch sources to help tune this setting:

- `vectorFetchFromGraph`: Vectors read from `.vecgraph` file (when `storeVectorsInGraph=true`)
- `vectorFetchFromDocuments`: Vectors read via RID lookups from documents
- `vectorFetchFromQuantized`: Vectors read from quantized pages

Access metrics via Java API:

```
Map<String, Long> stats = lsmIndex.getStats();
System.out.println("Fetched from graph: " + stats.get("vectorFetchFromGraph"));
System.out.println("Fetched from docs: " + stats.get("vectorFetchFromDocuments"));
```

Memory Tuning Examples

Maximum Memory Efficiency (Edge deployments, resource-constrained environments):

```
-- Configure globally for all vector indexes
ALTER DATABASE `arcadedb.vectorIndex.locationCacheSize` 50000;
ALTER DATABASE `arcadedb.vectorIndex.graphBuildCacheSize` 5000;

-- Or configure per-index for small datasets
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 8,
  "beamWidth": 100,
  "locationCacheSize": 50000,        -- Limit location cache: ~2.8 MB
  "graphBuildCacheSize": 5000,       -- Limit graph building: ~15 MB peak
  "mutationsBeforeRebuild": 150,     -- Moderate rebuild frequency
  "storeVectorsInGraph": false       -- Use document cache for small datasets
};
```

Memory Impact: For 1M vectors with 768 dimensions: - Location cache: 50K entries = ~2.8 MB (vs ~56 MB unlimited) - Graph build cache: 5K vectors = ~15 MB (vs ~3 GB unlimited) - **Total savings: ~98.5% memory reduction during graph building**

Balanced Configuration (Production workloads with moderate scale):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 16,
  "beamWidth": 100,
  "locationCacheSize": 100000,       -- Balance memory vs performance
  "graphBuildCacheSize": 10000,      -- Default (30 MB for 768-dim)
  "mutationsBeforeRebuild": 100,     -- Standard rebuild frequency
  "storeVectorsInGraph": false       -- Use document cache for moderate scale
};
```

Memory Impact: For 1M vectors with 768 dimensions: - Location cache: 100K entries = ~5.6 MB (90% reduction) - Graph build cache: 10K vectors = ~30 MB (99% reduction)

Large-Scale Configuration with Inline Vector Storage (100K+ vectors, high-throughput search):

```
-- Optimal for large indexes: combine quantization + graph storage
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "quantization": "INT8",            -- 4x memory reduction
  "maxConnections": 16,
  "beamWidth": 100,
  "locationCacheSize": 200000,       -- Larger cache for hot vectors
  "graphBuildCacheSize": 10000,      -- Standard build cache
  "mutationsBeforeRebuild": 1000,    -- Reduce rebuild frequency (vectors in graph)
  "storeVectorsInGraph": true        -- Co-locate vectors with graph topology
};
```

Benefits for Large-Scale: - Eliminates RID lookups during search (no document page cache thrashing) - Reduces RAM pressure by ~70-80% compared to document fetching at scale - Vectors co-located with graph topology (sequential reads, better I/O patterns) - Combined with INT8 quantization: 4x smaller graph file + fast access - Disk trade-off: Vectors duplicated (~6 GB for 1M × 1536-dim vectors)

When to Use: - Index size: 100K+ vectors - Search workload: High QPS (queries per second) - RAM constraint: Limited memory for page cache - Update frequency: Moderate to low (vectors not frequently updated)

Maximum Performance (Unlimited memory, prioritize speed):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 32,
  "beamWidth": 200,
  "locationCacheSize": -1,           -- Unlimited cache (no eviction)
  "graphBuildCacheSize": 50000,      -- Large cache: ~150 MB for 768-dim
  "mutationsBeforeRebuild": 50,      -- Frequent rebuilds for freshness
  "storeVectorsInGraph": true        -- Enable for large indexes to reduce cache
pressure
};
```

Performance Impact: Best query performance with highest memory usage. For large indexes (100K+ vectors), enabling `storeVectorsInGraph` reduces document cache pressure even with unlimited location cache.

Performance Tuning Examples

Write-Heavy Workload (High ingestion rate, acceptable search latency):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
```

```
  "maxConnections": 12,
  "beamWidth": 80,
  "locationCacheSize": 100000,
  "graphBuildCacheSize": 10000,
  "mutationsBeforeRebuild": 300,     -- Reduce rebuild frequency for write throughput
  "storeVectorsInGraph": false       -- Disable: frequent updates make graph storage
costly
};
```

Note: For write-heavy workloads with frequent vector updates, keep `storeVectorsInGraph=false` to avoid expensive graph rebuilds. Each rebuild must rewrite all vectors when this option is enabled.

Read-Heavy Workload (Search-intensive, prioritize freshness):

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "maxConnections": 24,
  "beamWidth": 150,
  "locationCacheSize": 200000,       -- Larger cache for hot vectors
  "graphBuildCacheSize": 10000,
  "mutationsBeforeRebuild": 50,      -- Frequent rebuilds for fresh results
  "storeVectorsInGraph": true        -- Enable: search-heavy benefits from co-location
};
```

Note: For search-intensive workloads with large indexes (100K+ vectors), enabling `storeVectorsInGraph` significantly reduces search latency by eliminating document lookups. The rebuild cost is amortized across many search operations.

Global Configuration via SQL

Set global defaults for all vector indexes in the database:

```
-- Configure global defaults
ALTER DATABASE `arcadedb.vectorIndex.locationCacheSize` 100000;
ALTER DATABASE `arcadedb.vectorIndex.graphBuildCacheSize` 10000;
ALTER DATABASE `arcadedb.vectorIndex.mutationsBeforeRebuild` 100;
ALTER DATABASE `arcadedb.vectorIndex.inactivityRebuildTimeoutMs` 15000;

-- View current settings (see the `settings` array of the result)
SELECT FROM schema:database;
```

Per-index metadata settings override global defaults. Use global settings to establish baseline memory limits across all indexes, then tune individual indexes as needed. `storeVectorsInGraph` is not a global setting: it defaults to `false` and you enable it per index, selectively for large indexes (100K+ vectors).

Memory Consumption Reference

For 1 million vectors with 768 dimensions:

Configuration	Location Cache	Graph Build Cache	Total Peak Memory
Unlimited (legacy)	~56 MB	~3 GB	~3.06 GB
Recommended (100K/10K)	~5.6 MB	~30 MB	~35.6 MB
Memory Constrained (50K/5K)	~2.8 MB	~15 MB	~17.8 MB
Savings vs Unlimited	90-95%	99%	98.8%

Formula for Graph Build Cache:

```
RAM (MB) = cacheSize x (dimensions x 4 + 64) / 1024 / 1024
```

Example: - 10,000 vectors × 768 dimensions = 10,000 × (768 × 4 + 64) / 1024 / 1024 ≈ 30 MB - 10,000 vectors × 1536 dimensions = 10,000 × (1536 × 4 + 64) / 1024 / 1024 ≈ 59 MB

Best Practices for Tuning

1. **Start with Defaults**: Default settings (locationCacheSize=-1, graphBuildCacheSize=10000, mutationsBeforeRebuild=100, storeVectorsInGraph=false) work well for most use cases under 100K vectors.
2. **Monitor Memory Usage**: For large-scale deployments (1M+ vectors), set explicit limits to prevent unbounded growth.
3. **Scale Graph Build Cache with Dimensions**: Higher dimensionality requires proportionally more memory during graph building.
4. **Balance Rebuild Frequency**: Lower `mutationsBeforeRebuild` provides fresher search results but increases CPU cost. Higher values improve write throughput at the cost of search staleness.
5. **Per-Index Tuning**: Use global settings as baseline, then override for specific indexes based on workload characteristics.
6. **Cache Hit Monitoring**: If queries become noticeably slower after setting `locationCacheSize` limits, increase the cache size or use `-1` (unlimited) for that index.
7. **Write vs Read Trade-offs**: Write-heavy workloads benefit from higher `mutationsBeforeRebuild` (200-500), while read-heavy workloads benefit from lower values (50-100).
8. **Enable Graph Storage for Large Indexes**: For indexes with 100K+ vectors and read-heavy workloads, enable `storeVectorsInGraph=true` to eliminate document lookups and reduce RAM pressure. Keep disabled for small indexes (<10K) and write-heavy workloads.
9. **Combine Quantization + Graph Storage**: For maximum efficiency on large indexes, use `quantization=INT8` (or BINARY) with `storeVectorsInGraph=true`. This provides 4x-32x memory reduction plus fast co-located access.
10. **Monitor Fetch Sources**: Use the metrics API to track `vectorFetchFromGraph`, `vectorFetchFromDocuments`, and `vectorFetchFromQuantized` to verify your configuration is working

as expected. If `vectorFetchFromDocuments` remains high with `storeVectorsInGraph=true`, investigate graph loading issues.

11. **Test Before Production**: Benchmark both `storeVectorsInGraph=true` and `storeVectorsInGraph=false` on your actual dataset and workload. Performance characteristics vary by scale, and what works for 1K vectors may differ significantly from 1M vectors.

4.10.4. Vector Quantization

Vector quantization compresses vector embeddings to reduce memory usage and improve search performance. ArcadeDB supports three quantization strategies, each offering different trade-offs between memory consumption, search speed, and accuracy.

Quantization Options

NONE (No Quantization)

Full precision float32 vectors (4 bytes per dimension). This is the default and provides 100% accuracy but highest memory usage.

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE",
  "quantization": "NONE"  -- Default, can be omitted
};
```

INT8 Quantization

8-bit integer quantization (1 byte per dimension). Provides 4x memory reduction with minimal accuracy loss.

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE",
  "quantization": "INT8"  -- Recommended for production
};
```

BINARY Quantization

Binary quantization (1 bit per dimension). Provides 32x memory reduction for extreme compression scenarios.

Each vector is stored as one bit per dimension (at or above the vector's own median, or below it). The graph is searched on the sign pattern, which for COSINE is a Hamming distance, and a search then fetches four times the requested neighbours and reranks them with the similarity function on the vectors the records store, so every BINARY query pays up to `4 * k` record reads and the distances it returns are exact. Candidates whose record is gone or carries no usable vector are dropped, so a query can return fewer than `k` rows. Since 26.10.2 the grouped search (`groupBy`) is

reranked too: it collects four times the groups and four times the members per group, reranks them on the stored vectors, and only then applies `limit`, `groupSize` and `maxDistance`, so its recall and its distances match the ungrouped search. An index created by an earlier version keeps its graph until it is rebuilt (`REBUILD INDEX`), which is recommended to get the full recall.

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE",
  "quantization": "BINARY"  -- Use when memory is severely constrained
};
```

PRODUCT Quantization

Product quantization (PQ) keeps a compact code per vector in memory (the `.vecpq` file). Since 26.11.1 a search of a PRODUCT index through SQL (`vector.neighbors`), openCypher (`db.index.vector.queryNodes`) or the fluent select walks the graph on the PQ codes, keeps `k * rerankOversample` candidates (the `rerankOversample` index setting, default 10; the `rerank` option of `vector.neighbors` per query) and scores them on the stored vectors, so the distances it returns are exact and its recall is close to a NONE index, at the cost of up to that many record reads per query. Before 26.11.1 these searches scored every visited node on the stored vectors and did not use the PQ codes. These three doors share `LSMVectorIndex.findNeighbors(query, k, efSearch, rerankOversample, allowedRIDs, guard)` (a negative `rerankOversample` is the index's setting); a Java caller that wants the same answer calls it, since `findNeighborsFromVector(…)` still walks every quantization but BINARY exactly. A BINARY index walks its one-bit codes and re-ranks `k * rerankOversample` candidates (or the `efSearch` beam when larger) through both methods since 26.11.1; `findNeighbors` also takes a per-query `rerankOversample` for it.

The PQ-approximate Java API, `LSMVectorIndex.findNeighborsFromVectorApproximate(query, k)` and the select `approximate(true)`, returns the PQ scores as they are, with no record read and no re-rank: its recall@10 is about 0.72 on SIFT-1M with 32 subspaces whatever the beam, and the distances are PQ estimates. To re-rank, call `findNeighborsFromVectorApproximate(query, k, efSearch, rerankOversample)` or add `.rerank()` (default 10 candidates per neighbor) or `.rerank(n)` to the select; the distances are then exact.

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE",
  "quantization": "PRODUCT",
  "rerankOversample": 10,   -- 0 = PQ scores alone, no stored vector read
  "pqTrainingLimit": 128000 -- vectors (a random sample) the codebook is trained on
};
```

The graph file does not store the PQ codes of each node's neighbors (jvector FusedPQ) unless `"fusedPQ": true`; it then grows to several times the size of a NONE graph file, and pays off only when the PQ codes cannot stay in memory.

Performance and Accuracy Comparison

The table below compares memory and recall for 100,000 vectors with 384 dimensions using COSINE similarity. Search latency depends on the data and the recall target, so the table gives it relative to NONE only:

Quantization	Memory Usage	Search Latency	Recall@10	Recall@100	Recommended Use Case
NONE (baseline)	1.0x (156 MB)	Baseline	100%	100%	Maximum accuracy required, unlimited memory
INT8	0.25x (39 MB)	Lower than NONE	95-98%	98-99%	**Recommended for most production use cases**
BINARY	0.03x (5 MB)	Lower than NONE	85-92%	90-95%	Memory severely constrained, approximate search acceptable

Memory Usage Calculation for 100K vectors:

- **384 dimensions**: NONE = 156 MB, INT8 = 39 MB (75% reduction), BINARY = 5 MB (97% reduction)
- **1536 dimensions**: NONE = 624 MB, INT8 = 156 MB (75% reduction), BINARY = 19 MB (97% reduction)

Formula: `Memory (MB) = numVectors x dimensions x bytesPerDim / 1024 / 1024`

Where `bytesPerDim` = 4 (NONE), 1 (INT8), or 0.125 (BINARY)

When to Use Each Quantization Type

Use NONE (No Quantization) when:

- Accuracy is critical (e.g., medical imaging, legal document similarity)
- Dataset is small (<10K vectors) where memory is not a concern
- You need exact nearest neighbor results
- Memory budget is unlimited

Use INT8 Quantization when (RECOMMENDED):

- You need to balance memory usage with accuracy
- Dataset is medium to large (10K-10M vectors)
- 2-5% accuracy loss is acceptable
- Production deployments with constrained resources
- You want faster search performance with minimal quality degradation

INT8 quantization is the **recommended default** for most production workloads. It provides: - 4x memory reduction - Usually lower search latency, because vectors are read from compact index pages instead of documents (the gain depends on the data and the recall target) - 95-98% recall (excellent for most applications) - Transparent operation (no query changes needed)

Use BINARY Quantization when:

- Memory is severely constrained (edge devices, mobile, embedded systems)
- Dataset is very large (10M+ vectors)
- You can tolerate 8-15% accuracy loss
- You're implementing two-stage search (binary pre-filter + full precision rerank)
- Approximate results are acceptable

Quantization Best Practices

1. **Start with INT8**: For most use cases, INT8 provides the best balance. Only use NONE if accuracy loss is unacceptable.
2. **Measure Recall on Your Data**: Quantization accuracy depends on your embedding model and data distribution. Always benchmark on representative queries:

```
// Compare recall between NONE and INT8
LSMVectorIndex indexNone = createIndex(quantization=NONE);
LSMVectorIndex indexInt8 = createIndex(quantization=INT8);

double recall = measureRecall(indexNone, indexInt8, testQueries, k=10);
// Expect: recall > 0.95 for INT8
```

1. **Combine with Graph Storage**: For large indexes (100K+ vectors), combine quantization with `storeVectorsInGraph=true`:

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "quantization": "INT8",           -- 4x memory reduction
  "storeVectorsInGraph": true,      -- Fast co-located access
  "locationCacheSize": 200000,
  "mutationsBeforeRebuild": 1000
};
```

This combination provides maximum benefit: - INT8: 4x smaller vectors (624 MB → 156 MB for 100K × 1536-dim) - Graph storage: Eliminates document lookup overhead - Result: ~87% total RAM reduction compared to NONE + document fetching

1. **Monitor Fetch Sources**: Use metrics to verify quantization is working:

```
Map<String, Long> stats = index.getStats();
System.out.println("Fetched from quantized: " + stats.get(
"vectorFetchFromQuantized"));
System.out.println("Fetched from documents: " + stats.get(
"vectorFetchFromDocuments"));
// Should see: vectorFetchFromQuantized > 0, vectorFetchFromDocuments = 0
```

1. **Quantization is Transparent**: Vectors are automatically quantized on insert and dequantized on retrieval. No query changes needed:

```
-- Same query works for all quantization types
SELECT expand(vector.neighbors('Document[embedding]', :queryVector, 10))
```

1. **Two-Stage Search with BINARY**: For very large datasets, use BINARY quantization as a fast pre-filter, then rerank with full precision:

```
// Stage 1: Fast binary search (top 100)
List<RID> candidates = binaryIndex.findNeighborsFromVector(query, 100);

// Stage 2: Rerank with full precision (top 10)
List<Result> finalResults = rerankWithFullPrecision(candidates, query, 10);
```

Expected Performance Improvements

Based on benchmarks with 100,000 vectors (384 dimensions, COSINE similarity):

Search Latency (k=10):

- NONE: 1,234 µs (baseline)
- INT8: 1,050 µs (15% faster)
- BINARY: 980 µs (21% faster)

Memory Usage:

- NONE: 156 MB (baseline)
- INT8: 39 MB (75% reduction)
- BINARY: 5 MB (97% reduction)

Accuracy:

- NONE: 100% recall (baseline)
- INT8: 96.2% recall@10, 98.5% recall@100
- BINARY: 88.4% recall@10, 92.1% recall@100

Speedup Explanation: Quantization improves search speed despite being computationally simpler

operations because:

1. **Better CPU cache utilization**: Smaller vectors fit better in L1/L2 cache
2. **Reduced memory bandwidth**: 4x-32x less data to transfer from RAM
3. **Faster distance calculations**: Integer arithmetic (INT8) and bit operations (BINARY) are faster than floating-point

Configuration Examples by Use Case

Small Dataset (<10K vectors) - No Quantization Needed:

```
CREATE INDEX ON SmallDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE"
  -- quantization defaults to NONE (full precision)
};
```

Medium Dataset (10K-100K vectors) - INT8 Recommended:

```
CREATE INDEX ON MediumDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 768,
  "similarity": "COSINE",
  "quantization": "INT8"  -- 4x memory reduction, minimal accuracy loss
};
```

Large Dataset (100K-1M vectors) - INT8 + Graph Storage:

```
CREATE INDEX ON LargeDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "quantization": "INT8",
  "storeVectorsInGraph": true,     -- Co-locate vectors with graph
  "locationCacheSize": 200000,     -- Cache hot vectors
  "mutationsBeforeRebuild": 1000   -- Reduce rebuild frequency
};
```

Very Large Dataset (1M+ vectors) - BINARY for Pre-filtering:

```
-- Create both indexes: BINARY for fast pre-filter, NONE for reranking
CREATE INDEX ON MassiveDoc (embeddingBinary) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "quantization": "BINARY",         -- 32x compression for fast search
  "storeVectorsInGraph": true
};
```

```
CREATE INDEX ON MassiveDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE",
  "quantization": "NONE"            -- Full precision for reranking
};
```

Then implement two-stage search in your application.

Edge/Mobile Deployment - BINARY + Aggressive Caching:

```
CREATE INDEX ON EdgeDoc (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE",
  "quantization": "BINARY",         -- Extreme compression (32x)
  "maxConnections": 8,              -- Reduce graph size
  "beamWidth": 100,
  "locationCacheSize": 50000,       -- Limit cache
  "graphBuildCacheSize": 5000,      -- Limit build cache
  "storeVectorsInGraph": true       -- Reduce I/O
};
```

Benchmarking Quantization

To measure quantization performance on your specific dataset:

1. **Create test indexes with different quantization types**
2. **Run representative queries**
3. **Compare recall, latency, and memory usage**

Example benchmark:

```
// Create indexes with different quantization
LSMVectorIndex indexNone = createIndex("NONE", 1536);
LSMVectorIndex indexInt8 = createIndex("INT8", 1536);
LSMVectorIndex indexBinary = createIndex("BINARY", 1536);

// Generate test queries
List<float[]> testQueries = generateTestQueries(100);

// Measure recall against NONE baseline
for (float[] query : testQueries) {
  List<RID> groundTruth = indexNone.findNeighborsFromVector(query, 10);
  List<RID> int8Results = indexInt8.findNeighborsFromVector(query, 10);
  List<RID> binaryResults = indexBinary.findNeighborsFromVector(query, 10);

  double int8Recall = calculateRecall(int8Results, groundTruth);
  double binaryRecall = calculateRecall(binaryResults, groundTruth);
```

```
  System.out.printf("Query recall - INT8: %.2f%%, BINARY: %.2f%%\n",
                    int8Recall * 100, binaryRecall * 100);
}

// Measure memory usage
Map<String, Long> statsNone = indexNone.getStats();
Map<String, Long> statsInt8 = indexInt8.getStats();
Map<String, Long> statsBinary = indexBinary.getStats();

System.out.println("Memory estimates:");
System.out.println("  NONE:   " + statsNone.get("estimatedMemoryBytes") / 1024 / 1024
+ " MB");
System.out.println("  INT8:   " + statsInt8.get("estimatedMemoryBytes") / 1024 / 1024
+ " MB");
System.out.println("  BINARY: " + statsBinary.get("estimatedMemoryBytes") / 1024 /
1024 + " MB");
```

For detailed benchmarking methodology, see the JMH benchmark implementation at: `engine/src/test/java/com/arcadedb/index/vector/LSMVectorIndexStorageJMHBenchmark.java`

4.10.5. Querying Vector Indexes with SQL

Use the `vector.neighbors()` function to perform similarity searches:

Basic Similarity Search

Find the 10 most similar documents to a query vector:

```
-- Returns one row per neighbor: the document's properties and its distance
SELECT expand(vector.neighbors('Document[embedding]', :queryVector, 10))
```

The `vector.neighbors()` function takes the **index name** (format: `'TypeName[propertyName]'`) as its first argument and returns a list of maps. Each map contains all document properties (e.g., `name`, `title`), plus `distance` (a distance, not a similarity: smaller means closer, and results come back nearest-first), `@rid`, `@type`, and `record` (the full document object). Use `expand()` to flatten the list into individual result rows so you can access these fields directly (e.g., `SELECT name, distance FROM (SELECT expand(⋯))`). The `distance` semantics depend on the similarity metric (for COSINE: 0 = identical, higher = more different). Use named parameters (e.g., `$queryVector`) instead of hardcoded vectors for real-world applications.

Combining with Other Filters

Perform a vector search and filter results client-side, or use a subquery:

```
-- Find the 10 nearest neighbors
SELECT expand(vector.neighbors('Document[embedding]', :queryVector, 10))
```

After `expand()`, each row contains the document's properties and a `distance` field. You can filter with a WHERE clause in the outer query (e.g., `SELECT ⋯ FROM (SELECT expand(⋯)) WHERE source = 'wiki'`) or in your application code.

Multi-Modal Vector Search

Create and query multiple vector indexes for multi-modal data:

```
-- Create multi-modal schema
CREATE VERTEX TYPE MultiModalRecord;
CREATE PROPERTY MultiModalRecord.imageEmbedding LIST OF FLOAT;
CREATE PROPERTY MultiModalRecord.textEmbedding LIST OF FLOAT;

-- Create separate indexes
CREATE INDEX ON MultiModalRecord (imageEmbedding) LSM_VECTOR METADATA {
  dimensions: 512,
  similarity: 'COSINE'
};

CREATE INDEX ON MultiModalRecord (textEmbedding) LSM_VECTOR METADATA {
  dimensions: 768,
  similarity: 'COSINE'
};

-- Query by image modality
SELECT expand(vector.neighbors('MultiModalRecord[imageEmbedding]', [0.1, 0.2, ...],
5))

-- Query by text modality
SELECT expand(vector.neighbors('MultiModalRecord[textEmbedding]', [0.2, 0.3, ...], 5))
```

Retrieving Neighbor Details

Get detailed information about nearest neighbors:

```
-- Returns array of objects with distance and keys
SELECT vector.neighbors('Document[embedding]', :queryVector, 10) AS neighbors;
```

Note: The first argument to `vector.neighbors()` must always be the **index name** in `'TypeName[propertyName]'` format (e.g., `'Document[embedding]'`). Without `expand()`, you get a single row with the `neighbors` field containing the entire list. Use `expand()` to flatten the results into individual rows where you can access document properties and `distance` directly.

Type-Specific Search with Inheritance

When a vector index is created on a parent type, ArcadeDB automatically creates sub-indexes for each bucket, including buckets of child types. This enables powerful filtering capabilities:

Use Case: Multi-Modal Embeddings

Imagine a system storing embeddings for different content types (images, documents, audio) that all extend a common parent type:

```
-- Create parent type with vector index
CREATE VERTEX TYPE Embedding;
CREATE PROPERTY Embedding.vector ARRAY_OF_FLOATS;
CREATE INDEX ON Embedding (vector) LSM_VECTOR METADATA {
  dimensions: 768,
  similarity: 'COSINE'
};

-- Create specialized child types
CREATE VERTEX TYPE ImageEmbedding EXTENDS Embedding;
CREATE VERTEX TYPE DocumentEmbedding EXTENDS Embedding;
CREATE VERTEX TYPE AudioEmbedding EXTENDS Embedding;
```

Class-Specific Search: Find the top 50 closest IMAGE embeddings only:

```
SELECT expand(vector.neighbors('ImageEmbedding[vector]', :queryVector, 50))
```

All results will be of type `ImageEmbedding` - records from `DocumentEmbedding` and `AudioEmbedding` are excluded.

Cross-Class Search: Find the top 50 closest embeddings across ALL types:

```
SELECT expand(vector.neighbors('Embedding[vector]', :queryVector, 50))
```

Results may include records from any child type (`ImageEmbedding`, `DocumentEmbedding`, `AudioEmbedding`) based on similarity.

This approach is efficient because:

- Only one index is created and maintained (on the parent type)
- Child type searches use the same underlying index, filtering by bucket
- No need to create separate indexes for each child type

4.10.6. Using the Java API

For programmatic control and embedded applications, use the Java API to create and manage vector indexes.

Creating LSMVectorIndex Programmatically

```
import com.arcadedb.database.Database;
```

```
import com.arcadedb.schema.Schema;
import com.arcadedb.schema.Type;
import com.arcadedb.index.lsm.LSMVectorIndex;
import com.arcadedb.index.lsm.LSMVectorIndexBuilder;
import com.arcadedb.index.vector.VectorSimilarityFunction;

// Get or create schema
final Schema schema = database.getSchema();
if (!schema.existsType("Document")) {
  schema.createVertexType("Document");
}

// Create vector property
if (!schema.existsProperty("Document", "embedding")) {
  schema.createProperty("Document", "embedding", Type.ARRAY_OF_FLOATS);
}

// Create LSMVectorIndex with builder pattern
final LSMVectorIndexBuilder builder = new LSMVectorIndexBuilder(
    database,
    "Document",
    new String[]{"embedding"})
    .withDimensions(768)
    .withSimilarity(VectorSimilarityFunction.COSINE)
    .withMaxConnections(16)
    .withBeamWidth(100)
    .withIdProperty("id")
    .withAddHierarchy(false)          // Enable for large/complex datasets (100K+)
    .withStoreVectorsInGraph(false);  // Enable for large indexes (100K+)

final LSMVectorIndex index = builder.create();
```

For pre-quantized ingest from an int8 embedding provider, declare the property as `BINARY` and call `withEncoding(VectorEncoding.INT8)` (or the string overload). The factory rejects mismatches between property type and encoding at create time:

```
import com.arcadedb.index.vector.VectorEncoding;

// Property MUST be BINARY when encoding=INT8
schema.createProperty("Document", "embedding", Type.BINARY);

final LSMVectorIndexBuilder builder = new LSMVectorIndexBuilder(
    database,
    "Document",
    new String[]{"embedding"})
    .withDimensions(1024)
    .withSimilarity(VectorSimilarityFunction.COSINE)
    .withEncoding(VectorEncoding.INT8);  // wire/storage encoding, not internal
quantization
```

```
final LSMVectorIndex index = builder.create();
```

See INT8 Pre-Quantized Ingest for the trade-offs and when to choose this over `quantization=INT8`.

Adding Vectors to the Index

```
// Get existing index
final LSMVectorIndex index = (LSMVectorIndex) database.getSchema()
    .getIndexByName("Document[embedding]");

// Add vectors with callback
index.addAll(embeddings, (vertex, item, total) -> {
  // Optional callback for handling vertex creation
  // Useful for cascading operations like creating relationships
  if (vertex != null) {
    // Create relationships, update metadata, etc.
    System.out.println("Indexed vertex " + vertex.getIdentity() +
                       " (" + item + "/" + total + ")");
  }
});
```

The callback in the `addAll()` method is useful when you have a graph structure connected to the indexed vertices. For example, when indexing a book, you might calculate embeddings per statement and then create relationships between statements and paragraphs.

Configuring from JSON Metadata

Load configuration from JSON metadata:

```
import com.arcadedb.utility.JSONObject;

final JSONObject metadata = new JSONObject()
    .put("dimensions", 768)
    .put("similarity", "COSINE")
    .put("encoding", "FLOAT32")         // Optional: FLOAT32 (default) or INT8 for
pre-quantized ingest
    .put("maxConnections", 16)
    .put("beamWidth", 100)
    .put("idPropertyName", "id")
    .put("addHierarchy", false)         // Enable for large/complex datasets (100K+)
    .put("storeVectorsInGraph", false)  // Enable for large indexes (100K+)
    .put("locationCacheSize", 100000)   // Optional: limit cache size
    .put("graphBuildCacheSize", 10000)  // Optional: limit build cache
    .put("mutationsBeforeRebuild", 100); // Optional: rebuild frequency

final LSMVectorIndexBuilder builder = new LSMVectorIndexBuilder(
    database,
    "Document",
    new String[]{"embedding"})
```

```
    .fromMetadata(metadata);

final LSMVectorIndex index = builder.create();
```

Querying Vectors from Java

```
import com.arcadedb.query.sql.executor.ResultSet;

// Define an example query vector. Replace with your actual vector.
float[] queryVector = new float[] {0.1f, 0.2f, 0.3f}; // Vector must match index
dimensions

// Perform similarity search using SQL
final ResultSet resultSet = database.query("sql",
    "SELECT expand(vector.neighbors('Document[embedding]', ?, 10))",
    queryVector);

while (resultSet.hasNext()) {
  final Result result = resultSet.next();
  // Each result has .record (the document) and .distance
  System.out.println("Similar document: " + result.toJSON());
}
```

Transaction Support

LSMVectorIndex fully supports transactions with automatic persistence:

```
database.transaction(() -> {
  // Create vertices with embeddings
  final MutableVertex vertex = database.newVertex("Document");
  vertex.set("content", "Sample text");

  // In a real application, this vector would be generated by an embedding model
  // and should match the dimensions of the index.
  float[] embeddingVector = new float[] {0.5f, 0.6f, 0.7f};
  vertex.set("embedding", embeddingVector);
  vertex.save();

  // Index is automatically updated on transaction commit
});
```

4.10.7. Migration from HnswVectorIndexRAM

If you're migrating from the older HnswVectorIndexRAM approach, note the following differences:

Feature	HnswVectorIndexRAM	LSMVectorIndex
Persistence	Requires explicit conversion	Automatic with transactions

Feature	HnswVectorIndexRAM	LSMVectorIndex
SQL Support	Limited	Full SQL CREATE INDEX and queries
Transaction Support	Manual management	Automatic ACID compliance
Compaction	Manual	Automatic
Recommended For	Bulk loading scenarios	All production use cases

Legacy Bulk Loading Pattern

For backwards compatibility, the HnswVectorIndexRAM bulk loading pattern is still available for initial large-scale imports:

```
import com.arcadedb.index.vector.HnswVectorIndexRAM;
import com.arcadedb.index.vector.VectorSimilarityFunction;
import com.arcadedb.index.vector.Item;
import com.arcadedb.schema.Type;
import java.util.Collection;
import java.util.List;

// For bulk loading only - use LSMVectorIndex for production
String indexName = "Document[embedding]";
if (!database.getSchema().existsIndex(indexName)) {
  // Define parameters for the index
  int dimensions = 768;
  VectorSimilarityFunction distanceFunction = VectorSimilarityFunction.COSINE;
  int m = 16;
  int ef = 200;
  int efConstruction = 200;
  // Example embeddings. In a real scenario, this would be a large collection.
  Collection<Item<Object, float[]>> embeddings = List.of(
      new Item<>("doc1", new float[]{0.1f, 0.2f}),
      new Item<>("doc2", new float[]{0.3f, 0.4f})
  );

  // Step 1: Load into RAM-based index
  final HnswVectorIndexRAM<Object, float[], Item<Object, float[]>, Float> hnswIndex =
      HnswVectorIndexRAM.newBuilder(dimensions, distanceFunction, 100_000)
          .withM(m)
          .withEf(ef)
          .withEfConstruction(efConstruction)
          .build();

  hnswIndex.addAll(embeddings,
                   Runtime.getRuntime().availableProcessors(), null);

  // Step 2: Create persistent index
  hnswIndex.createPersistentIndex(database)
      .withVertexType("Document")
```

```
        .withEdgeType("Proximity")
        .withVectorProperty("embedding", Type.ARRAY_OF_FLOATS)
        .withIdProperty("id")
        .create();
}
```

For most use cases, directly creating an LSMVectorIndex via SQL or the Java API is simpler and provides better transaction support. The HnswVectorIndexRAM approach is only recommended for specific bulk loading scenarios with millions of vectors.

4.10.8. Best Practices

1. **Choose the Right Similarity Function**: Use COSINE for normalized embeddings (most common), DOT_PRODUCT for performance with normalized vectors, or EUCLIDEAN for spatial data.
2. **Normalize Your Vectors**: For COSINE and DOT_PRODUCT similarity, ensure vectors are normalized to unit length for best results.
3. **Tune Performance Parameters**: Start with defaults (maxConnections=32, beamWidth=100) and increase for better recall if needed.
4. **Use Transactions**: Always insert vectors within transactions for data consistency and automatic index updates.
5. **Batch Insertions**: For large datasets, batch your insertions in reasonably-sized transactions (1000-10000 records) for optimal performance.
6. **Monitor Index Size**: Vector indexes can be large. Monitor disk usage and consider the dimensions parameter carefully.
7. **SQL for Simplicity**: Prefer SQL for index creation and queries unless you need programmatic control.

4.10.9. Common Use Cases

Semantic Document Search

```
-- Create index for document embeddings
CREATE VERTEX TYPE Document;
CREATE PROPERTY Document.content STRING;
CREATE PROPERTY Document.embedding LIST OF FLOAT;
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  "dimensions": 384,
  "similarity": "COSINE"
};

-- Query for the 10 nearest documents
SELECT expand(vector.neighbors('Document[embedding]', :queryEmbedding, 10))
```

Image Similarity Search

```
-- Create index for image feature vectors
CREATE VERTEX TYPE Image;
CREATE PROPERTY Image.url STRING;
CREATE PROPERTY Image.features LIST OF FLOAT;
CREATE INDEX ON Image (features) LSM_VECTOR METADATA {
  dimensions: 512,
  similarity: 'COSINE'
};

-- Find the 5 most similar images
SELECT expand(vector.neighbors('Image[features]', :imageFeatures, 5))
```

Recommendation System

```
-- Create index for user/item embeddings
CREATE VERTEX TYPE Product;
CREATE PROPERTY Product.embedding LIST OF FLOAT;
CREATE INDEX ON Product (embedding) LSM_VECTOR METADATA {
  "dimensions": 128,
  "similarity": "COSINE"
};

-- Find the 20 most similar products for recommendations
SELECT expand(vector.neighbors('Product[embedding]', :userPreferenceVector, 20))
```

For more information, see:

- Indexes
- CREATE INDEX
- vector.neighbors()

4.11. Java API (Remote)

ArcadeDB comes with a convenient Java library to work with a remote database.

4.11.1. RemoteServer Class

The RemoteServer class is used to manage remote server connections and databases.

The server administration calls (databases, users, groups and API tokens) prefer the cluster leader and wait out an election in progress. If the server becomes unreachable, they switch to another server of the cluster only when `arcadedb.network.sameServerErrorRetries` is greater than 0 (the default is 0). A write whose answer was lost is never sent again, because the server may already have applied it. `createApiToken` refuses to send the token over plain HTTP to a non-loopback host, whichever server the request is sent to, unless `setAllowInsecureApiTokenTransport(true)` is set.

When a server becomes unreachable, `RemoteServer` and `RemoteDatabase` refresh the cluster configuration by asking every known replica in turn, then the old leader, then the address the client was created with, and the retry goes to the server that answered. If no server can give the configuration, for example during an election, the client keeps the leader and the replica list it had and spends the remaining retries (`arcadedb.ha.errorRetries`) on that list, so a read can still reach a follower. The error is raised when those retries run out. *(Since v26.12.1.)*

(Since v26.10.2.) `acceptDivergedDatabase(databaseName)` and `acceptStaleSnapshot()` are the operator overrides for a node of a replicated cluster that has no peer left to resync from. Both are root only and accept the node's local copy as the truth without a resync, so the entries that were skipped are not replayed. `acceptDivergedDatabase` lifts the quarantine, and its read floor, standing on one database, through `POST /api/v1/cluster/accept-diverged/{database}`. It is allowed on a node that is the only voter of its cluster, or when every voter holds the same database quarantined, since no node then has a copy to resync from. It returns `result`, `database`, `localServer` and `appliedIndex`, plus `divergenceCause` and `readFloor` when they stood. `acceptStaleSnapshot` lifts the node-wide stale-snapshot read floor, through `POST /api/v1/cluster/accept-stale-snapshot`, on a sole voter only, and returns `result`, `localServer`, `readFloor`, `snapshotIndex` and `appliedIndex`. The server answers `404` when nothing stands to be lifted (for `acceptDivergedDatabase`, when the database carries no quarantine and no read floor) and `409` where a resync from a peer is still the way out; `acceptStaleSnapshot` also answers `409` while a snapshot download is running. The gRPC client `RemoteGrpcServer` has the same two methods, returning the `AcceptDivergedDatabaseResponse` and `AcceptStaleSnapshotResponse` messages and raising its not-found and failed-precondition errors for the same cases (see gRPC API). When to use them is described in Lifting a quarantine without a resync and Lifting the stale-snapshot read floor.

(Since v26.10.2.) A create refused because the name is already taken raises `com.arcadedb.remote.RemoteAlreadyExistsException`: a user (`createUser`), an API token (`createApiToken`) or a database (`create`). The server answers these with `409 Conflict`, and the gRPC server with `ALREADY_EXISTS`; `RemoteGrpcServer` raises the same exception from `createUser`, `createApiToken` and `createDatabase`, where it used to report a duplicated key (or, for `createDatabase`, a plain `RuntimeException`). It extends `RemoteException`, so code that caught `RemoteException` for these cases keeps working, and a provisioning tool can catch it to treat "already there" as success. Restoring a backup into a database name that is taken still answers `400`.

(Since v26.11.1.) A failing statement raises the same exception remotely as embedded: a syntax error is a `CommandSQLParsingException`, an unknown function or a missing bucket a `CommandExecutionException`, an `INSERT` sent through `query()` a `QueryNotIdempotentException`, an unknown query language an `IllegalArgumentException`, and so on for every ArcadeDB exception that carries only its message, over HTTP and gRPC alike. Its message is the server's own sentence. A missing query parameter arrives as the `CommandSemanticException` that `CommandParameterMissingException` extends, since the latter is built from the parameter names rather than from a message. `RemoteException` itself now extends `ArcadeDBException`, so a `catch (ArcadeDBException e)` written against the database interfaces catches the same failures whether the database is embedded or remote, including a failure whose type the client cannot rebuild (a class of a newer server, or of a plugin). In 26.10.2 and earlier these failures reached the caller as a `RemoteException` that was not an `ArcadeDBException`, with a message of the form `Error on executing remote operation … (cause:<class> detail:<message>)`.

(Since v26.11.1.) `getType()` answers the record's type on a mutable remote record too (`newDocument()`,

`newVertex()`, `modify()`), as it does on a record read from the server, instead of throwing `UnsupportedOperationException`. `toMap()` and `toJSON()` of a remote edge include `@in` and `@out`, as an embedded edge does.

(Since v26.12.1, issue #12190.) The record factories of `RemoteDatabase` (and of `RemoteGrpcDatabase`, which inherits them) follow the embedded API. `newEmbeddedDocument()` works on a remote document, vertex or edge, also nested, in a list or in a map: the embedded document is built on the client and sent with its owner on `save()`. Only a document type can be embedded. `newDocument()` on a vertex or edge type, and `newVertex()` on a type that is not a vertex type, throw `IllegalArgumentException`. `newRecord()` of a remote type builds a document on a document type and a vertex on a vertex type, and throws `UnsupportedOperationException` on an edge type: create an edge from its source vertex with `newEdge()`. Before, `newEmbeddedDocument()` threw a `NullPointerException`, `newDocument()` on a vertex type saved a vertex, `newVertex()` on a document type threw a `ClassCastException` and `newRecord()` on an edge type was accepted. Code that created vertices with `newDocument()` has to use `newVertex()`.

4.11.2. RemoteDatabase Class

The RemoteDatabase class is used to work with a remote database instance.

Add the following dependencies in your Maven pom.xml file under the tag `<dependencies>`:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-engine</artifactId>
    <version>26.11.1</version>
</dependency>
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-network</artifactId>
    <version>26.11.1</version>
</dependency>
```

4.11.3. 10-Minute Tutorial

Connect to a remote database

There are 2 main classes that handle the connection to the remote database by translating the Java API into http-calls:

- `RemoteServer`, to create new database on a remote server
- `RemoteDatabase`, to work with a remote database instance

We start by creating a RemoteServer:

```
RemoteServer server = new RemoteServer("localhost", 2480, "root", "playwithdata");
```

Next, let's create a RemoteDatabase instance pointing to the database "mydb" on ArcadeDB Server located on "localhost", port 2480 — the HTTP port, which is what `RemoteDatabase` speaks. User and passwords are respectively "root" and "playwithdata":

```
RemoteDatabase database = new RemoteDatabase("localhost", 2480, "mydb", "root",
"playwithdata");
```

The `RemoteDatabase` is not thread-safe. It holds a backend session. If a thread closes a transaction, it also closes the session, so that any request performed by a concurrent thread using the same RemoteDatabase instance leads to unexpected behavior. To avoid concurrency problems you should not hold RemoteDatabase instances as fields on the heap, but rather create a new instance in method scope as a variable on the stack.

4.11.4. RemoteImmutableVertex and RemoteMutableVertex

Vertices retrieved through `RemoteDatabase` are instances of `RemoteImmutableVertex` (or `RemoteMutableVertex` after calling `modify()`). These classes expose graph-traversal methods that translate directly into SQL queries over HTTP — there is no per-vertex round trip.

Traversal methods

Method	Description
`getVertices(direction, edgeTypes…)`	Returns all neighbor vertices, transparently paginating in batches of `DEFAULT_PAGE_SIZE` (2,000) records per HTTP call. The vertices come in the order of their edges, once per edge, and a vertex deleted since its edge was read is left out.
`getVerticesPaged(direction, pageSize, edgeTypes…)`	Same as above but with an explicit page size. Use to tune round-trip vs. payload trade-off for supernodes. A page size of zero or less throws `IllegalArgumentException`.
`getVertices(direction, limit, skip, edgeTypes…)`	Returns a single page of up to `limit` neighbor vertices starting at offset `skip`. Useful for implementing your own pagination.
`getEdges(direction, edgeTypes…)`	Returns all connected edges in RID order, transparently paginating with `DEFAULT_PAGE_SIZE` per HTTP call. `BOTH` returns the outgoing edges, then the incoming ones, so a self-loop comes twice, as in the embedded API.
`getEdgesPaged(direction, pageSize, edgeTypes…)`	Same as above but with an explicit page size.
`getEdges(direction, limit, skip, edgeTypes…)`	Returns a single page of up to `limit` edges starting at offset `skip`.
`countEdges(direction, edgeTypes…)`	Returns the number of connected edges (single HTTP call, no result materialization).

`DEFAULT_PAGE_SIZE = 2_000` is the batch size used by `getVertices()` / `getEdges()`. Values up to 20,000

are safe against the server's default result limit.

Example

```
RemoteDatabase db = new RemoteDatabase("localhost", 2480, "mydb", "root",
"playwithdata");

// standard traversal -- paginates automatically, no cast needed
Vertex hub = db.lookupByRID(new RID(db, "#1:0"), true);
for (Vertex neighbor : hub.getVertices(Vertex.DIRECTION.OUT, "Knows")) {
    System.out.println(neighbor.getString("name"));
}

// supernode -- explicit page size requires cast to the concrete remote type
RemoteImmutableVertex hubRemote = (RemoteImmutableVertex) hub;
for (Vertex neighbor : hubRemote.getVerticesPaged(Vertex.DIRECTION.OUT, 5_000,
"Knows")) {
    System.out.println(neighbor.getString("name"));
}
```

Each page of `getEdges()`, `getVertices()`, `getEdgesPaged()` and `getVerticesPaged()` resumes after the RID of the last edge of the previous page, so deleting the edges or the vertices while iterating (`for (Edge e : hub.getEdges(OUT)) e.delete();`), or another client deleting edges already read, does not make the iteration skip live ones. A lightweight edge has no RID of its own: deleting, while iterating, an already read lightweight edge of the same type between the same two vertices as a live one can still hide the live one. The server still reads the whole edge list of the vertex on every page, so for supernodes with millions of edges prefer a larger page size, or a SQL query with a server-side `WHERE` filter to narrow results before fetching them.

The database, which name you have given as the third parameter in the constructor is not automatically created nor is it checked whether it exists. Before you use the `RemoteDatabase` make sure that the database exists, or, if it doesn't, create it:

```
if (!server.exists("mydb"))
    server.create();
```

For a list of all databases hosted by the server use:

```
List<String> databases = server.databases();
assert(databases.contains("mydb"));
```

Now that we have the database created we may create the schema (but we may also work without):

```
String schema =
"""
```

```
    create vertex type Customer if not exists;
    create property Customer.name if not exists string;
    create property Customer.surname if not exists string;
    create index if not exists on Customer (name, surname) unique;
""";
database.command("sqlscript", schema);
```

As language we have used `sqlscript`, which has the advantage that all the statements in the script are executed within one single transaction. Of course you can also send these requests line by line with `sql` as language. If you go line by line it is important to enclose the requests in a common transaction:

```
database.begin();
database.command("sql", "create vertex type Customer if not exists");
database.command("sql", "create property Customer.name if not exists string");
database.command("sql", "create property Customer.surname if not exists string");
database.command("sql", "create index if not exists on Customer (name, surname)
unique");
database.commit();
```

Or:

```
database.transaction(() -> {
    ...
});
```

The `transaction` method encloses the lambda function with a `begin` - `commit`, or a `rollback` in case of any exception.

If the transaction fails because another client modified the same records, it is rolled back and the lambda is executed again, up to `arcadedb.txRetries` times (3 by default). There is one exception: when `transaction(lambda, true)` is called from inside a transaction you opened yourself, the error is thrown to you instead of being retried, because re-executing the lambda inside your still open transaction would leave the effects of the failed attempt behind. In that case catch the exception and retry the whole unit of work.

The same happens when the server refuses the transaction before running it: the `503` a node answers while it installs a snapshot, a leader change, or a `begin` whose connection could not be established. Each retry waits first, with the exponential backoff the embedded `transaction()` uses (`arcadedb.txRetryDelayBase` doubling up to `arcadedb.txRetryDelay`), and when the refusal carries a `Retry-After` header the retry waits at least that long, up to `arcadedb.network.retryAfterMaxWait` (30 seconds by default). A duplicated key is retried only once, since failing twice means the duplicate is genuine. A commit the server answered with an error also releases the server-side session right away instead of leaving it to the session timeout.

A commit whose connection drops after the request was sent is different: the server may already have applied it. Over gRPC such a commit (status `UNAVAILABLE` with no error from the server) raises a

`TransactionException` and is **not** retried, so the work is never applied twice. Read the data back to find out whether the commit landed.

(Since v26.10.2.) The gRPC client draws the same line for the other writes whose answer can be lost. A bare `UNAVAILABLE` (no error class from the server, and not a failure to connect) on an admin write of `RemoteGrpcServer` (creating or dropping a database, a user or a token, and the other server writes) or on a streaming write of `RemoteGrpcDatabase` (`ingestStream`, `ingestBidi`, `timeSeriesWriteStream`, graph batch loads) raises a non-retryable `RemoteException` whose message says the server may already have applied the request. A streaming write commits as it goes, so any part of it, from none to all, may be durable. Reads, and `validate_only` dry runs, stay retryable. A client-side deadline that expires before the server answers raises a `TimeoutException` whose message says that the write may already have been applied. Before, these failures were reported as retryable, so a caller that retried could apply a write twice.

A statement stopped on the server (by the `terminate query` server command, Cypher `TERMINATE TRANSACTIONS` or Studio) raises `com.arcadedb.exception.QueryTerminatedException`, over HTTP and over gRPC alike. It is not retried by `transaction()`: somebody asked for the work to stop. *(Since v26.10.2.)*

(Since v26.12.1, issue #12193.) The index methods of the schema API answer the index over a `RemoteDatabase` as they do embedded. `createTypeIndex()` and `getOrCreateTypeIndex()`, of the `Schema` and of a type, return the index they created or found, and `getIndexByProperties()`, `getPolymorphicIndexByProperties()`, `getIndexesByProperties()` and `getAllIndexes()` of a type find an existing index, so a setup such as `if (type.getIndexByProperties("email") == null) type.createTypeIndex(…)` creates the index once. The returned index describes it (name, type, properties, index type, unique, null strategy, key types) and `drop()` drops it. The client holds none of its entries, so `get()`, `range()`, `countEntries()`, `put()`, `setNullStrategy()` and the other methods reading or changing them throw `UnsupportedOperationException`: run a query instead, the server picks the index, and set the null strategy when creating the index (`CREATE INDEX … NULL_STRATEGY`). Each lookup asks the server and is not cached, so it sees an index another client created or dropped. Before, the lookups answered `null` or an empty list for an existing index, `getIndexesByProperties(String…)` and `getAllIndexes()` threw `UnsupportedOperationException`, and the creation methods returned `null`.

Now that the schema is in place, we will insert some data:

```
database.command("sql", "insert into Customer(name, surname) values(?, ?)", "Jay",
"Miner");
database.command("sql",
        "update Customer set lastcall = :lastcall where name = :name and surname =
:surname",
        Map.of("lastcall", Date.from(Instant.now()), "name", "Jay", "surname",
"Miner"));
```

The first command creates a new vertex of type 'Customer'. The second command updates this vertex by adding a new value for the property 'lastcall'. The first command uses unnamed parameters where the values are mapped by their order in the parameter list. Whereas the second command uses named parameters, which are mapped by the parameter names.

(Since v26.12.1.) A parameter, and a value set on a record that `save()` sends, keeps its Java type as the embedded API does, also for a property the schema does not declare: a `RID` (or a saved record) is stored as a link, a `BigDecimal` as a DECIMAL with its scale (`SELECT :a + :b` with `0.1` and `0.2` answers `0.3`), a `byte[]` as BINARY, a `LocalDate` as a DATE, `java.util.Date`, `LocalDateTime` and `Instant` as a DATETIME and a `ZonedDateTime` as a ZONED_DATETIME. The client sends them as typed parameter markers, which a server older than 26.12.1 does not decode, so upgrade the server first. Before, these values reached the server as plain JSON: a `RID` was stored as a string, a record as an embedded copy, a `BigDecimal` as a double, a `byte[]` as a list and the dates as numbers. A record read back from the server holds an undeclared link as a `RID`, and a list or map of links as a list or map of `RID`s, as a projection of the same property does (since v26.12.1, issue #12195; before, the record answered the rid text `"#1:0"). The `RID` is not bound to the remote database, so load the record it points to with `database.lookupByRID(rid, true)` (issue #12603). Links in a property declared as a bare `LIST` or `MAP`, without `OF LINK`, still read back as strings (issue #12598), and an undeclared date, decimal or binary property still reads back as a string, a double or a list, so `modify()` and `save()` write such an untouched property back with that type (issue #12599).

We can now query some customer data from our database:

```
ResultSet resultSet = database.query("sql", "select from Customer where name = :name",
        Map.of("name", "Jay"));
Map<String, Object> customer = resultSet.stream().findFirst().map(Result::toMap
).orElse(Map.of());
```

The query returns a `ResultSet`, which holds an array of objects of type `Result`, which can be read as Map:

```
{name=Jay, @cat=d, @rid=#1:0, lastcall=1675961747290, surname=Miner, @type=Customer}
```

Streaming results

`query()` and `command()` wait until the server has serialized the whole result, and keep it in memory. For a large result use `queryStream()` or `commandStream()` instead: they ask the server for the NDJSON encoding and return a `ResultSet` that reads each row off the connection as it arrives, so neither the server nor the client holds more than one row at a time.

```
try (ResultSet resultSet = database.queryStream("sql", "select from Customer where
surname = ?", "Miner")) {
  while (resultSet.hasNext()) {
    Result customer = resultSet.next();
    ...
  }
}
```

- The `ResultSet` keeps an HTTP connection open until it is read to the end or closed, so open it in a try-with-resources. It can be read only once.
- `hasNext()` returns `false` only after the server sent the `stats` trailer. A failure on the server raises

the exception the buffered call would have raised, and a stream that ends without its trailer (a dropped connection) raises a `RemoteException`: a short result is never returned silently.

- Without `setMaxResultRows()` the stream returns every row, up to the server's hard ceiling `arcadedb.server.httpQueryMaxResultRows` (1,000,000 rows by default). A result above the ceiling raises an exception after the rows that fit, instead of ending short. The server's default cap for buffered responses (`arcadedb.server.httpQueryDefaultLimit`, 20,000 rows) does not apply to a stream *(since v26.12.1, issue #12185; before, the stream stopped at that cap)*. Call `setMaxResultRows()` to cap the stream yourself: when the cap drops rows, the client logs a warning and `hasNext()` returns `false`.
- `commandStream()` takes read-only statements only. The server refuses a statement that writes, because its rows would reach the client before it committed. Use `command()` for writes.
- The response has no `explain` or write statistics. Use `query()` or `command()` when you need them.

`iterateType()` and `iterateBucket()` return every record of the type or bucket, as on the embedded API, whatever `setMaxResultRows()` is set to *(since v26.12.1, issue #12185; before, they stopped at the server's default cap of 20,000 rows while `countType()` reported the full count)*. They read the records in one buffered response, so a type or bucket larger than `arcadedb.server.httpQueryMaxResultRows` is refused with an exception rather than returned in part.

4.12. Java API (gRPC)

ArcadeDB provides a high-performance gRPC-based Java library for remote database communication. gRPC (gRPC Remote Procedure Call) is a modern, open-source, high-performance RPC framework that can efficiently connect services in and across data centers.

4.12.1. Why Use gRPC for ArcadeDB?

gRPC offers several advantages over traditional HTTP-based communication:

- **High Performance**: gRPC uses HTTP/2 for transport, providing lower latency and higher throughput compared to traditional HTTP/1.1
- **Efficient Binary Protocol**: Uses Protocol Buffers for serialization, resulting in smaller message sizes and faster parsing
- **Streaming Support**: Built-in support for bidirectional streaming, ideal for real-time applications
- **Language Agnostic**: While this guide focuses on Java, gRPC supports multiple programming languages with consistent APIs
- **Low Latency**: Optimized for scenarios requiring minimal latency and maximum throughput
- **Connection Multiplexing**: Multiple requests can be sent over a single TCP connection

gRPC is particularly well-suited for:

- Real-time applications requiring fast data access
- Microservices architectures
- High-throughput data processing

- Mobile applications with limited bandwidth
- Services requiring bidirectional streaming

4.12.2. RemoteGrpcServer Class

The `RemoteGrpcServer` class manages connections to the gRPC server and provides server-level operations such as creating databases, listing available databases, and managing server resources.

Key features:

- Establishes and maintains the connection to the ArcadeDB gRPC server
- Handles authentication and security
- Provides methods for database lifecycle management
- Supports connection pooling and resource management

4.12.3. RemoteGrpcDatabase Class

The `RemoteGrpcDatabase` class is used to work with a specific remote database instance via gRPC. It provides methods for executing queries, commands, and transactions with optimal performance characteristics.

Key features:

- Execute SQL queries and commands
- Support for parameterized queries (both named and positional)
- Transaction management (begin, commit, rollback)
- Schema manipulation
- Data manipulation (insert, update, delete)
- High-performance data retrieval

4.12.4. Maven Dependencies

Add the following dependency in your Maven pom.xml file under the tag `<dependencies>`:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-grpc-client</artifactId>
    <version>26.11.1</version>
</dependency>
```

The gRPC client dependency includes all necessary Protocol Buffer definitions and gRPC runtime libraries. No additional dependencies are required.

4.12.5. 10-Minute Tutorial

This tutorial will guide you through the essential operations using ArcadeDB's gRPC API, from establishing a connection to executing various types of database operations.

Connect to a gRPC Server

To begin working with ArcadeDB via gRPC, you need to establish a connection to the server. The gRPC API provides two main classes:

- `RemoteGrpcServer`, to manage server connections and create databases
- `RemoteGrpcDatabase`, to work with a specific remote database instance

The gRPC server listens on port 50051 by default (different from the HTTP port 2480).

Create a RemoteGrpcServer Instance

First, create a `RemoteGrpcServer` instance to connect to your ArcadeDB server:

```
RemoteGrpcServer server = new RemoteGrpcServer(
    "localhost",           // Server hostname
    50051,                 // gRPC port (default: 50051)
    "root",                // Username
    "password",            // Password
    false,                 // Use TLS/SSL (false for development)
    List.of()              // Additional channel credentials (if needed)
);
```

The constructor parameters are:

- `hostname`: The server hostname or IP address
- `port`: The gRPC port (typically 50051)
- `username`: Database user credentials
- `password`: User password
- `useTls`: Whether to use TLS/SSL encryption
- `channelCredentials`: Optional additional credentials for advanced security configurations

Before creating or accessing a database, you can check if it exists:

```
if (!server.exists("mydb")) {
    server.create("mydb");
    System.out.println("Database 'mydb' created successfully");
}
```

You can also list all available databases on the server:

```
List<String> databases = server.databases();
System.out.println("Available databases: " + databases);
```

Create a RemoteGrpcDatabase Instance

Once the server connection is established and the database exists, create a `RemoteGrpcDatabase` instance to interact with a specific database:

```
RemoteGrpcDatabase database = new RemoteGrpcDatabase(
    server,                 // RemoteGrpcServer instance
    "localhost",            // Server hostname
    50051,                  // gRPC port
    2480,                   // HTTP port (for fallback operations)
    "mydb",                 // Database name
    "root",                 // Username
    "password"              // Password
);
```

Like `RemoteDatabase`, the `RemoteGrpcDatabase` class is not thread-safe. Each thread should create its own instance. Avoid sharing `RemoteGrpcDatabase` instances across threads to prevent unexpected behavior.

Best practice for thread safety:

```
// Create database instance in method scope, not as a field
public void performDatabaseOperation() {
    try (RemoteGrpcDatabase database = new RemoteGrpcDatabase(
        server, "localhost", 50051, 2480, "mydb", "root", "password"
    )) {
        // Perform operations
        ResultSet results = database.query("sql", "SELECT * FROM MyType");
        // Process results...
    }
}
```

Create Schema with SQL Commands

With the database connection established, you can create your schema. The gRPC API supports both single SQL statements and SQL scripts:

```
// Using sqlscript for multiple statements in one transaction
String schema = """
    CREATE VERTEX TYPE Article IF NOT EXISTS;
    CREATE PROPERTY Article.id IF NOT EXISTS INTEGER;
    CREATE PROPERTY Article.title IF NOT EXISTS STRING;
    CREATE PROPERTY Article.content IF NOT EXISTS STRING;
```

```
    CREATE PROPERTY Article.published IF NOT EXISTS DATETIME;
    CREATE PROPERTY Article.author IF NOT EXISTS STRING;
    CREATE INDEX IF NOT EXISTS ON Article(id) UNIQUE;
    """;

database.command("sqlscript", schema);
System.out.println("Schema created successfully");
```

The `sqlscript` language is recommended for schema creation because:

- All statements execute within a single transaction
- Improved performance due to reduced network round-trips
- Atomic operations - either all succeed or all fail

Alternatively, you can execute statements individually:

```
database.begin();
try {
    database.command("sql", "CREATE VERTEX TYPE Article IF NOT EXISTS");
    database.command("sql", "CREATE PROPERTY Article.id IF NOT EXISTS INTEGER");
    database.command("sql", "CREATE PROPERTY Article.title IF NOT EXISTS STRING");
    database.command("sql", "CREATE INDEX IF NOT EXISTS ON Article(id) UNIQUE");
    database.commit();
    System.out.println("Schema created successfully");
} catch (Exception e) {
    database.rollback();
    System.err.println("Schema creation failed: " + e.getMessage());
}
```

Execute Queries

The gRPC API provides optimized query execution with support for both simple and parameterized queries:

```
// Simple query
ResultSet resultSet = database.query("sql", "SELECT * FROM Article");

// Process results
while (resultSet.hasNext()) {
    Result result = resultSet.next();
    System.out.println("Article: " + result.toMap());
}
```

Using parameterized queries (recommended for security and performance):

```
// Named parameters
ResultSet resultSet = database.query(
```

```
    "sql",
    "SELECT * FROM Article WHERE author = :author AND published > :date",
    Map.of(
        "author", "John Doe",
        "date", LocalDateTime.now().minusDays(30)
    )
);

// Positional parameters
ResultSet resultSet2 = database.query(
    "sql",
    "SELECT * FROM Article WHERE id = ? AND published = ?",
    1,
    LocalDateTime.now()
);
```

Working with query results:

```
// Convert results to a list of maps
List<Map<String, Object>> articles = database.query("sql", "SELECT * FROM Article")
    .stream()
    .map(Result::toMap)
    .collect(Collectors.toList());

// Find first result
Optional<Map<String, Object>> firstArticle = database.query(
        "sql",
        "SELECT * FROM Article WHERE id = :id",
        Map.of("id", 1)
    )
    .stream()
    .findFirst()
    .map(Result::toMap);

if (firstArticle.isPresent()) {
    System.out.println("Found article: " + firstArticle.get());
}
```

Records come back in the same shape as with the embedded API *(since v26.12.1)*. A row that returns a record, such as Cypher `MATCH (n:Person) RETURN n`, is that vertex or edge with its own properties, so `result.getVertex().get().get("name")` answers the name. A column that holds a record, such as `a`, `r` and `b` in `RETURN a, r, b`, SQL `@this`, or each record of `collect(n)`, is a `Document`, `Vertex` or `Edge` instead of a raw `Map`, and the columns keep the order of the projection. Earlier versions answered `RETURN n` with a vertex whose only property was `n` (issue #12196). A map is read as a record when it has a valid `@rid` and its `@type` names a type of the schema, so a map a query builds with those two keys is returned as that record too. A projection row with a `@type` column, such as `SELECT @type, name FROM Person`, is a projection

row with both columns, not a record of that type (issue #12519).

A link, in a record property or in a projection column, is read back as a `RID`, and a list or map of links as a list or map of `RID`s, as the embedded API answers them *(since v26.12.1)*. Earlier versions answered the rid text, such as `"#1:0"`, so `(Identifiable) doc.get("friend")` threw `ClassCastException` (issue #12195). The `RID` is not bound to the remote database, so load the record it points to with `database.lookupByRID(rid, true)` (issue #12603).

Update Data

Updating existing records is straightforward with the gRPC API:

```
// Simple update with positional parameters
database.command(
    "sql",
    "UPDATE Article SET title = ? WHERE id = ?",
    "Updated Article Title",
    3
);

// Update with named parameters
database.command(
    "sql",
    "UPDATE Article SET title = :title, content = :content WHERE id = :id",
    Map.of(
        "title", "Understanding gRPC in ArcadeDB",
        "content", "gRPC provides high-performance communication...",
        "id", 3
    )
);

// Update with complex expressions
database.command(
    "sql",
    "UPDATE Article SET published = :now, views = views + 1 WHERE id = :id",
    Map.of(
        "now", LocalDateTime.now(),
        "id", 3
    )
);
```

A persistent record set as a property value (`doc.set("owner", otherDocument)`), or passed as a query parameter, is sent as a link to that record and stored as a `LINK`, as the embedded API does *(since v26.12.1)*. Earlier versions stored it as the text of the record, such as `"#1:0@Doc[name=target]"`, so `SELECT owner.name` answered `null` (issue #12197). A document that was never saved is still stored as an embedded document.

Batch updates using transactions:

```
database.transaction(() -> {
    List<Integer> articleIds = List.of(1, 2, 3, 4, 5);

    for (Integer id : articleIds) {
        database.command(
            "sql",
            "UPDATE Article SET views = views + 1 WHERE id = ?",
            id
        );
    }

    System.out.println("Updated " + articleIds.size() + " articles");
});
```

Insert Data

The gRPC API supports multiple approaches for inserting data:

```
// Traditional INSERT with values
database.command(
    "sql",
    "INSERT INTO Article (id, title, author, published) VALUES (?, ?, ?, ?)",
    1,
    "Getting Started with ArcadeDB",
    "Jane Smith",
    LocalDateTime.now()
);

// INSERT with named parameters
database.command(
    "sql",
    "INSERT INTO Article (id, title, author, published) VALUES (:id, :title, :author,
:published)",
    Map.of(
        "id", 2,
        "title", "Advanced Query Techniques",
        "author", "John Doe",
        "published", LocalDateTime.now()
    )
);
```

Using JSON content for complex documents:

```
// INSERT with JSON CONTENT
String jsonContent = """
    {
```

```
        "id": 3,
        "title": "gRPC Performance Guide",
        "content": "Learn how to optimize your gRPC implementations...",
        "author": "Alice Johnson",
        "published": "2024-03-15T10:30:00",
        "tags": ["performance", "gRPC", "optimization"],
        "metadata": {
            "category": "Tutorial",
            "difficulty": "Intermediate"
        }
    }
    """;

database.command("sql", "INSERT INTO Article CONTENT " + jsonContent);
```

Bulk insert operations:

```
database.transaction(() -> {
    List<Map<String, Object>> articles = List.of(
        Map.of("id", 10, "title", "Article 10", "author", "Author A"),
        Map.of("id", 11, "title", "Article 11", "author", "Author B"),
        Map.of("id", 12, "title", "Article 12", "author", "Author C")
    );

    for (Map<String, Object> article : articles) {
        database.command(
            "sql",
            "INSERT INTO Article (id, title, author) VALUES (:id, :title, :author)",
            article
        );
    }

    System.out.println("Inserted " + articles.size() + " articles");
});
```

Create Vertices with RETURN Clause

When creating vertices, you often need to retrieve the created record's RID (Record ID) or other properties. The RETURN clause makes this efficient:

```
// Create vertex and return the new record
ResultSet result = database.command(
    "sql",
    "CREATE VERTEX Article SET id = ?, title = ?, author = ? RETURN @this",
    100,
    "Real-time Data Processing",
    "Bob Wilson"
);
```

```
// Get the created vertex
if (result.hasNext()) {
    Result vertex = result.next();
    String rid = vertex.getIdentity().toString();
    System.out.println("Created vertex with RID: " + rid);
    System.out.println("Vertex data: " + vertex.toMap());
}
```

Return specific properties:

```
// Return only the RID
ResultSet result = database.command(
    "sql",
    "CREATE VERTEX Article SET id = :id, title = :title RETURN @rid",
    Map.of(
        "id", 101,
        "title", "Graph Traversal Patterns"
    )
);

if (result.hasNext()) {
    Result vertex = result.next();
    System.out.println("New vertex RID: " + vertex.getProperty("@rid"));
}

// Return multiple properties
ResultSet result2 = database.command(
    "sql",
    "CREATE VERTEX Article SET id = :id, title = :title, published = :now " +
    "RETURN @rid, id, title",
    Map.of(
        "id", 102,
        "title", "Distributed Database Architecture",
        "now", LocalDateTime.now()
    )
);
```

Creating edges between vertices:

```
// Create vertices
ResultSet author = database.command(
    "sql",
    "CREATE VERTEX Author SET name = ? RETURN @rid",
    "Emily Davis"
);

if (author.hasNext()) {
    String authorRid = author.next().getProperty("@rid").toString();
```

```
    ResultSet article = database.command(
        "sql",
        "CREATE VERTEX Article SET title = ? RETURN @rid",
        "Mastering Graph Databases"
    );

    if (article.hasNext()) {
        String articleRid = article.next().getProperty("@rid").toString();

        // Create edge connecting them
        database.command(
            "sql",
            "CREATE EDGE Authored FROM ? TO ?",
            authorRid,
            articleRid
        );

        System.out.println("Created relationship between author and article");
    } else {
        System.err.println("Failed to create article, no RID returned.");
    }
} else {
    System.err.println("Failed to create author, no RID returned.");
}
```

4.12.6. Transaction Management

The gRPC API provides comprehensive transaction support for ensuring data consistency:

```
// Explicit transaction control
database.begin();
try {
    database.command("sql", "CREATE VERTEX Article SET id = 200");
    database.command("sql", "CREATE VERTEX Article SET id = 201");
    database.commit();
    System.out.println("Transaction committed successfully");
} catch (Exception e) {
    database.rollback();
    System.err.println("Transaction rolled back due to error: " + e.getMessage());
}

// Using transaction lambda (recommended)
database.transaction(() -> {
    database.command("sql", "CREATE VERTEX Article SET id = 300");
    database.command("sql", "CREATE VERTEX Article SET id = 301");
    // Automatically commits if no exception, rolls back on exception
});
```

4.12.7. Error Handling

Proper error handling is essential for robust applications:

```
try {
    RemoteGrpcDatabase database = new RemoteGrpcDatabase(
        server, "localhost", 50051, 2480, "mydb", "root", "password"
    );

    ResultSet results = database.query("sql", "SELECT * FROM Article");
    // Process results...

} catch (SecurityException e) {
    System.err.println("Authentication failed: " + e.getMessage());
} catch (DatabaseOperationException e) {
    System.err.println("Database operation failed: " + e.getMessage());
} catch (Exception e) {
    System.err.println("Unexpected error: " + e.getMessage());
    e.printStackTrace();
}
```

Since 26.10.2 a statement stopped on the server (by `terminate query`, Cypher `TERMINATE TRANSACTIONS` or Studio) raises `com.arcadedb.exception.QueryTerminatedException`. It is not a transient error and is never retried. Cancelling a call, or letting its deadline expire, also stops the server-side work. See Cancellation and running statements.

Since 26.10.2 a write whose answer was lost is not reported as retryable. When the connection fails with a bare `UNAVAILABLE` (no exception class from the server, and not a failure to connect) after an admin write of `RemoteGrpcServer` or a streaming write of `RemoteGrpcDatabase` (`ingestStream`, `ingestBidi`, `timeSeriesWriteStream`, graph batch loads) was sent, the client raises a `RemoteException` instead of a `NeedRetryException`: the server may already have applied the write, or part of a stream, and a replay could apply it twice. Read the data back before sending it again. Reads and `validate_only` dry runs are still retryable. A client-side deadline that expires before any answer raises a `TimeoutException` whose message says the write may already have been applied. A commit, a command or a single-record write sent outside a transaction follows the same rule.

`RemoteGrpcServer` also exposes `acceptDivergedDatabase(database)` and `acceptStaleSnapshot()` (since 26.10.2), the root-only overrides that lift a quarantine or a stale-snapshot read floor where no peer can serve a resync. `acceptDivergedDatabase` is allowed on a node that is the only voter of its cluster, or when every voter holds the database quarantined; `acceptStaleSnapshot` on a sole voter only. A node with nothing to lift (for `acceptDivergedDatabase`, a database with no quarantine and no read floor) raises the client's not-found error, and a node that has peers to resync from, or runs without HA, its failed-precondition error. See the `AcceptDivergedDatabase` and `AcceptStaleSnapshot` RPCs in gRPC API, and Lifting a quarantine without a resync and Lifting the stale-snapshot read floor for when to use them.

Since 26.11.1 a request to a database the server has but does not serve right now (closed by `close database`, being restored or being dropped) raises `com.arcadedb.exception.DatabaseNotAvailableException`, whose `getReason()` names the state (`CLOSED`,

`RESTORING`, `DROPPING`, ...), as over HTTP. It is not retried: `transaction()` fails once instead of retrying until an operator reopens the database. Previously it surfaced as a retryable `NeedRetryException`.

Since 26.12.1 a failure that arrives on a later batch of a streamed read (`iterateType`, `iterateBucket`, `queryStream`) is mapped to the same exception a unary call raises, for example `com.arcadedb.exception.TimeoutException` for an expired deadline and `SecurityException` for an authentication or permission refusal, instead of a bare `RuntimeException`. An interrupt while waiting for a batch keeps the thread's interrupt flag set and surfaces as a `RemoteException`. A `UNAUTHENTICATED` status that carries no exception class from the server raises a `SecurityException`, as the HTTP client does, on every call including `RemoteGrpcServer.listDatabases()`.

A create refused because the user, API token or database name is already taken raises `com.arcadedb.remote.RemoteAlreadyExistsException` (since 26.10.2), the same exception the HTTP client raises; see RemoteServer Class.

4.12.8. Performance Tips

To get the best performance from the gRPC API:

1. **Use Connection Pooling**: Create a single `RemoteGrpcServer` instance and reuse it across your application
2. **Batch Operations**: Group multiple operations in transactions to reduce network round-trips
3. **Parameterized Queries**: Always use parameterized queries for better performance and security
4. **Use sqlscript**: For multiple statements, use `sqlscript` to execute them in a single transaction
5. **Close Resources**: Ensure proper cleanup of database instances, especially in multi-threaded environments
6. **Stream Results**: For large result sets, process results as you iterate rather than loading everything into memory
7. **Enable TLS in Production**: Use TLS/SSL in production for security without significant performance impact

4.12.9. Complete Example

Here's a complete example demonstrating the gRPC API usage:

```
import com.arcadedb.remote.grpc.RemoteGrpcServer;
import com.arcadedb.remote.grpc.RemoteGrpcDatabase;
import com.arcadedb.query.sql.executor.Result;
import com.arcadedb.query.sql.executor.ResultSet;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Map;

public class ArcadeDBGrpcExample {
    public static void main(String[] args) {
```

```
        // Connect to gRPC server
        try (RemoteGrpcServer server = new RemoteGrpcServer(
            "localhost", 50051, "root", "password", false, List.of()
        )) {
            // Create database if it doesn't exist
            if (!server.exists("mydb")) {
                server.create("mydb");
            }

            // Create database instance
            try (RemoteGrpcDatabase database = new RemoteGrpcDatabase(
                server, "localhost", 50051, 2480, "mydb", "root", "password"
            )) {
                // Create schema
                String schema = """
                    CREATE VERTEX TYPE Article IF NOT EXISTS;
                    CREATE PROPERTY Article.id IF NOT EXISTS INTEGER;
                    CREATE PROPERTY Article.title IF NOT EXISTS STRING;
                    CREATE PROPERTY Article.author IF NOT EXISTS STRING;
                    CREATE INDEX IF NOT EXISTS ON Article(id) UNIQUE;
                    """;
                database.command("sqlscript", schema);

                // Insert data
                database.transaction(() -> {
                    database.command(
                        "sql",
                        "INSERT INTO Article (id, title, author, published) VALUES
(:id, :title, :author, :published)",
                        Map.of(
                            "id", 1,
                            "title", "Introduction to gRPC",
                            "author", "Jane Doe",
                            "published", LocalDateTime.now()
                        )
                    );
                });

                // Query data
                ResultSet results = database.query(
                    "sql",
                    "SELECT * FROM Article WHERE author = :author",
                    Map.of("author", "Jane Doe")
                );

                // Process results
                results.stream()
                    .map(Result::toMap)
                    .forEach(article ->
                        System.out.println("Article: " + article.get("title"))
                    );
```

```
                System.out.println("gRPC API example completed successfully");
            }
        }
    }
}
```

4.12.10. See Also

- Java API (Remote) - HTTP-based remote API
- Java API (Local) - Embedded database API
- Java API Reference - Complete API documentation

4.13. JDBC Driver

If you're using Java you can use the Postgres JDBC driver. This means the Postgres Plugin needs to be loaded by the server.

```
Class.forName("org.postgresql.Driver");

Properties props = new Properties();
props.setProperty("user", "user");
props.setProperty("password", "password");
props.setProperty("ssl", "false");

try (Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost/mydb",
props) ) {
  try (Statement st = conn.createStatement()) {
    st.executeQuery("create vertex type Hero");
    st.executeQuery("create vertex Hero set name = 'Jay', lastName = 'Miner'");

    PreparedStatement pst = conn.prepareStatement("create vertex Hero set name = ?,
lastName = ?");
    pst.setString(1, "Rocky");
    pst.setString(2, "Balboa");
    pst.execute();
    pst.close();

    try( ResultSet rs = st.executeQuery("SELECT * FROM Hero") ) { // Type and property
names are case sensitive!
      while (rs.next()) {
        System.out.println("First Name: " + rs.getString(1) + " - Last Name: " + rs
.getString(2));
      }
    }
  }
}
```

4.14. Neo4j BOLT Protocol

ArcadeDB Server supports the Neo4j BOLT protocol, enabling connectivity from any BOLT-compatible client or driver. This allows you to use the official Neo4j drivers with ArcadeDB, leveraging the native OpenCypher query engine for graph operations.

The BOLT implementation is certified against all five official Neo4j drivers (Java, JavaScript, Python, .NET, and Go) through a shared conformance spec. See the Driver Compatibility Matrix for the exact, per-driver certification status.

The BOLT protocol implementation supports:

- **BOLT v3.0, v4.0, v4.4, and v5.0–v5.4** protocol versions, negotiated automatically with the client
- **Full Cypher query support** via ArcadeDB's native OpenCypher implementation
- **Parameterized queries** for security and performance
- **TLS/SSL** encrypted connections (`bolt+s://`, `neo4j+s://`)
- **Explicit and managed transactions**, including automatic retry on transient errors
- **Bookmarks** for causal consistency across sessions
- **Multi-database support** with database selection per session
- **HA-aware routing** so `neo4j://` discovery reflects the actual cluster topology
- **Native BOLT structures** for graph (Node, Relationship, Path), temporal, `Duration`, and spatial `Point` values
- **Multi-label vertices** following Neo4j's node labeling conventions

4.14.1. Choosing a protocol for low latency

For single-row reads and writes from Python on the same host, a community measurement (loopback, one connection) ranked the options as follows, relative to HTTP with a persistent connection (lower is faster):

Route	One-row Cypher read
PostgreSQL wire with the `{cypher}` prefix (psycopg 3)	~0.5x
HTTP with a persistent connection (`http.client`)	1x
gRPC	~1.1x
BOLT (`neo4j` Python driver)	~1.6x
HTTP through `requests.Session`	~4x

Most of the difference is client-side cost in the Python drivers, so treat the numbers as a guide and measure your own workload. The PostgreSQL wire uses positional parameters (`$1`, `$2`) for Cypher. Reuse connections and sessions on every route: opening one per request dominates the latency.

BOLT sends the records of a `PULL` batched in a few large network writes, so a slow, long-running query shows its first rows when a batch fills or the `PULL` ends. Drivers fetch in batches (1000 records

by default), which makes this invisible in normal use.

4.14.2. Setup

If you're using ArcadeDB as embedded, add the dependency to the `arcadedb-bolt` library. If you're using Maven, include this dependency in your `pom.xml` file:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-bolt</artifactId>
    <version>26.11.1</version>
</dependency>
```

To start the BOLT plugin, enlist it in the `server.plugins` settings. To specify multiple plugins, use the comma `,` as separator. Example:

```
~/arcadedb $ bin/server.sh -Darcadedb.server.plugins
="Bolt:com.arcadedb.bolt.BoltProtocolPlugin"
```

If you're using MS Windows OS, replace `server.sh` with `server.bat`.

In case you're running ArcadeDB with Docker, use `-e` to pass settings and open the BOLT default port `7687`:

```
docker run --rm -p 2480:2480 -p 7687:7687 \
       --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata \
          -Darcadedb.server.plugins=Bolt:com.arcadedb.bolt.BoltProtocolPlugin " \
          arcadedata/arcadedb:latest
```

The Server output will contain this line:

```
2026-10-29 05:19:34.813 INFO  [BoltNetworkListener] <ArcadeDB_0> Listening for
incoming BOLT connections on /0.0.0.0:7687 (protocol v.4, TLS: DISABLED)
2026-10-29 05:19:34.814 INFO  [PluginManager] <ArcadeDB_0> - Neo4j-Bolt plugin started
```

4.14.3. Configuration

The BOLT plugin supports the following configuration options:

Setting	Default	Description
`arcadedb.bolt.port`	7687	TCP/IP port for BOLT connections

Setting	Default	Description
`arcadedb.bolt.host`	0.0.0.0	Host/IP address to bind to. A name that resolves to several local addresses, such as `localhost` (`127.0.0.1` and `::1`), is bound on every one of them; a port held on any of them is skipped
`arcadedb.bolt.defaultDatabase`	(none)	Default database when not specified by the client; if unset, the first database the user has access to is used
`arcadedb.bolt.maxConnections`	0	Maximum concurrent connections (0 = unlimited)
`arcadedb.bolt.ssl`	DISABLED	TLS mode for BOLT connections: `DISABLED`, `OPTIONAL`, or `REQUIRED` (see TLS/SSL)
`arcadedb.bolt.routing.ttl`	300	Time-to-live in seconds for routing table entries returned to `neo4j://` clients
`arcadedb.bolt.maxOpenStreams`	1024	Maximum number of result streams one connection may keep open at the same time (see Transactions)
`arcadedb.bolt.debug`	false	Enable BOLT protocol debug logging

Example configuration:

```
bin/server.sh -Darcadedb.server.plugins="Bolt:com.arcadedb.bolt.BoltProtocolPlugin" \
              -Darcadedb.bolt.port=7687 \
              -Darcadedb.bolt.defaultDatabase=mydatabase
```

4.14.4. Access Control

A BOLT connection can only use the databases its user has access to, as with every other ArcadeDB protocol:

- Selecting a database the user has no access to (in the session, `RUN` or `BEGIN`) fails with `Neo.ClientError.Security.Forbidden`. The same error is returned for a database that does not exist, so a user cannot discover which databases the server hosts. The default database is checked the same way.
- `SHOW DATABASES` lists only the databases the user has access to (plus the virtual `system` entry). *(Since v26.11.1)* the list includes the databases that are on disk but not open (not opened at startup, or closed for idleness, see Closing idle databases), and the connection that names no database while `arcadedb.bolt.defaultDatabase` is not set picks the first database the user is granted among them too. The first request on such a database opens it.
- `SHOW INDEXES`, `SHOW CONSTRAINTS`, `CALL db.labels()`, `db.relationshipTypes()`, `db.propertyKeys()` and `db.schema.visualization()` hide the types the user is not allowed to read.

(Since v26.10.2.) The user is checked again on every request, so a user deleted, given a new password or stripped of a database grant after the connection logged in loses access on its next

request. See User changes on open connections.

4.14.5. TLS/SSL

The BOLT plugin supports encrypted connections. Set `arcadedb.bolt.ssl` to one of:

Mode	Behavior
`DISABLED`	(default) plaintext only. Clients connect with `bolt://` / `neo4j://` and encryption off.
`OPTIONAL`	the server auto-detects the connection: a TLS handshake is served encrypted, a plaintext connection is served in the clear. Useful during migration.
`REQUIRED`	only encrypted connections are accepted. Clients must use `bolt+s://` / `neo4j+s://`.

There is no STARTTLS-style upgrade: the server looks at the first bytes of each connection and starts a TLS handshake directly when the client speaks TLS. With `OPTIONAL`, TLS and plaintext share the same port, so a client configured without encryption sends its credentials in the clear. Switch to `REQUIRED` once certificates are in place.

When TLS is enabled (`OPTIONAL` or `REQUIRED`), the plugin reuses the server's shared HTTPS keystore and truststore. Configure them with the standard network settings:

```
bin/server.sh -Darcadedb.server.plugins="Bolt:com.arcadedb.bolt.BoltProtocolPlugin" \
              -Darcadedb.bolt.ssl=REQUIRED \
              -Darcadedb.network.ssl.keyStore=/path/to/keystore.p12 \
              -Darcadedb.network.ssl.keyStorePassword=secret \
              -Darcadedb.network.ssl.trustStore=/path/to/truststore.jks \
              -Darcadedb.network.ssl.trustStorePassword=secret
```

The keystore defaults to PKCS12 and the truststore to JKS. Connect from a driver using the `bolt+s://` (or `neo4j+s://` for routing) scheme.

4.14.6. Server Identity and Feature Envelope

Neo4j drivers gate feature negotiation off the server identity they receive during the initial handshake. ArcadeDB deliberately advertises a Neo4j-compatible identity so that the official drivers enable the correct feature set:

- In the `HELLO`/`SUCCESS` response the server reports itself as `Neo4j/5.26.0 compatible (ArcadeDB <version>)`.
- `CALL dbms.components()` returns `Neo4j Kernel`, version `5.26.0`, edition `community`.

This advertised identity is the contract against which every certification result is measured: it declares the Neo4j 5.26-era feature envelope the drivers may assume. The following capabilities are **certified as working** against the official drivers:

- Protocol negotiation for BOLT 3.0/4.0/4.4 and 5.0–5.4.

- Basic authentication.
- Autocommit, explicit (`BEGIN`/`COMMIT`/`ROLLBACK`), and managed transactions with retry-on-transient.
- Bookmarks (causal consistency) and per-session database selection.
- Streaming (`PULL`/`DISCARD`) with accurate result-summary counters.
- Native BOLT structures for Node, Relationship, Path, temporal types, `Duration`, spatial `Point`, byte arrays, and nested lists/maps.
- Structured `Neo.ClientError.` and `Neo.TransientError.` error codes.
- HA-aware `neo4j://` routing.

Explicit, documented differences from a real Neo4j server (never silent omissions):

- The `none` authentication scheme is **intentionally rejected** — ArcadeDB always requires credentials.
- `RID` and `UUID` values are serialized as strings (there is no native BOLT type for them).
- `BigDecimal` and out-of-range `BigInteger` values are down-converted to `double`, which may lose precision.
- Aura and enterprise-clustering-specific driver behaviors are out of scope.
- The advertised `server_agent` version (`5.26.0`) is a fixed, deliberate choice; changing it is a compatibility-affecting decision.

4.14.7. Compatible Drivers

The Neo4j official drivers are open source and licensed under Apache 2.0. You can use them with ArcadeDB's BOLT protocol implementation:

Language	Driver	Installation
Java	neo4j-java-driver	Maven: `org.neo4j.driver:neo4j-java-driver`
Python	neo4j	`pip install neo4j`
JavaScript	neo4j-driver	`npm install neo4j-driver`
.NET	Neo4j.Driver	NuGet: `Neo4j.Driver`
Go	neo4j-go-driver	`go get github.com/neo4j/neo4j-go-driver/v5`

Each driver is certified across a pinned version range that is re-tested nightly against driver-side releases. The exact versions are tracked in the driver-version matrix; see the Driver Compatibility Matrix for per-scenario results.

For the complete list of community drivers, check the Neo4j Driver documentation.

4.14.8. Connection URI Schemes

Scheme	Meaning
`bolt://`	Direct connection to a single server, plaintext.
`bolt+s://`	Direct connection to a single server, TLS-encrypted.
`neo4j://`	Routing/discovery connection — the driver asks the server for the cluster routing table (HA-aware).
`neo4j+s://`	Routing/discovery connection, TLS-encrypted.

4.14.9. Java Example

```
import org.neo4j.driver.*;

// Create driver (without encryption for local development)
Driver driver = GraphDatabase.driver(
    "bolt://localhost:7687",
    AuthTokens.basic("root", "playwithdata"),
    Config.builder().withoutEncryption().build()
);

// Open session for specific database
try (Session session = driver.session(SessionConfig.forDatabase("mydatabase"))) {

    // Execute a simple query
    Result result = session.run("MATCH (n:Person) RETURN n.name AS name LIMIT 10");
    while (result.hasNext()) {
        Record record = result.next();
        System.out.println(record.get("name").asString());
    }

    // Execute parameterized query
    Result paramResult = session.run(
        "MATCH (p:Person) WHERE p.age >= $minAge RETURN p.name, p.age",
        Values.parameters("minAge", 25)
    );

    // Execute in explicit transaction
    try (Transaction tx = session.beginTransaction()) {
        tx.run("CREATE (n:Person {name: $name, age: $age})",
               Values.parameters("name", "Alice", "age", 30));
        tx.run("CREATE (n:Person {name: $name, age: $age})",
               Values.parameters("name", "Bob", "age", 25));
        tx.commit();
    }
}

driver.close();
```

4.14.10. Python Example

```
from neo4j import GraphDatabase

# Create driver
driver = GraphDatabase.driver(
    "bolt://localhost:7687",
    auth=("root", "playwithdata")
)

# Query example
with driver.session(database="mydatabase") as session:
    # Simple query
    result = session.run("MATCH (n:Person) RETURN n.name AS name LIMIT 10")
    for record in result:
        print(record["name"])

    # Parameterized query
    result = session.run(
        "MATCH (p:Person) WHERE p.age >= $minAge RETURN p.name, p.age",
        minAge=25
    )

    # Explicit transaction
    with session.begin_transaction() as tx:
        tx.run("CREATE (n:Person {name: $name, age: $age})", name="Alice", age=30)
        tx.run("CREATE (n:Person {name: $name, age: $age})", name="Bob", age=25)
        tx.commit()

driver.close()
```

4.14.11. JavaScript Example

```
const neo4j = require('neo4j-driver');

// Create driver
const driver = neo4j.driver(
    'bolt://localhost:7687',
    neo4j.auth.basic('root', 'playwithdata')
);

// Query example
const session = driver.session({ database: 'mydatabase' });

try {
    // Simple query
    const result = await session.run('MATCH (n:Person) RETURN n.name AS name LIMIT 10
');
    result.records.forEach(record => {
```

```
        console.log(record.get('name'));
    });

    // Parameterized query
    const paramResult = await session.run(
        'MATCH (p:Person) WHERE p.age >= $minAge RETURN p.name, p.age',
        { minAge: 25 }
    );

    // Explicit transaction
    const tx = session.beginTransaction();
    await tx.run('CREATE (n:Person {name: $name, age: $age})', { name: 'Alice', age:
30 });
    await tx.run('CREATE (n:Person {name: $name, age: $age})', { name: 'Bob', age: 25
});
    await tx.commit();

} finally {
    await session.close();
    await driver.close();
}
```

4.14.12. Go Example

```
package main

import (
    "context"
    "fmt"

    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Create driver (empty realm string for basic auth)
    driver, err := neo4j.NewDriverWithContext(
        "bolt://localhost:7687",
        neo4j.BasicAuth("root", "playwithdata", ""),
    )
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)

    session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "mydatabase"})
    defer session.Close(ctx)
```

```
    // Simple query
    result, err := session.Run(ctx,
        "MATCH (n:Person) RETURN n.name AS name LIMIT 10", nil)
    if err != nil {
        panic(err)
    }
    for result.Next(ctx) {
        fmt.Println(result.Record().AsMap()["name"])
    }

    // Managed write transaction with automatic retry on transient errors
    _, err = session.ExecuteWrite(ctx, func(tx neo4j.ManagedTransaction) (any, error)
{
        return tx.Run(ctx,
            "CREATE (n:Person {name: $name, age: $age})",
            map[string]any{"name": "Alice", "age": 30})
    })
    if err != nil {
        panic(err)
    }
}
```

4.14.13. Cypher Query Examples

Since BOLT protocol uses Cypher as its query language, you can execute any query supported by ArcadeDB's OpenCypher implementation:

```
// Create vertices
CREATE (alice:Person {name: 'Alice', age: 30})
CREATE (bob:Person {name: 'Bob', age: 25})

// Create relationship
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2020}]->(b)

// Query with pattern matching
MATCH (p:Person)-[:KNOWS]->(friend)
WHERE p.age > 20
RETURN p.name, friend.name

// Variable-length paths
MATCH path = (start:Person)-[:KNOWS*1..3]->(end:Person)
RETURN path

// Aggregations
MATCH (p:Person)
RETURN avg(p.age) AS averageAge, count(p) AS totalPeople
```

4.14.14. Transactions

The BOLT protocol supports three transaction styles:

- **Auto-commit mode**: a query run outside a transaction is one transaction, committed when it completes. A query that fails leaves nothing behind *(since v26.11.1; before, each write clause of an auto-commit query committed on its own)*.
- **Explicit transactions**: use `BEGIN`/`COMMIT`/`ROLLBACK` for multi-statement transactions. Transactions are automatically rolled back if an error occurs.
- **Managed transactions**: the driver's `executeRead`/`executeWrite` transaction functions run your work in a retried transaction. If the server returns a `Neo.TransientError.*` (for example, a write-write conflict), the driver **automatically retries** the transaction function. Prefer this style for write paths that may contend.

```
// Auto-commit (implicit transaction)
session.run("CREATE (n:Person {name: 'Charlie'})");

// Explicit transaction with multiple operations
try (Transaction tx = session.beginTransaction()) {
    tx.run("CREATE (a:Person {name: 'David'})");
    tx.run("CREATE (b:Person {name: 'Eve'})");
    tx.run("MATCH (a:Person {name: 'David'}), (b:Person {name: 'Eve'}) CREATE (a)-
[:FRIENDS]->(b)");
    tx.commit();
}

// Managed transaction — retried automatically on a transient error
session.executeWrite(tx -> {
    tx.run("CREATE (n:Person {name: $name})", Values.parameters("name", "Grace"));
    return null;
});
```

Inside an explicit transaction you can run a second query before you have finished reading the results of the first one. This happens on its own whenever the driver's fetch size is smaller than the number of rows the first query returns, because the driver then keeps that result open while your code moves on:

```
// Driver configured with Config.builder().withFetchSize(1)
try (Transaction tx = session.beginTransaction()) {
    Result people = tx.run("MATCH (p:Person) RETURN p");
    people.next();                              // the first result is still open

    tx.run("MATCH (c:Company) RETURN c").consume();  // a second query in the same
transaction

    while (people.hasNext()) people.next();       // finish reading the first one
    tx.commit();
```

```
}
```

Each open result holds server-side resources until you read it to the end or discard it, so a single connection may keep at most `arcadedb.bolt.maxOpenStreams` of them open at once (1024 by default, far more than any driver uses).

`DISCARD n` discards the next `n` records of a result and, like `PULL n`, keeps it open (`has_more = true`) while records remain, so a following `PULL` or `DISCARD` reads on from there, as Bolt 4.0 and later define it. `DISCARD` with `n = -1` discards the rest. *(Since v26.12.1; before, any `DISCARD` discarded every remaining record and closed the result.)*

Access mode and transaction timeout

(Since v26.12.1.) The access mode and the timeout a driver declares are enforced:

- **Access mode**: a write sent in a READ transaction (`BEGIN` with mode `r`, which is what `session.executeRead` and a session opened with `AccessMode.READ` send) or as a READ auto-commit query (`RUN` with mode `r`) is refused before it runs with `Neo.ClientError.Statement.AccessMode`, as Neo4j does.
- **Transaction timeout**: the `tx_timeout` sent with `BEGIN` (`TransactionConfig.withTimeout()` in the drivers) bounds the whole explicit transaction. A statement that runs past what is left of it is stopped with `Neo.ClientError.Transaction.TransactionTimedOut`, and a `COMMIT` after the timeout is refused with the same code and the transaction is rolled back. The `tx_timeout` of an auto-commit `RUN` bounds that query. The timeout is enforced through the engine's command timeout (`arcadedb.command.timeout`): when both are set, the earlier one wins.

```
// Refused with Neo.ClientError.Statement.AccessMode
session.executeRead(tx -> tx.run("CREATE (n:Person {name: 'Heidi'})").consume());

// The whole transaction must finish within 5 seconds
try (Transaction tx = session.beginTransaction(TransactionConfig.builder().
withTimeout(Duration.ofSeconds(5)).build())) {
    tx.run("MATCH (p:Person) SET p.checked = true");
    tx.commit();
}
```

Before v26.12.1 both were ignored: a write ran in a READ transaction, and `tx_timeout` had no effect.

When an auto-commit write commits

An auto-commit write commits when its `RUN` executes, not at the end of its result stream as the Bolt specification says, so a `RESET` or a dropped connection between `RUN` and `PULL` does not roll it back. This is deliberate: ArcadeDB runs an auto-commit write as one transaction with server-side retry on MVCC conflicts, which deferring the commit would give up. The official drivers send `PULL` right after `RUN`, so ordinary driver use is unaffected, and so are explicit transactions.

4.14.15. Listing and terminating running statements

Since v26.10.2 the statements a Bolt client runs are listed and can be stopped like the ones of every other protocol: by Neo4j's own `SHOW TRANSACTIONS` and `TERMINATE TRANSACTIONS`, sent over Bolt as any other Cypher statement, or by the HTTP `list queries` and `terminate query` server commands. A statement appears with `protocol bolt` and `connectionId bolt-<n>` from its `RUN` until its result has been read to the end or discarded, so a result the client leaves open keeps it listed.

```
try (Session admin = driver.session(SessionConfig.forDatabase("mydatabase"))) {
    for (Record row : admin.run("SHOW TRANSACTIONS YIELD transactionId, username,
currentQuery, elapsedTime").list())
        System.out.println(row);

    admin.run("TERMINATE TRANSACTIONS $ids", Values.parameters("ids", List.of("q118-
5e1f0a9c"))).consume();
}
```

The terminated statement fails with `Neo.ClientError.Transaction.Terminated`, which drivers do not retry, not even in a managed transaction. A statement terminated inside an explicit transaction fails the transaction: it is rolled back, as after any other failure in it.

4.14.16. Causal Consistency (Bookmarks)

Bookmarks let a client guarantee it reads its own (or another session's) writes. Capture the bookmarks from one session and pass them to the next; the server will not serve the follow-up read until it has caught up to that bookmark. This works across sessions and is exercised by the certification suite.

```
Bookmark bookmark;
try (Session write = driver.session(SessionConfig.forDatabase("mydatabase"))) {
    write.run("CREATE (n:Person {name: 'Heidi'})").consume();
    bookmark = write.lastBookmark();
}

// A later session reads with the bookmark to observe the write
try (Session read = driver.session(SessionConfig.builder()
        .withDatabase("mydatabase")
        .withBookmarks(bookmark)
        .build())) {
    read.run("MATCH (n:Person {name: 'Heidi'}) RETURN n").list();
}
```

4.14.17. Data Type Mapping

ArcadeDB emits native BOLT/PackStream structures for the following, so drivers receive typed objects (not strings):

Category	Types
Graph	Node, Relationship, Path
Temporal	`Date`, `Time`, `LocalTime`, `DateTime`, `LocalDateTime` (DateTime uses the UTC encoding on BOLT 5.0+, legacy encoding on 4.x)
Other structured	`Duration`, spatial `Point` (2D/3D), byte arrays, nested lists and maps, null

Values without a native BOLT type — `RID` and `UUID` — are serialized as strings. `BigDecimal` and out-of-range `BigInteger` values are converted to `double` and may lose precision.

Temporal values (`Date`, `Time`, `LocalTime`, `DateTime`, `LocalDateTime`) are carried as native PackStream temporal structures in **both** directions. Read them with the driver's native temporal accessors — e.g. `Value.asZonedDateTime()` or `Value.asLocalDate()` — exactly as you would against Neo4j.

4.14.18. Error Codes

Server errors are mapped to Neo4j-style structured status codes so driver-side error handling and retry logic behave as expected:

Code	Raised when
`Neo.ClientError.Security.Unauthorized`	authentication failed or credentials were missing
`Neo.ClientError.Security.Forbidden`	an operation is not permitted for the user, including selecting a database the user has no access to (see Access Control)
`Neo.ClientError.Statement.SyntaxError`	the Cypher statement failed to parse, or calls a function that does not exist (`Unknown function 'apoc.coll.sortMulti'`). A namespaced function, which may be a user-defined one, is checked before the statement runs; before v26.11.1 an unknown one failed while running, with `UnknownError`
`Neo.ClientError.Statement.SemanticError`	the statement parsed but is semantically invalid, or names a type, an index, a property or another schema object that does not exist. Since v26.12.1 the missing-object cases are all `SemanticError`; before, many were `UnknownError`
`Neo.ClientError.Procedure.ProcedureNotFound`	a `CALL` (or `OPTIONAL CALL`) of a procedure that does not exist. Since v26.12.1; before, `UnknownError`
`Neo.ClientError.Statement.AccessMode`	a write sent in a READ transaction or a READ auto-commit query (see Access mode and transaction timeout). Since v26.12.1; before, the write ran

Code	Raised when
`Neo.ClientError.Statement.TypeError`	a value cannot be stored in a property (a map, or a list containing one, see Map-Valued Properties), or an operation met a value of a type it is not defined for while the query ran: a property read off a list (`UNWIND [[1]] AS l RETURN l.x`), an index into a value that is no list. Since v26.11.1 the runtime cases are `TypeError` too; before, `UnknownError` or `ArgumentError`
`Neo.ClientError.Transaction.TransactionNotFound`	a transaction reference is unknown
`Neo.ClientError.Schema.ConstraintValidationFailed`	a unique-index (constraint) violation, which is permanent, so drivers should not retry. Since v26.12.1 also a `DELETE` of a node that still has relationships (use `DETACH DELETE`); before, `UnknownError`
`Neo.ClientError.Statement.EntityNotFound`	an openCypher statement reads, sets or links a node it deleted earlier in the same statement (`MATCH (n:A) DELETE n RETURN n.x`). Since v26.12.1; before, `UnknownError`
`Neo.ClientError.Statement.ExternalResourceFailed`	a `LOAD CSV` URL that cannot be opened or read, such as a file that does not exist. Since v26.12.1; before, `UnknownError`
`Neo.ClientError.Schema.IndexWithNameAlreadyExists`, `Neo.ClientError.Schema.ConstraintWithNameAlreadyExists`, `Neo.ClientError.Schema.EquivalentSchemaRuleAlreadyExists`	a create or rename onto a name already held by an index, by a constraint, or by another schema object (a type, a property, …). Since v26.11.1; before, `SemanticError` or `UnknownError`. Since v26.12.1 an index or constraint equivalent to an existing one under a different name answers `EquivalentSchemaRuleAlreadyExists`, while a same-name duplicate keeps `IndexWithNameAlreadyExists` / `ConstraintWithNameAlreadyExists`
`Neo.ClientError.Transaction.TransactionTimedOut`	a query or transaction ran out of time, including the `tx_timeout` a driver sets (see Access mode and transaction timeout)
`Neo.ClientError.Transaction.Terminated`	the statement was stopped by `TERMINATE TRANSACTIONS`, `terminate query` or a terminate from Studio; drivers do not retry it (see Listing and terminating running statements). Since v26.10.2
`Neo.ClientError.Request.Invalid`	a malformed or out-of-sequence protocol message
`Neo.TransientError.Transaction.DeadlockDetected`	a concurrent-modification / lock-timeout conflict — drivers retry managed transactions on this code
`Neo.ClientError.Cluster.NotALeader`	a write sent to an HA replica that refuses it and names the leader. A routing (`neo4j://`) driver drops the node as writer and retries against the leader; a direct (`bolt://`) client receives the code with the leader's address in the message. See Routing and High Availability. Since v26.10.2
`Neo.DatabaseError.General.UnknownError`	an unexpected server-side failure

In `production` server mode the message of a `FAILURE` raised by the engine (a duplicated key, a syntax error, a failed commit) is replaced by "The request failed. Check the server log for the details", and the detail is written to the server log. The status code is kept, so driver-side error handling and retries are unchanged. Text Bolt writes itself about the request, such as the database name of a failed database selection, stays visible. Before v26.10.2 the message was sent as is in every mode. See Error details in production mode.

4.14.19. Routing and High Availability

`neo4j://` clients ask the server for a routing table. The response is HA-aware:

- **Single-node deployment**: the node advertises itself for all roles (WRITE, READ, ROUTE).
- **HA cluster with a known leader**: the leader is returned as WRITE + ROUTE and the followers as READ + ROUTE, so driver-side load balancing works. A node that joined the Raft group at runtime (a Kubernetes scale-up beyond `arcadedb.ha.serverList`, or a manual join) is listed too, as a follower or as the leader. *(Since v26.11.1: such a node was replicated to but left out of the routing table.)*
- **HA cluster mid-election** (leader unknown): the node is advertised as READ + ROUTE only, never WRITE, until a leader is elected.

The routing table's time-to-live is controlled by `arcadedb.bolt.routing.ttl` (default 300 s).

A write that reaches a replica and is refused there, rather than forwarded, fails with `Neo.ClientError.Cluster.NotALeader` *(Since v26.10.2)*. The message ends with the leader's Bolt address when the routing table knows one (`Reconnect to the leader at '<address>' (Bolt address) and retry`), otherwise with its HTTP address. A `neo4j://` driver re-routes on this code by itself; a `bolt://` client, which talks to one node only, has to reconnect to the named leader. During an election, when no leader is known yet, the refusal names nobody and keeps the retryable `Neo.TransientError.Transaction.DeadlockDetected`: retry shortly. See how-to/operations/ha.pdf.

For clusters where nodes expose the BOLT port on **different** ports, each node's client-reachable BOLT address must be declared using the object form of `arcadedb.ha.serverList` (`host:{raft:..,http:..,bolt:..}`). When omitted, each peer's BOLT address is derived from its Raft host plus this node's BOLT port, which is correct only for homogeneous deployments (for example, a Kubernetes StatefulSet).

When the derived addresses cannot tell two peers apart - nodes that share a host and differ only by port - no routing table is advertised at all, and the node answers ROUTE with itself as READ and ROUTE only, exactly as it does mid-election. Declaring `bolt` is what restores a WRITE entry. See the HA guide, *Client Routing Ports*.

4.14.20. Current Limitations

- **Authentication**: the `none` scheme is not accepted — ArcadeDB always requires credentials. This

is intentional, not a bug.

- **Type fidelity**: `RID` and `UUID` are returned as strings; `BigDecimal` / oversized `BigInteger` are down-converted to `double`.
- **Heterogeneous cluster ports**: see the note under Routing and High Availability.
- **Auto-commit commit point**: an auto-commit write commits when its `RUN` executes, not at the end of its result stream; see When an auto-commit write commits.

4.14.21. Troubleshooting

If you encounter connection issues:

1. **Enable debug logging**: Start the server with `-Darcadedb.bolt.debug=true` to see detailed protocol messages. Passwords are never written to the log.
2. **Check port availability**: Ensure port 7687 (or your configured port) is not in use by another service.
3. **Verify authentication**: ArcadeDB always requires authentication. Ensure you're providing valid credentials — the `none` scheme is rejected.
4. **Match the encryption mode**: If the server runs with `arcadedb.bolt.ssl=REQUIRED`, connect with `bolt+s://` / `neo4j+s://`. If TLS is `DISABLED` (the default), connect with `bolt://` / `neo4j://` and encryption turned off in the driver.

To connect without encryption against a plaintext server:

```
// Java
Config.builder().withoutEncryption().build()
```

```
# Python
driver = GraphDatabase.driver(uri, auth=auth, encrypted=False)
```

4.14.22. See Also

- Bolt Driver Compatibility Matrix
- OpenCypher Query Language
- PostgreSQL Protocol Plugin
- Neo4j BOLT Protocol Specification
- Neo4j Driver Documentation

4.14.23. Numeric ids

The numeric id of a node or relationship sent over Bolt (`Node.id()` in the 4.x drivers) is the same value the Cypher `id()` function returns, so `MATCH (n) WHERE id(n) = $id` finds the node it came from. It follows the `arcadedb.opencypher.idBucketBits` setting. Ids kept by a client from an earlier version are different and must be read again. Prefer `elementId()`, which is the record id (for example `#3:0`).

A relationship of a lightweight edge type has no record. *Since v26.12.1* it is sent with a negative numeric id derived from its element id, which can never equal the id of a record, and its element id is the same value `elementId(r)` returns (earlier versions failed the query instead of returning it). `id(r)` of such a relationship is `null`, so find it again by its element id, not by the numeric one.

4.15. Native Drivers

Which page do I want?

- Building a new application in Python, JavaScript/TypeScript or Go — start here.
- Prefer the clients you already have in Python — Python quickstart, which teaches `psycopg`, the Neo4j driver and `httpx` against ArcadeDB.
- Prefer the clients you already have in JavaScript/TypeScript — JavaScript / TypeScript quickstart, which teaches `pg`, `neo4j-driver` and `fetch` against ArcadeDB.
- Prefer the clients you already have in Go — Go quickstart, which teaches `pgx` against ArcadeDB.
- Already running a PostgreSQL stack — PostgreSQL wire protocol.
- Already running a Neo4j stack — Neo4j BOLT.
- Any other language — HTTP/JSON API, no driver required.

ArcadeDB publishes six native drivers, an HTTP one and a gRPC one for each of Python, JavaScript/TypeScript and Go, from the arcadedb-drivers repository. Every one of them is generated from a shared contract — an OpenAPI specification for the HTTP drivers, a Protobuf `.proto` for the gRPC drivers — and each driver's build regenerates from that contract and fails if the checked-in code and a fresh regeneration disagree.

That matters to you, not just to the people who build the drivers: a generated client cannot silently disagree with the server about the wire format. When ArcadeDB adds a field, the driver gets it from the contract rather than from someone remembering to hand-write it.

4.15.1. HTTP or gRPC?

Choose	When
HTTP	General application traffic, browsers, serverless functions, and anywhere you want the smallest dependency footprint. This is the right default.
gRPC	Throughput-sensitive server-to-server work: large result sets you want to stream rather than page, and bulk inserts.

There is no browser gRPC driver, and there cannot be one until the server changes. ArcadeDB's `GrpcServerPlugin` is plain grpc-java over HTTP/2, with no gRPC-Web handler and no Connect protocol in front of it. A browser cannot speak raw HTTP/2 gRPC framing at all, so no client library in any language can reach this server from a browser. Use an HTTP driver there.

4.15.2. The result envelope, and why `truncated` matters

The HTTP drivers' `query` and `command` do not return bare rows. They return the whole response envelope: `result`, `limit`, `returned`, and `truncated`.

`truncated` is true when the server's serializer hit its row cap while the query still had rows left to write. `result` is then a partial answer, not a short-but-complete one — and the two are indistinguishable by shape. A caller that reads `result` and ignores `truncated` will silently work off half an answer.

Always check `truncated` before treating `result` as the whole result set. When it is true, re-query with a narrower filter or a higher `limit`. Raising `limit` is not always the fix: a result whose true size exceeds the server's hard ceiling (`arcadedb.server.httpQueryMaxResultRows`) is refused outright with HTTP 413 rather than truncated, so past that ceiling a narrower filter is the only way forward.

4.15.3. Versions and server compatibility

All six drivers are released together, at one version, and each release is generated against one ArcadeDB server release's contract:

Driver version	ArcadeDB server	Packages
0.4.0	26.11.1	All six
0.3.0	26.10.2	All six: `arcadedb-driver` and `arcadedb-driver-grpc` (PyPI), `@arcadedb/driver` and `@arcadedb/driver-grpc` (npm), `github.com/ArcadeData/arcadedb-drivers/go/arcadedb` and `…/go/arcadedbgrpc` (Go modules)

Pointing a driver at a server on a materially different release may work for the endpoints both versions share, but is not tested or supported. Install commands in these pages are deliberately unpinned; the registries carry the current version. Each release's changes are listed in the arcadedb-drivers release notes.

4.15.4. Upgrading to 0.4.0

0.4.0 is generated against ArcadeDB 26.11.1. It adds a transaction isolation level to `transaction()` in all six drivers (`READ_COMMITTED` or `REPEATABLE_READ`), and a row limit to `command` in the Python, TypeScript and Go HTTP drivers, with the same meaning as on `query`. The isolation level relies on a field that the 26.11.1 contract introduced, so older servers are not tested. Three changes can affect existing code:

- **TypeScript: `limit` moved from `QueryOptions` to `CommandOptions`.** `QueryOptions` is still exported, as an alias of `CommandOptions`, so existing imports keep compiling.
- **Go HTTP: `WithLimit` now returns the exported `LimitOption`,** which is both a `QueryOption` and a `CommandOption`. Only code that holds the function itself as a `func(int) QueryOption` breaks.
- **Go gRPC: `Transaction` takes trailing `opts ...TransactionOption`.** Callers compile unchanged.

On a 26.11.1 server, wrong Basic credentials answer 401 instead of 403, and committing a terminated transaction answers 404 instead of a silent 204. The driver READMEs describe both, and

keep the old behaviour documented for earlier servers.

4.15.5. Upgrading to 0.3.0

0.3.0 adds server administration, login sessions and API tokens, listing and terminating running work, query tags, and a duplex insert session to the three HTTP drivers; each language's page below shows them. Three changes can affect existing code:

- **gRPC insert streams no longer copy the database into `InsertOptions.database`.** All three gRPC drivers send `database` on the first chunk only, as the `.proto` specifies. Every supported server reads it there. An older server reads only `options.database`, so against it a stream now reports its rows as received, inserts nothing, and raises no error. If you must use such a server, set `options.database` yourself.
- **A stream's in-band error carries the server's own status.** An error event in the middle of an HTTP query or command stream used to surface with status 200. It now carries the event's `status` (500, 503, 409 and so on), and falls back to 200 only when the event has none. Retry logic keyed on the status now sees the real one.
- **Go: the option types are split.** `QueryOption` is now a sealed interface, joined by `CommandOption` and `TransactionOption`. Existing calls compile unchanged; only code that implemented `QueryOption` itself, or passed a `[]QueryOption` to `Command`, needs updating.

4.16. Python Embedded Bindings

The embedded bindings run the ArcadeDB engine inside your Python process, so there is no network hop and no server to start. They ship as two packages, and both import as `arcadedb`:

- `arcadedb`, the default: the engine built as a GraalVM native-image library. It needs no Java and no JPype, starts fast and holds little memory.
- `arcadedb-jvm`: the engine on a Java runtime that the wheel bundles, reached through JPype. Pick it for heavy, long-running loads, where the JIT compiler pays off, or when you need the Java-object APIs.

Install one of them, not both. With both installed, `import arcadedb` raises an `ImportError` that names the two.

The bindings live in the ArcadeData/arcadedb-python repository, and their guides are in the ArcadeDB Python docs. This page covers the first steps.

4.16.1. Install

With uv:

```
uv add arcadedb
```

For the JVM build, install `arcadedb-jvm` instead.

Both packages support Python 3.11 to 3.14, and `arcadedb` also supports 3.15. Wheels exist for Linux

x86_64 and ARM64 (glibc 2.28 or later), macOS on Apple Silicon, and Windows x86_64.

These names start with release 26.11.1. The packages were formerly `arcadedb-embedded` (now `arcadedb-jvm`, the same build) and `arcadedb-embedded-native` (now `arcadedb`, archived on PyPI). The last `arcadedb-embedded` release, 26.11.1, is a deprecation shim: it installs `arcadedb`, the native build, not `arcadedb-jvm`. To stay on the JVM build, replace `arcadedb-embedded` with `arcadedb-jvm`.

4.16.2. Open a database and query it

```
import arcadedb

with arcadedb.create_database("./mydb") as db:
    db.command("sql", "CREATE VERTEX TYPE Person")
    db.command("sql", "CREATE PROPERTY Person.name STRING")
    db.command("sql", "CREATE PROPERTY Person.age INTEGER")

    with db.transaction():
        db.command("sql", "INSERT INTO Person SET name = ?, age = ?", "Alice", 30)
        db.command("sql", "INSERT INTO Person SET name = ?, age = ?", "Bob", 25)

    for row in db.query("sql", "SELECT name, age FROM Person WHERE age > ?", 26):
        print(row.get("name"), row.get("age"))
```

```
Alice 30
```

Reopen the same directory later with `arcadedb.open_database("./mydb")`. `create_database` raises if the path already holds a database. Pass `read_only=True` to `open_database` to refuse writes.

Queries in other languages use the same two calls, for example openCypher:

```
with arcadedb.open_database("./mydb") as db:
    for row in db.query("opencypher", "MATCH (p:Person) RETURN p.name AS name ORDER BY
name"):
        print(row.get("name"))
```

```
Alice
Bob
```

Schema, inserts and graph work are best written as SQL or openCypher through `db.command(…)` and `db.query(…)`. The bindings also expose wrapper objects for the rest of the Java API; the repository documents them.

Writes need a transaction. `INSERT`, `UPDATE` and `DELETE` go inside `with`

`db.transaction()`:. Schema statements apply immediately.

4.16.3. Pick the embedded bindings or the HTTP driver

	Embedded bindings	**HTTP driver (`arcadedb-driver`)**
Where the engine runs	In your Python process	In a separate ArcadeDB server
Who can reach the data	One process at a time: the database directory is locked while it is open	Any number of clients, local or remote
Network round trip	None	One per call
Setup	`uv add arcadedb`	A running server, plus `uv add arcadedb-driver`

Choose the embedded bindings for a single script, a notebook, or an application with one Python process that owns its data. Several threads of one process can share a database.

Choose the HTTP driver when more than one process, another language, or a remote machine must reach the same database, or when the server must outlive your Python process.

You can also combine the two. The bindings can start an in-process HTTP server (`arcadedb.create_server(⋯)`), so the process that owns the database keeps direct access while other clients connect over HTTP. That server lives and dies with your Python process. For a long-running service, run the regular ArcadeDB server instead.

4.16.4. Where to go next

- ArcadeDB Python docs: installation, which package to pick, databases and queries, concurrency, bulk loads and vectors.
- Bindings repository: the source, the examples and the issue tracker.
- Native drivers: the HTTP and gRPC clients, if you would rather talk to a server.
- Python quickstart: the PostgreSQL wire protocol route.

4.17. Python HTTP Driver

`arcadedb-driver` is ArcadeDB's Python HTTP client, generated from the server's OpenAPI contract. It wraps queries, commands and transactions in a small facade over `httpx`, plus a matching `asyncio` facade for async applications.

4.17.1. Install

```
pip install arcadedb-driver
```

Or with uv:

```
uv add arcadedb-driver
```

Requires Python 3.10 or later.

4.17.2. Connect and query

```
import httpx
from arcadedb_driver import ArcadeDBServer, basic_auth

with ArcadeDBServer(
    base_url="http://localhost:2480",
    auth=basic_auth("root", "playwithdata"),
    timeout=httpx.Timeout(5.0),
) as srv:
    db = srv.db("mydb")

    envelope = db.query(language="sql", command="SELECT FROM Person WHERE age > ?",
params={"0": 21})
    print(envelope.result, envelope.returned, envelope.truncated)
```

```
[{'@rid': '#1:0', '@type': 'Person', '@cat': 'v', 'name': 'Alice', 'age': 30},
{'@rid': '#1:1', '@type': 'Person', '@cat': 'v', 'name': 'Bob', 'age': 25}] 2 False
```

Positional parameters are zero-indexed — `{"1": 21}` silently returns nothing.

Positional parameters are keyed by their zero-based index as a string. For the single placeholder above, the key is `"0"`, not `"1"`. Passing `params={"1": 21}` does not raise and does not match the wrong rows — it runs the query with no bound value for `?` at all and comes back empty:

```
[] 0 False
```

That form looks plausible and returns quietly, which is exactly what makes it easy to ship. The driver's README showed `{"1": 21}` up to 0.2.0, so examples copied from an older release may carry the mistake. Always start positional parameter keys at `"0"`.

`basic_auth("root", "playwithdata")` builds the `Authorization` header for HTTP basic auth; `bearer_auth("AU-…")` substitutes directly for it wherever you already hold a session token from `srv.login()` (see Sessions and API tokens), or any other bearer credential.

An `EXPLAIN` or `PROFILE` statement answers with its execution plan instead of rows: the envelope's `explain` holds it as indented text and `explain_plan` as the same plan in structured form, while `result` is empty and `returned` is `0`. An empty `result` there is not "no rows matched".

4.17.3. Sync and async

Every facade method has an async twin. `AsyncArcadeDBServer` mirrors `ArcadeDBServer` method for method, just `await`-ed:

```
import asyncio
from arcadedb_driver import AsyncArcadeDBServer, basic_auth

async def main() -> None:
    async with AsyncArcadeDBServer(
        base_url="http://localhost:2480", auth=basic_auth("root", "playwithdata")
    ) as srv:
        envelope = await srv.db("mydb").query(language="sql", command="SELECT FROM
Person")
        print(envelope.result)

asyncio.run(main())
```

```
[{'@rid': '#1:0', '@type': 'Person', '@cat': 'v', 'name': 'Alice', 'age': 30},
{'@rid': '#1:1', '@type': 'Person', '@cat': 'v', 'name': 'Bob', 'age': 25}]
```

Beyond `query`, `command` and `transaction`, a database handle also exposes `db.ts`, `db.grafana` and `db.promql` for time-series ingestion and querying (including `db.promql.label_values`), and `db.vector` for vector, hybrid and full-text search — see the package README linked below for those. Everything in the sections below has the same async twin.

4.17.4. Close your client

Both `ArcadeDBServer` and `AsyncArcadeDBServer` are context managers, because each owns an `httpx` client with its own connection pool that must be released. Prefer `with` / `async with`. When you cannot use one — a long-lived client held across requests, say — call `close()` on the sync class or `await aclose()` on the async one yourself.

Timeouts are off by default.

Omitting `timeout` does not mean "use httpx's five-second default" — it means no timeout at all. `ArcadeDBServer` and `AsyncArcadeDBServer` default `timeout` to `None`, and in httpx an explicit `timeout=None` means exactly that: requests are free to hang forever on a stalled connection. Pass an explicit `httpx.Timeout` to bound requests, as in the example above: `timeout=httpx.Timeout(5.0)`.

4.17.5. Transactions

```
with db.transaction() as tx:
    tx.command(language="sql", command="INSERT INTO Account SET balance = 100")
    total = tx.query(language="sql", command="SELECT sum(balance) AS total FROM
```

```
Account").result[0]["total"]
print("total:", total)
```

```
total: 100
```

`db.transaction()` returns a context manager whose `enter` gives you a second database handle — here, `tx` — that carries the transaction's session id. The contract:

- If the block exits cleanly, the transaction commits.
- If the block raises, the transaction rolls back and the block's own exception propagates. If the rollback itself also fails, that failure is attached as the exception's `cause` rather than replacing it — the block's error is what you asked about.
- If the commit itself fails, a best-effort rollback is issued first, so the server-side session is not left open for `arcadedb.server.httpSessionExpireTimeout` to reap it.

The trap: calls made through the outer `db` handle while a transaction is open do not join that transaction — they auto-commit individually, outside it. Only calls made through the handle `transaction()` gives you (`tx` above) take part.

To choose the isolation level, pass `isolation_level="READ_COMMITTED"` or `isolation_level="REPEATABLE_READ"` to `db.transaction()` (driver 0.4.0 and later, against ArcadeDB 26.11.1 or later). Any other value raises `ValueError` before anything is sent. Without it the server begins the transaction at the database's default level.

4.17.6. Two error models

Facade methods raise `ArcadeDBError` on any non-2xx response, carrying `status`, `error`, `detail` and `request_id`. This holds for `query`, `command` and `transaction`, and equally for the `ts`, `grafana` and `promql` namespaces — every one of them raises the same `ArcadeDBError` rather than letting an `httpx` exception or anything else through:

```
from arcadedb_driver import ArcadeDBError

try:
    db.query(language="sql", command="SELECT FROM NoSuchType")
except ArcadeDBError as err:
    print(err.status, err.error, err.detail, err.request_id)
```

```
500 Internal error Type with name 'NoSuchType' was not found 7f43ebd2c31c5c12-24
```

`request_id` is the server's `X-Request-Id` for that call and is different on every request; use it to correlate a failure with the server log. Two more fields come from the response's headers: `retry_after`, the seconds the server asked you to wait before retrying (on a 503, or a 409 for an identical request still in flight), and `commit_index`, the Raft index to send back as `X-ArcadeDB-Read-After` on a later read. Both are `None` when the server sent none.

A 409 has two meanings. It is either "an identical request is still in flight", which is retryable after `retry_after`, or "query terminated", because an operator stopped the statement and its writes were rolled back. Retrying the second undoes the operator's decision. Tell them apart with `is_query_terminated(err)`, never by `err.status == 409`.

A rejected time-series write raises `TimeSeriesWriteError`, a subclass of `ArcadeDBError` that says how many samples were `written` and `dropped` and why. A write is not atomic: the `written` samples landed, so re-sending the whole batch writes them twice.

`srv.raw` takes the opposite approach: it never raises. It is not a second facade with its own `query` and `command` methods — it is the generated `openapi-python-client` `Client` object, the value you hand to the generated endpoint functions as their `client=` argument. Each of those functions returns a response whose `status_code` and `parsed` body you inspect yourself, the same shape you get for a successful call. Use `raw` when you would rather branch on a status code than catch an exception:

```
from arcadedb_driver import ArcadeDBServer, basic_auth
from arcadedb_driver._generated.api.database import list_databases

with ArcadeDBServer(base_url="http://localhost:2480", auth=basic_auth("root",
"playwithdata")) as srv:
    response = list_databases.sync_detailed(client=srv.raw)
    print(response.status_code, response.parsed.result)
```

```
200 ['mydb']
```

4.17.7. Labelling work with a query tag

`query`, `command`, `query_stream`, `command_stream`, `batch_load`, `batch_load_stream` and `transaction()` take an optional `query_tag`, sent as the `X-ArcadeDB-Query-Tag` header, so you can list and terminate the work by it:

```
db.query(language="sql", command="SELECT FROM Event", query_tag="nightly-report")

with db.transaction(query_tag="etl-run-42") as tx:
    tx.command(language="sql", command="INSERT INTO Event SET n = 1")  # labelled etl-
run-42
```

A transaction's tag is sent on the begin request only; the server applies it to every statement in the transaction that carries no tag of its own.

4.17.8. Duplex insert session

`db.insert_session()` opens an insert session on the server's `/ws` endpoint. Unlike `batch_load`, whose commit policy is fixed before the load starts, a session lets you read each chunk's acknowledgement

before deciding what to send next, and whether to commit at all:

```
with db.insert_session(target_type="Person") as session:
    for batch in batches:
        ack = session.send_chunk(batch)  # nothing is durable yet
        if ack["failed"]:
            session.rollback()
            break
    else:
        session.commit()
```

Leaving the `with` block with the session still open rolls it back.

4.17.9. Administration

`create_database`, `drop_database`, `create_user`, `update_user`, `drop_user`, `list_users`, `save_group`, `delete_group` and `list_groups` are on the server handle, and `server_command` runs any other server command. All of them are root-only:

```
srv.create_database("tenant_a")
srv.create_user("alice", "a-long-password", databases={"tenant_a": ["admin"]})
users = srv.list_users()
srv.drop_user("alice")
srv.drop_database("tenant_a")
```

A user or database that already exists is a 409 on ArcadeDB 26.10.2 and later. Older servers answer 403 for a duplicate user and 400 for a duplicate database.

4.17.10. Sessions and API tokens

```
# Log in as each user once, then keep a client that holds only that user's token.
login = srv.login_as("alice", "a-long-password")  # or srv.login()
with srv.with_token(login.token) as alice:         # srv is unchanged
    alice.list_databases()                          # only what alice may see
    alice.logout()

# A long-lived machine credential (root only).
created = srv.create_api_token("ci", database="tenant_a")
store(created.token)                          # the only time the plaintext exists
srv.delete_api_token(created.token_hash)   # by hash, never by token
```

`list_sessions` and `list_api_tokens` list what exists. Each user holds at most 100 sessions, and a further login silently evicts the oldest, so log in once per user and reuse the token rather than logging in per request. Wrong Basic credentials, like an invalid, expired or logged-out token, are answered 401; 403 is what an authenticated user gets for an operation it is not allowed (*since v26.11.1*: before, wrong Basic credentials were answered 403).

4.17.11. Running work

```
from datetime import timedelta
from arcadedb_driver import is_query_terminated

running = srv.list_queries(tag="nightly-report")  # or database=...
results = srv.terminate_queries_by_tag("nightly-report", wait=timedelta(seconds=3))
srv.terminate_query(running[0].id)
open_txs = srv.list_transactions()
srv.terminate_transaction(open_txs[0].id)
progress = srv.db("mydb").progress()  # long-running maintenance operations
```

Any authenticated user may call these: root sees and stops every statement, another user only their own. Each terminate answer carries a `status`, and which one you get decides whether you have anything to undo:

`status`	**Meaning**
`terminated`	It stopped, and what it wrote was rolled back.
`completed`	It finished before it noticed, so what it did stands.
`terminating`	It had not stopped when the wait ended; the request's own outcome is the answer.
`not found`	No such statement on this server, or another user's.

- **Each server sees only its own work.** These commands are not forwarded to the cluster leader, so behind a load balancer the server you reach may not be the one running the statement.
- **A commit after `terminate_transaction` fails on ArcadeDB 26.11.1 and later.** The transaction's writes are rolled back, and the server answers its owner's commit with 404. Earlier servers answered 204, as if it had committed (ArcadeData/arcadedb#10164).

4.17.12. Choosing a protocol from Python

ArcadeDB can be queried from Python over three protocols. For small results (a few rows per call) they perform the same; for **large results** the choice matters.

	HTTP API	**Bolt (`neo4j` driver)**	**gRPC (`grpcio`)**
Install	`pip install requests`	`pip install neo4j`	`pip install grpcio protobuf`, plus stubs generated from `arcadedb-server.proto`
Server setup	enabled by default (port 2480)	add the `Bolt` plugin (port 7687)	add the `GRPC` plugin (port 50051)

	HTTP API	Bolt (`neo4j` driver)	gRPC (`grpcio`)
Authentication	HTTP Basic	user and password	call metadata `x-arcade-user` / `x-arcade-password`
Transactions	yes (`/begin`, `/commit`)	yes	yes
Streams a large result	no, one JSON body	yes, in `fetch_size` batches	yes, in `batch_size` batches
633K vertex ids	0.54 s	2.59 s	**0.30 s**
633K PageRank scores	0.98 s	2.81 s	**0.79 s**
2M edge rows	1.61 s	8.55 s	**1.34 s**
Few rows per call (a count)	fast	fast	not measured

Times are for the whole result consumed in Python (warm median of 3). Bolt is shown with the whole result in one PULL (`fetch_size=-1`); with the driver's default of 1,000 rows per PULL the 2M-row call took about 15 s. gRPC is shown with 100,000 rows per batch; 10,000 is nearly as fast, while 1,000 is slower than HTTP on wider rows (1.8 s and 2.9 s for the last two calls).

Recommendation

- Large results from Python: use **gRPC streaming** with a batch size of 10,000 or more, or the HTTP API when you want the simplest setup.
- Use **Bolt** when you need Neo4j tooling and driver compatibility. From Python, set a large `fetch_size` for big results. The Python driver is the limit here, not the server: the official Java driver reads the same 2M rows over Bolt in 1.5 s.
- Small results, or many small calls: any of the three.

Measurement notes

- Server: ArcadeDB 26.10.2-SNAPSHOT image, with the Bolt buffering fix (#9165).
- Machine: Docker Desktop on macOS, on battery power, so only the ratios are meaningful. The benchmark script is `scripts/grpc_transfer_bench.py`, and the details are in `ArcadeDB-release-progress.md`.
- Rows are narrow (one or two numeric columns). Wide rows, strings and whole vertices or edges were not measured.
- Docker Desktop adds about 1.3 ms to each client-server round trip, which penalizes small batches. A native Linux server will show less of that penalty.

4.17.13. Next steps

- Native Drivers — the result envelope and why `truncated` matters, and how this driver compares to the others.
- Python — gRPC Driver — the gRPC counterpart for throughput-sensitive server-to-server work.
- The package README for the full method list and every failure mode of the methods above,

including `db.ts`, `db.grafana`, `db.promql` and `db.vector`.

4.18. Python gRPC Driver

`arcadedb-driver-grpc` is ArcadeDB's Python gRPC client, generated from the server's Protobuf contract. It wraps the generated stub in a small facade for streaming queries, streaming inserts and transactions, plus a matching `asyncio` facade for async applications.

4.18.1. Install

```
pip install arcadedb-driver-grpc
```

Or with uv:

```
uv add arcadedb-driver-grpc
```

Requires Python 3.10 or later.

The gRPC server is a plugin, and it is not started by default.

Register it in the server's plugin list before connecting:

```
-Darcadedb.server.plugins
=GRPC:com.arcadedb.server.grpc.GrpcServerPlugin
```

Once registered, it listens on port `50051`. See the gRPC API reference for the full option list, including TLS and message-size settings.

4.18.2. Connect and query

```
from arcadedb_driver_grpc import create_client, messages, password_auth

with create_client(
    "localhost:50051",
    insecure=True,
    auth=password_auth("root", "playwithdata", "mydb"),
) as client:
    response = client.raw.ExecuteQuery(
        messages.ExecuteQueryRequest(database="mydb", query="SELECT FROM Person WHERE
age > 21", language="sql")
    )
    for result in response.results:
        for record in result.records:
            print(record.rid, dict(record.properties))
```

```
#1:0 {'age': int32_value: 30
, '@type': string_value: "Person"
, '@rid': link_value {
  rid: "#1:0"
}
logical_type: "rid"
, '@cat': string_value: "v"
, 'name': string_value: "Alice"
}
#1:1 {'age': int32_value: 25
, '@type': string_value: "Person"
, '@rid': link_value {
  rid: "#1:1"
}
logical_type: "rid"
, '@cat': string_value: "v"
, 'name': string_value: "Bob"
}
```

The target is gRPC's native `host:port` form, not a URL — there is no scheme to parse and nothing to default. That is why TLS and plaintext are both explicit: pass `credentials=grpc.ssl_channel_credentials()` for TLS, or `insecure=True` to opt into plaintext.

The output above is not a copy-paste mistake — it is exactly what `dict(record.properties)` prints. See below for why, and how to get plain Python values instead.

The unary `ExecuteQuery` has a row ceiling, and it fails rather than truncates.

`ExecuteQuery` builds its whole answer into one gRPC message, so the server caps it: `server.grpcQueryMaxResultRows` (100000 by default) is a hard ceiling, and a result that would exceed it fails the call with `RESOURCE_EXHAUSTED` instead of coming back short. Unlike the HTTP drivers there is no `truncated` flag to check — a capped query does not return at all.

Streaming is the answer for result sets that large: `stream_query` emits incrementally and is bounded instead by `server.grpcStreamMaxMaterializedRows` (and only in `MATERIALIZE_ALL` mode) and by `server.grpcStreamWriteTimeoutMs`. See Settings for all three, and the gRPC API reference for the service itself.

4.18.3. `raw`, `raw_admin` and the wrappers

`client.raw` is the generated stub for `ArcadeDbService`, the contract's data plane, and all 21 of its RPCs are reachable through it exactly as shown above with `ExecuteQuery`. `create_client` adds five wrappers on top of `raw` — `stream_query`, `insert_stream`, `time_series_query`, `time_series_write_stream` and `transaction` — which between them cover seven of those 21: `StreamQuery`, `InsertStream`, `TimeSeriesQuery`, `TimeSeriesWriteStream`, and `BeginTransaction`/`CommitTransaction`/`RollbackTransaction`. These are the RPCs the generated stub alone handles badly — consuming a server-side stream one message at a time, feeding a client-side stream from an iterable, and pairing

begin, commit and rollback into a context manager. The remaining fourteen — `ExecuteQuery`, `ExecuteCommand`, `CreateRecord`, `UpdateRecord`, `DeleteRecord`, `LookupByRid`, `VectorSearch`, `HybridSearch`, `FullTextSearch`, `BulkInsert`, `InsertBidirectional`, `GraphBatchLoad`, `TimeSeriesWrite` and `TimeSeriesLatest` — are used directly through `raw`.

Administration is a second service, `ArcadeDbAdminService`, the control plane. It carries 46 RPCs (database lifecycle, users, groups, API tokens, settings, backups, the profiler, cluster and shutdown operations, and the `Health` and `Ready` probes), and `raw` does not reach it: `client.raw.GetServerInfo` raises `AttributeError`. Use `client.raw_admin` instead. Reading it raises `InsecureChannelError` unless the client was created with `credentials=` or `insecure=True`, because 44 of the 46 RPCs take their credentials in the request message, not from the channel, so the `auth=` argument does nothing for them:

```
from arcadedb_driver_grpc import create_client, messages

with create_client("localhost:50051", insecure=True) as client:
    response = client.raw_admin.ListDatabases(
        messages.ListDatabasesRequest(
            credentials=messages.DatabaseCredentials(username="root", password
="playwithdata")
        )
    )
    print(list(response.databases))
```

```
['mydb']
```

`raw` also means every value that comes back is exactly what the wire sent: a generated protobuf message, not a Python scalar. The `rid` and `type` fields of a `GrpcRecord` are already plain strings, but each entry in its `properties` map is a `GrpcValue` — a protobuf message with a `oneof` field named `kind` holding the actual value under one of `int32_value`, `string_value`, `link_value`, and so on. `dict(record.properties)` therefore gives you a dict of `GrpcValue` objects, not a dict of names, ages and RIDs. Unwrap each one by checking which field of the `oneof` is set:

```
def unwrap(value):
    field = value.WhichOneof("kind")
    if field == "link_value":
        return value.link_value.rid
    return getattr(value, field) if field else None

props = {k: unwrap(v) for k, v in record.properties.items()}
print(record.rid, props)
```

```
#1:0 {'age': 30, 'name': 'Alice', '@type': 'Person', '@rid': '#1:0', '@cat': 'v'}
#1:1 {'age': 25, 'name': 'Bob', '@type': 'Person', '@rid': '#1:1', '@cat': 'v'}
```

`link_value` is the one case that does not unwrap to a scalar directly — it is itself a message (`GrpcLink`), so reach into `.rid` for the record ID string. There is no library function that does this unwrapping for you; it is a plain `oneof` read, shown above for reference.

4.18.4. Authentication is attached to the channel

`bearer_auth(token)` and `password_auth(user, password, database=None)` do not attach credentials to individual calls. `create_client` turns whichever one you pass into a gRPC channel interceptor, so it runs on every call made through that channel — `client.raw.ExecuteCommand(…)` carries the same authentication headers that `client.stream_query(…)` does, with no extra work at the call site.

This matters because the facade's wrappers cover only seven of the data plane's 21 RPCs. If authentication were attached per-call instead — say, only on the methods `create_client` itself wraps — every one of the other fourteen RPCs reached through `raw` would go out silently anonymous. Attaching auth to the channel instead of the call means there is no RPC you can reach through this client without it.

Your own per-call headers are a separate thing. Every facade method, on the client and on a transaction handle alike, takes keyword-only `timeout=` and `metadata=` (a sequence of `(key, value)` pairs). `metadata` is appended to what the channel's auth interceptor adds, never a replacement for it.

The insecure-channel guard, and its limit.

`password_auth` sends the password as plaintext call metadata. To keep that from happening by accident over an unencrypted channel, `create_client` raises `InsecureChannelError` unless you either pass channel credentials or explicitly opt in with `insecure=True`:

```
from arcadedb_driver_grpc import InsecureChannelError, create_client,
password_auth

try:
    create_client("localhost:50051", auth=password_auth("root",
"playwithdata", "mydb"))
except InsecureChannelError as err:
    print("guard fired:", err)
```

```
guard fired: create_client: refusing to send a plaintext password over
an insecure channel to "localhost:50051". Pass
credentials=grpc.ssl_channel_credentials(), switch to bearer_auth, or
pass insecure=True to opt in explicitly.
```

The check is honest about what it can and cannot see: it keys on whether channel credentials were supplied, not on whether the channel is actually encrypted end-to-end. A `bearer_auth` token never trips it, at any security level, because a bearer token is not a password — the guard is specifically about `password_auth` sending a plaintext secret.

4.18.5. Streaming queries

```
for record in client.stream_query(
    messages.StreamQueryRequest(
        database="mydb",
        query="SELECT FROM Person",
        language="sql",
        retrieval_mode=messages.StreamQueryRequest.RetrievalMode.CURSOR,
        batch_size=500,
    )
):
    print(record.rid, dict(record.properties))
```

```
#1:0 {'age': int32_value: 30
, '@cat': string_value: "v"
, 'name': string_value: "Alice"
}
#1:1 {'age': int32_value: 25
, '@cat': string_value: "v"
, 'name': string_value: "Bob"
}
```

`stream_query` flattens the server's stream of `QueryResult` batches into one `GrpcRecord` at a time and does nothing else — it picks no default for `retrieval_mode` or `batch_size`. There are three retrieval modes:

- `CURSOR` — the proto default (value `0`). The server runs the query once and streams rows as you iterate.
- `MATERIALIZE_ALL` — the server runs the query to completion first, then batches the already-materialized result set back to you.
- `PAGED` — same as `CURSOR` *(since v26.12.1)*: the query runs once and its rows are streamed in batches. Earlier versions re-ran a SQL query with `LIMIT`/`SKIP` for every batch, which made a large stream slow.

Leaving both `retrieval_mode` and `batch_size` unset does not pick `CURSOR` and a sensible batch size as a considered choice — it sends whatever protobuf's zero values happen to be for those fields (`0`, which is `CURSOR`, and `0` rows per batch). Set both explicitly.

4.18.6. Streaming inserts

`insert_stream` owns the envelope bookkeeping the client-streaming `InsertStream` RPC needs — one session id for the stream, an incrementing chunk sequence, `database` on the first chunk, `last` on the final one — and leaves how rows are batched to you.

`database` goes on the first chunk only, and older servers insert nothing.

`insert_stream` sends `database` on the first chunk only, as the `.proto` specifies, and

> passes your options through unchanged: it never sets `options.database`. Older servers read the database from `InsertOptions.database` alone (ArcadeData/arcadedb#6597), so against one the stream reports its rows as received, inserts none, and succeeds. Drivers up to 0.2.0 copied `database` into the options to cover those servers; 0.3.0 dropped the copy. Check `inserted`, not just that the call returned, and set `options.database` yourself if you must use such a server.

To make the inserted rows part of a transaction, call `tx.insert_stream` on the transaction handle below: they then commit or roll back with it.

4.18.7. Transactions

```
from arcadedb_driver_grpc import create_client, messages, password_auth

with create_client(
    "localhost:50051",
    insecure=True,
    auth=password_auth("root", "playwithdata", "mydb"),
) as client:
    with client.transaction("mydb") as tx:
        tx.execute_command(
            messages.ExecuteCommandRequest(
                command="INSERT INTO Account SET balance = 100, owner = 'grpc-demo'",
                language="sql",
            )
        )
        response = tx.execute_query(
            messages.ExecuteQueryRequest(
                query="SELECT sum(balance) AS total FROM Account WHERE owner = 'grpc-
demo'",
                language="sql",
            )
        )
    print("total:", response.results[0].records[0].properties["total"].int32_value)
```

```
total: 100
```

`client.transaction(database)` is a context manager. `enter` issues `BeginTransaction` and hands back a `TransactionHandle`; the block commits on a clean exit and rolls back on an exception, which then propagates. A rollback that fails on that path is attached as the original exception's `cause` rather than replacing it, and a commit that fails triggers a best-effort rollback first so the server-side session is not left for the reaper.

The handle is how you run work inside the transaction. It exposes `execute_query`, `execute_command`, `create_record`, `update_record`, `delete_record`, `lookup_by_rid`, `vector_search`, `hybrid_search`, `full_text_search`, `time_series_query`, `time_series_latest`, `stream_query` and `insert_stream`, each of them binding the transaction's id and database onto a copy of the request you pass — so you never

set `transaction` or `database` yourself, and a request object you reuse elsewhere is left untouched. Rows sent through `tx.insert_stream` commit with the transaction and are discarded if the block raises:

```
from arcadedb_driver_grpc import InsertStreamRequest, messages

# batches() yields lists of messages.GrpcRecord, one list per wire chunk
with client.transaction("mydb") as tx:
    summary = tx.insert_stream(
        InsertStreamRequest(
            database="mydb",
            options=messages.InsertOptions(target_class="Person"),
            chunks=batches(),
        )
    )
```

Older servers ran an insert stream outside any transaction (ArcadeData/arcadedb#6607), so its rows survived a rollback. `BulkInsert`, `InsertBidirectional`, `GraphBatchLoad` and `TimeSeriesWrite` are not on the handle; they stay reachable through `client.raw` only, outside any transaction.

The trap is the same one the HTTP driver has: calls made through the outer `client` while a transaction is open do not join it. Only calls through the handle take part.

To choose the isolation level, pass `isolation_level="READ_COMMITTED"` or `isolation_level="REPEATABLE_READ"` to `client.transaction(…)` (driver 0.4.0 and later, against ArcadeDB 26.11.1 or later). Any other value raises `ValueError` before anything is sent. Without it the server begins the transaction at `READ_COMMITTED`.

4.18.8. Errors

Every RPC raises `grpc.RpcError` on failure — the driver adds no exception type of its own. `err.code()` is a `grpc.StatusCode` and `err.details()` is the server's message:

```
import grpc

from arcadedb_driver_grpc import create_client, messages, password_auth

with create_client(
    "localhost:50051",
    insecure=True,
    auth=password_auth("root", "playwithdata", "mydb"),
) as client:
    try:
        client.raw.ExecuteQuery(
            messages.ExecuteQueryRequest(database="mydb", query="SELECT FROM
NoSuchType", language="sql")
        )
    except grpc.RpcError as err:
```

```
        print(err.code(), err.details())
```

```
StatusCode.NOT_FOUND Type with name 'NoSuchType' was not found
```

The status you are most likely to meet first is `UNAUTHENTICATED`. A stock ArcadeDB gRPC server refuses anonymous calls, so a client built without `auth=` connects happily and then fails on the first RPC:

```
import grpc

from arcadedb_driver_grpc import create_client, messages

with create_client("localhost:50051", insecure=True) as anonymous:
    try:
        anonymous.raw.ExecuteQuery(
            messages.ExecuteQueryRequest(database="mydb", query="SELECT FROM Person",
language="sql")
        )
    except grpc.RpcError as err:
        print(err.code() is grpc.StatusCode.UNAUTHENTICATED, err.details())
```

```
True Authentication required
```

`RESOURCE_EXHAUSTED` is the other status worth branching on: it is what a result set larger than the server's row cap returns, as described above.

4.18.9. Async

Importing `create_client` from `arcadedb_driver_grpc.aio` instead of `arcadedb_driver_grpc` is the whole signal for which client you get: the async client mirrors the sync one method for method, `await`-ed, with the same `insecure` and `auth` arguments.

```
import asyncio

from arcadedb_driver_grpc import messages, password_auth
from arcadedb_driver_grpc.aio import create_client

async def main() -> None:
    async with create_client(
        "localhost:50051",
        insecure=True,
        auth=password_auth("root", "playwithdata", "mydb"),
    ) as client:
        response = await client.raw.ExecuteQuery(
            messages.ExecuteQueryRequest(database="mydb", query="SELECT FROM Person",
language="sql")
```

```
        )
        for result in response.results:
            for record in result.records:
                print(record.rid)

asyncio.run(main())
```

```
#1:0
#1:1
```

4.18.10. Next steps

- Choosing a protocol from Python — measured HTTP, Bolt and gRPC timings for large results.
- Native Drivers — how this driver compares to the others, and when to reach for gRPC instead of HTTP.
- Python — HTTP Driver — the HTTP counterpart for general application traffic.
- The package README for the internals, including why the async facade needs four interceptor classes — one per gRPC call shape (unary-unary, unary-stream, stream-unary, stream-stream) — where the sync facade needs only one.

4.19. ArcadeDB Python Quickstart

Connection options for Python

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **Native HTTP driver** — `pip install arcadedb-driver`, generated from ArcadeDB's OpenAPI contract. Start here for most applications.
- **Native gRPC driver** — `pip install arcadedb-driver-grpc`, for streaming and bulk inserts.
- **Neo4j BOLT** — `pip install neo4j`, run Cypher queries.
- **HTTP/JSON API** — use `requests` or `httpx`, no driver required.
- **Embedded bindings**: `pip install arcadedb`, runs the engine inside your Python process with no server. A JVM build, `arcadedb-jvm`, is the alternative for heavy, long-running loads.

This tutorial uses the PostgreSQL wire protocol, which is the right choice when you already run a PostgreSQL stack.

This tutorial shows how to connect to ArcadeDB from Python using the PostgreSQL wire protocol. You will create a graph, run queries, and use vector search — all from a Python script.

4.19.1. Prerequisites

- ArcadeDB running with PostgreSQL protocol enabled (default port 5432)
- Python 3.10+
- `psycopg` library

```
pip install "psycopg[binary]>=3.1,<4"
```

4.19.2. Connect to ArcadeDB

ArcadeDB speaks the PostgreSQL wire protocol, so you can use standard PostgreSQL drivers. No special SDK is needed.

```
import psycopg

conn = psycopg.connect(
    host='localhost',
    port=5432,
    dbname='mydb',
    user='root',
    password='arcadedb',
    autocommit=True,
)
print('Connected to ArcadeDB')
```

4.19.3. Create a Schema

Use SQL to create vertex and edge types:

```
with conn.cursor() as cur:
    # Create vertex types
    cur.execute("CREATE VERTEX TYPE Person IF NOT EXISTS")
    cur.execute("CREATE VERTEX TYPE Movie IF NOT EXISTS")

    # Create edge type
    cur.execute("CREATE EDGE TYPE Acted IF NOT EXISTS")

    # Insert data
    cur.execute("CREATE VERTEX Person SET name = 'Alice', age = 30")
    cur.execute("CREATE VERTEX Person SET name = 'Bob', age = 25")
    cur.execute("CREATE VERTEX Movie SET title = 'The Matrix', year = 1999")
    cur.execute("""
        CREATE EDGE Acted FROM (SELECT FROM Person WHERE name = 'Alice')
                          TO (SELECT FROM Movie WHERE title = 'The Matrix')
                          SET role = 'Trinity'
    """)
```

4.19.4. Query with SQL

```
with conn.cursor() as cur:
    cur.execute("""
        MATCH {type: Person, as: person}.outE('Acted'){as: acted}.inV(){type: Movie,
as: movie}
        RETURN person.name AS name, movie.title AS title, acted.role AS role
    """)
    for row in cur.fetchall():
        print(f'{row[0]} acted in {row[1]} as {row[2]}')
```

4.19.5. Query with Cypher

ArcadeDB supports Cypher queries through the same connection:

```
with conn.cursor() as cur:
    cur.execute("{cypher} MATCH (p:Person)-[a:Acted]->(m:Movie) RETURN p.name,
m.title, a.role")
    for row in cur.fetchall():
        print(f'{row[0]} acted in {row[1]} as {row[2]}')
```

Prefix Cypher queries with `{cypher}` when using the PostgreSQL protocol.

4.19.6. Vector Search

ArcadeDB supports vector similarity search. Add embeddings to your data and query by similarity:

```
with conn.cursor() as cur:
    # Create a type with a vector property
    cur.execute("CREATE VERTEX TYPE Product IF NOT EXISTS")
    cur.execute("CREATE PROPERTY Product.embedding IF NOT EXISTS LIST OF FLOAT")
    cur.execute("CREATE INDEX IF NOT EXISTS ON Product (embedding) LSM_VECTOR METADATA
{dimensions: 4, similarity: 'COSINE'}")

    # Insert products with embeddings
    cur.execute("CREATE VERTEX Product SET name = 'Laptop', embedding = [0.9, 0.1,
0.8, 0.2]")
    cur.execute("CREATE VERTEX Product SET name = 'Tablet', embedding = [0.8, 0.2,
0.7, 0.3]")
    cur.execute("CREATE VERTEX Product SET name = 'Phone',  embedding = [0.7, 0.3,
0.6, 0.4]")
    cur.execute("CREATE VERTEX Product SET name = 'Book',   embedding = [0.1, 0.9,
0.2, 0.8]")

    # Find the 3 most similar products to a query vector
    cur.execute("""
        SELECT name, distance FROM (
```

```
            SELECT expand(vector.neighbors('Product[embedding]', [0.85, 0.15, 0.75,
0.25], 3))
        )
    """)
    print('Similar products:')
    for row in cur.fetchall():
        print(f'  {row[0]}')
```

4.19.7. Full Example

Here is a complete script that connects, creates data, and queries:

```
#!/usr/bin/env python3
"""ArcadeDB Python quickstart -- graph + vector search."""

import psycopg

def main():
    conn = psycopg.connect(
        host='localhost', port=5432, dbname='mydb',
        user='root', password='arcadedb', autocommit=True,
    )

    with conn.cursor() as cur:
        # Create schema
        cur.execute("CREATE VERTEX TYPE Person IF NOT EXISTS")
        cur.execute("CREATE VERTEX TYPE Movie IF NOT EXISTS")
        cur.execute("CREATE EDGE TYPE Acted IF NOT EXISTS")

        # Insert data
        cur.execute("CREATE VERTEX Person SET name = 'Alice', age = 30")
        cur.execute("CREATE VERTEX Movie SET title = 'The Matrix', year = 1999")
        cur.execute("""
            CREATE EDGE Acted FROM (SELECT FROM Person WHERE name = 'Alice')
                              TO (SELECT FROM Movie WHERE title = 'The Matrix')
                              SET role = 'Trinity'
        """)

        # Query with SQL MATCH
        cur.execute("""
            MATCH {type: Person, as: person}.outE('Acted'){as: acted}.inV(){type:
Movie, as: movie}
            RETURN person.name AS name, movie.title AS title, acted.role AS role
        """)
        for row in cur.fetchall():
            print(f'{row[0]} acted in {row[1]} as {row[2]}')

    conn.close()

if __name__ == '__main__':
```

```
    main()
```

4.19.8. HTTP / JSON API

For lightweight scripts or serverless environments where you want minimal dependencies, talk to ArcadeDB's HTTP/JSON API directly with `requests` or `httpx` — no driver needed.

```python
import requests

base_url = 'http://localhost:2480/api/v1'
auth = ('root', 'arcadedb')

# Run a SQL query
response = requests.post(
    f'{base_url}/command/mydb',
    json={'language': 'sql', 'command': 'SELECT FROM V LIMIT 10'},
    auth=auth,
)
print(response.json())

# Run a Cypher query
response = requests.post(
    f'{base_url}/command/mydb',
    json={'language': 'cypher', 'command': 'MATCH (n) RETURN n LIMIT 10'},
    auth=auth,
)
print(response.json())
```

See the HTTP/JSON API reference for every endpoint and parameter.

4.19.9. Embedded Python Bindings

For use cases that need in-process access (no network overhead), see Python — Embedded Bindings. They run the engine inside your Python process.

4.19.10. Next Steps

- IAM Use Case — Full Python implementation with 7 query patterns
- PostgreSQL Protocol Reference
- SQL Reference
- Cypher Reference

4.20. JavaScript / TypeScript HTTP Driver

`@arcadedb/driver` is ArcadeDB's TypeScript/JavaScript HTTP client, generated from the server's OpenAPI contract.

Node.js 20+, ESM only.

The package requires Node.js 20 or later, and it ships **ESM only** — there is no CommonJS build and no `require()` entry point. Import it with `import`, not `require`.

4.20.1. Install

```
npm install @arcadedb/driver
```

4.20.2. Connect and query

```
import { ArcadeDBError, basicAuth, createClient } from "@arcadedb/driver";

const server = createClient({
  baseUrl: "http://localhost:2480",
  auth: basicAuth("root", "playwithdata"),
});

const db = server.db("mydb");

const envelope = await db.query({
  language: "sql",
  command: "SELECT FROM Person WHERE age > ?",
  params: { 0: 21 },
});
console.log(envelope.result, envelope.returned, envelope.truncated);
```

```
[
  {
    '@rid': '#1:0',
    '@type': 'Person',
    '@cat': 'v',
    name: 'Alice',
    age: 30
  },
  {
    '@rid': '#1:1',
    '@type': 'Person',
    '@cat': 'v',
    name: 'Bob',
    age: 25
  }
] 2 false
```

Positional parameters are zero-indexed — `{ 1: 21 }` silently returns nothing.

Positional parameters are keyed by their zero-based index. For the single placeholder above, the key is `0`, not `1`. Passing `params: { 1: 21 }` does not raise an error and does not match the wrong rows — it runs the query with no bound value for `?` at all and comes back with an empty `result` (`returned: 0`, `truncated: false`). The driver's own README shows `{ 1: 21 }`; it looks plausible and returns quietly, which is exactly what makes it easy to ship. Always start positional parameter keys at `0`.

There is no timeout option, and `fetch` has no default timeout.

`createClient` accepts exactly four options — `baseUrl`, `auth`, `fetch` and `WebSocket` (used only by the insert session described below). There is no `timeout`, and the client calls the runtime's global `fetch`, which waits forever by default. A request against a stalled connection therefore hangs indefinitely unless you bound it yourself. The `fetch` option is where to do that: pass a wrapper that attaches an `AbortSignal`.

```
import { basicAuth, createClient } from "@arcadedb/driver";

const server = createClient({
  baseUrl: "http://localhost:2480",
  auth: basicAuth("root", "playwithdata"),
  fetch: (input, init) => fetch(input, { ...init, signal: AbortSignal
.timeout(5000) }),
});

const envelope = await server.db("mydb").query({ language: "sql",
command: "SELECT FROM Person" });
console.log(envelope.returned);
```

```
2
```

The Python driver takes an explicit `timeout` argument instead; see the Python HTTP driver page, where the same "no timeout unless you ask" default applies.

`basicAuth("root", "playwithdata")` builds the `Authorization` header for HTTP basic auth; `bearerAuth("AU-…")` substitutes directly for it wherever you already hold a session token or other bearer credential; see Sessions and API tokens for getting one.

4.20.3. The result envelope

`query` and `command` do not return bare rows. They return the whole response envelope:

```
interface QueryEnvelope<T> {
  result: T[];
  limit: number;
```

```
  returned: number;
  truncated: boolean;
  explain?: string;                       // EXPLAIN / PROFILE only
  explainPlan?: Record<string, unknown>;  // present exactly when `explain` is
}
```

See Native Drivers for why `truncated` matters and what to do when it is `true`.

An `EXPLAIN` or `PROFILE` statement answers with its execution plan instead of rows: `explain` holds it as indented text and `explainPlan` as the same plan in structured form, while `result` is empty and `returned` is `0`. An empty `result` there is not "no rows matched".

4.20.4. Transactions

```
const total = await db.transaction(async (tx) => {
  await tx.command({ language: "sql", command: "INSERT INTO Account SET balance = 100"
});
  const { result } = await tx.query({ language: "sql", command: "SELECT sum(balance)
AS total FROM Account" });
  return result[0].total;
});
console.log("total:", total);
```

```
total: 200
```

(The total above reflects balances already present in this database from earlier examples on this page's server; a fresh `Account` type would show `100`.)

Every call inside the callback must go through the `tx` handle it receives, not the outer `db` — calls made through `db` while a transaction is open auto-commit individually, outside it. The contract:

- `transaction` commits when the callback resolves, and returns the callback's own return value.
- It rolls back and re-throws when the callback throws or rejects.
- The caller always sees the callback's own error. If the rollback itself also fails, that failure is attached to the error as `err.cause` rather than replacing it.
- If the commit itself fails, `transaction` issues a best-effort rollback first, to release the server-side session, before re-throwing the commit's error.

4.20.5. Two error models

Facade methods — `query`, `command`, `transaction`, and others — throw `ArcadeDBError` on any non-2xx response:

```
try {
  await db.query({ language: "sql", command: "SELECT FROM NoSuchType" });
```

```
} catch (err) {
  if (err instanceof ArcadeDBError) console.error(err.status, err.error, err.detail,
err.requestId);
}
```

```
500 Internal error Type with name 'NoSuchType' was not found 1a9b928039f2a6cd-30
```

`requestId` is the server's request identifier for that call and differs on every request; use it to correlate a failure with the server log. Two more fields come from the response's headers: `retryAfter`, the seconds the server asked you to wait before retrying (on a 503, or a 409 for an identical request still in flight), and `commitIndex`, the Raft index to send back as `X-ArcadeDB-Read-After` on a later read. Both are `undefined` when the server sent none, and `0` is a valid `retryAfter`, so compare against `undefined`.

A 409 has two meanings. It is either "an identical request is still in flight", which is retryable after `retryAfter`, or "query terminated", because an operator stopped the statement and its writes were rolled back. Retrying the second undoes the operator's decision. Tell them apart with `isQueryTerminated(err)`, never by `status === 409`.

A rejected time-series write throws `TimeSeriesWriteError`, a subclass of `ArcadeDBError` that says how many samples were `written` and `dropped` and why. A write is not atomic: the `written` samples landed, so re-sending the whole batch writes them twice.

`server.raw` — the underlying openapi-fetch client beneath the facade — takes the opposite approach: it never throws. It returns `{ data, error }` and leaves handling the error to the caller. Use the facade for the ergonomics of try/catch; use `raw` when you would rather branch on `{ data, error }` without exceptions.

Beyond `query`, `command` and `transaction`, a database handle also exposes `db.ts`, `db.grafana` and `db.promql` for time-series ingestion, Grafana panel queries and a PromQL-compatible query surface (including `db.promql.labelValues`), and `db.vector` for vector, hybrid and full-text search — see the package README linked below for those.

4.20.6. Labelling work with a query tag

`query`, `command`, `queryStream`, `commandStream`, `batchLoad` and `batchLoadStream` take an optional `queryTag`, and `transaction` takes one as its second argument. The tag travels as the `X-ArcadeDB-Query-Tag` header, so you can list and terminate the work by it:

```
await db.query({ language: "sql", command: "SELECT FROM Orders", queryTag: "nightly-
report" });

await db.transaction(
  async (tx) => {
    await tx.command({ language: "sql", command: "UPDATE Orders SET archived = true"
});
```

```
  },
  { queryTag: "archive-run" },
);
```

A transaction's tag is sent on the begin request only; the server applies it to every statement in the transaction that carries no tag of its own.

4.20.7. Duplex insert session

`db.insertSession()` opens an insert session on the server's `/ws` endpoint. Unlike `batchLoad`, whose commit policy is fixed before the load starts, a session lets you read each chunk's acknowledgement before deciding what to send next, and whether to commit at all:

```
const session = await db.insertSession({ targetType: "Person" });
try {
  const ack = await session.sendChunk([{ name: "a" }, { name: "b" }]);
  console.log(ack.inserted, ack.failed, ack.errors); // nothing is durable yet
  const done = await session.commit();               // or: await session.rollback()
  console.log(done.outcome, done.summary.inserted);
} finally {
  await session.close();                             // rolls back if still open
}
```

It uses the runtime's global `WebSocket` (Node 22+); on Node 20, pass the `ws` package's class as `createClient`'s `WebSocket` option.

4.20.8. Administration

`createDatabase`, `dropDatabase`, `createUser`, `updateUser`, `dropUser`, `listUsers`, `saveGroup`, `deleteGroup` and `listGroups` are on the server handle, and `serverCommand` runs any other server command. All of them are root-only:

```
await server.createDatabase("tenant_a");
await server.createUser("alice", "a-long-password", { databases: { tenant_a: ["admin"]
} });
const users = await server.listUsers();
await server.dropUser("alice");
await server.dropDatabase("tenant_a");
```

A user or database that already exists is a 409 on ArcadeDB 26.10.2 and later. Older servers answer 403 for a duplicate user and 400 for a duplicate database.

4.20.9. Sessions and API tokens

```
// Log in as each user once, then keep a client that holds only that user's token.
const login = await server.loginAs("alice", "a-long-password"); // or server.login()
```

```
const alice = server.withToken(login.token);                    // server is unchanged
await alice.listDatabases();                                    // only what alice may
see
await alice.logout();

// A long-lived machine credential (root only).
const created = await server.createAPIToken("ci", { database: "tenant_a" });
store(created.token);                           // the only time the plaintext exists
await server.deleteAPIToken(created.tokenHash); // by hash, never by token
```

`listSessions` and `listAPITokens` list what exists. Each user holds at most 100 sessions, and a further login silently evicts the oldest, so log in once per user and reuse the token rather than logging in per request. Wrong Basic credentials, like an invalid, expired or logged-out token, are answered 401; 403 is what an authenticated user gets for an operation it is not allowed (*since v26.11.1*: before, wrong Basic credentials were answered 403).

4.20.10. Running work

```
import { isQueryTerminated } from "@arcadedb/driver";

const running = await server.listQueries({ tag: "nightly-report" }); // or { database
}
const results = await server.terminateQueriesByTag("nightly-report", { waitMs: 3000
});
await server.terminateQuery(running[0].id);
const open = await server.listTransactions();
await server.terminateTransaction(open[0].id);
const progress = await db.progress(); // long-running maintenance operations
```

Any authenticated user may call these: root sees and stops every statement, another user only their own. Each terminate answer carries a `status`, and which one you get decides whether you have anything to undo:

`status`	**Meaning**
`"terminated"`	It stopped, and what it wrote was rolled back.
`"completed"`	It finished before it noticed, so what it did stands.
`"terminating"`	It had not stopped when the wait ended; the request's own outcome is the answer.
`"not found"`	No such statement on this server, or another user's.

- **Each server sees only its own work.** These commands are not forwarded to the cluster leader, so behind a load balancer the server you reach may not be the one running the statement.
- **A commit after `terminateTransaction` fails on ArcadeDB 26.11.1 and later.** The transaction's writes are rolled back, and the server answers its owner's

commit with 404. Earlier servers answered 204, as if it had committed (ArcadeData/arcadedb#10164).

4.20.11. Next steps

- Native Drivers — the result envelope and why `truncated` matters, and how this driver compares to the others.
- JavaScript / TypeScript — gRPC Driver — the gRPC counterpart for throughput-sensitive server-to-server work.
- The package README for the full method list and every failure mode of the methods above, including `db.ts`, `db.grafana`, `db.promql` and `db.vector`.

4.21. JavaScript / TypeScript gRPC Driver

`@arcadedb/driver-grpc` is ArcadeDB's TypeScript/JavaScript gRPC client, generated from the server's Protobuf contract. It wraps the generated Connect client in a small facade for streaming queries, streaming inserts and transactions.

4.21.1. Install

```
npm install @arcadedb/driver-grpc
```

The gRPC server is a plugin, and it is not started by default.

Register it in the server's plugin list before connecting:

```
-Darcadedb.server.plugins
=GRPC:com.arcadedb.server.grpc.GrpcServerPlugin
```

Once registered, it listens on port `50051`. See the gRPC API reference for the full option list, including TLS and message-size settings.

4.21.2. Runtime targets: Node, Bun, Deno — not browsers

The package requires Node.js 20 or later, and it ships **ESM only** — there is no CommonJS build and no `require()` entry point. Import it with `import`, not `require`.

Bun and Deno are intended targets for this driver, but only Node is what the arcadedb-drivers repository's continuous integration actually exercises against a live server. Treat Bun and Deno as likely to work rather than as verified.

There is no browser build, and there cannot be one until the server changes. ArcadeDB's `GrpcServerPlugin` is plain grpc-java over HTTP/2, with no gRPC-Web handler and no Connect protocol in front of it, so a browser cannot speak this driver's wire format at all. Use the HTTP driver there.

4.21.3. Connect and query

```
import { createClient, passwordAuth } from "@arcadedb/driver-grpc";

const grpc = createClient({
  baseUrl: "http://localhost:50051",
  auth: passwordAuth("root", "playwithdata", "mydb"),
  insecure: true,
});

const response = await grpc.raw.executeQuery({
  database: "mydb",
  query: "SELECT FROM Person WHERE age > 21",
  language: "sql",
});
console.log(response.results.flatMap((r) => r.records.map((rec) => [rec.rid, rec
.properties])));
```

```
[
  [ '#1:0', { name: [Object], age: [Object], '@rid': [Object], '@type': [Object],
'@cat': [Object] } ],
  [ '#1:1', { name: [Object], age: [Object], '@rid': [Object], '@type': [Object],
'@cat': [Object] } ]
]
```

Node's default `console.log` collapses nested objects two levels deep, and each property value here is itself a nested message — that is the summarizing behavior of `util.inspect`, not evidence of anything wrong. See below for what those objects actually contain and how to unwrap them.

The unary `executeQuery` has a row ceiling, and it fails rather than truncates.

`ExecuteQuery` builds its whole answer into one gRPC message, so the server caps it: `server.grpcQueryMaxResultRows` (100000 by default) is a hard ceiling, and a result that would exceed it fails the call with `RESOURCE_EXHAUSTED` instead of coming back short. Unlike the HTTP drivers there is no `truncated` flag to check — a capped query does not return at all.

Streaming is the answer for result sets that large: `streamQuery` emits incrementally and is bounded instead by `server.grpcStreamMaxMaterializedRows` (and only in `MATERIALIZE_ALL` mode) and by `server.grpcStreamWriteTimeoutMs`. See Settings for all three, and the gRPC API reference for the service itself.

`raw` is the generated Connect client for `ArcadeDbService`, the same one you get from `createClient(ArcadeDbService, transport)` if you built the transport yourself — every RPC in the data-plane contract is callable through it exactly as shown above with `executeQuery`. `createClient` from this package adds `streamQuery`, `insertStream` and `transaction` on top, covering the RPCs the generated client alone handles awkwardly: consuming a server-side stream, feeding a client-side

stream, and pairing begin/commit/rollback into one call.

4.21.4. `record.properties` and the `GrpcValue` wrapper

Every value in a record's `properties` map comes back as a `GrpcValue` object, not a plain JavaScript scalar. Printed at full depth, the `name` and `age` fields above look like this:

```
name: {
  '$typeName': 'com.arcadedb.grpc.GrpcValue',
  kind: { case: 'stringValue', value: 'Alice' },
  logicalType: ''
}
age: {
  '$typeName': 'com.arcadedb.grpc.GrpcValue',
  kind: { case: 'int32Value', value: 30 },
  logicalType: ''
}
'@rid': {
  '$typeName': 'com.arcadedb.grpc.GrpcValue',
  kind: {
    case: 'linkValue',
    value: { '$typeName': 'com.arcadedb.grpc.GrpcLink', rid: '#1:0', type: '' }
  },
  logicalType: 'rid'
}
```

`GrpcValue.kind` is this package's Connect-ES rendering of the contract's `oneof` field: a discriminated union with `case` naming which alternative is set and `value` holding it directly — there is no per-type field name to guess, unlike the generated Python client's `int32_value`/`string_value`/`link_value` fields for the same `oneof`. Unwrap it by branching on `kind.case`:

```
function unwrap(value) {
  if (value.kind.case === "linkValue") return value.kind.value.rid;
  return value.kind.value;
}

for (const result of response.results) {
  for (const rec of result.records) {
    const props = Object.fromEntries(Object.entries(rec.properties).map(([k, v]) => [
k, unwrap(v)]));
    console.log(rec.rid, props);
  }
}
```

```
#1:0 { name: 'Alice', age: 30, '@rid': '#1:0', '@type': 'Person', '@cat': 'v' }
#1:1 { name: 'Bob', age: 25, '@rid': '#1:1', '@type': 'Person', '@cat': 'v' }
```

`linkValue` is the one case that does not unwrap to a scalar directly — its `value` is itself a `GrpcLink` message, so reach into `.rid` for the record ID string, as `unwrap` does above. There is no library function that performs this unwrapping; it is a plain discriminated-union read, shown here for reference.

4.21.5. Authentication and the insecure guard

`bearerAuth(token)` sets `authorization: Bearer <token>` on every outgoing call's metadata. `passwordAuth(user, password, database?)` sets `x-arcade-user`, `x-arcade-password` and (when given) `x-arcade-database` instead. Both are passed once, as the `auth` option to `createClient` — there is no per-call credential to attach separately.

The insecure-channel guard, and its limit.

`passwordAuth` sends the password as plaintext call metadata. `createClient` refuses to pair it with a non-TLS `http://` `baseUrl` unless you pass `insecure: true` explicitly:

```
import { createClient, passwordAuth } from "@arcadedb/driver-grpc";

try {
  createClient({ baseUrl: "http://localhost:50051", auth: passwordAuth
("root", "playwithdata") });
} catch (err) {
  console.log("guard fired:", err.message);
}
```

```
guard fired: createClient: refusing to send a plaintext password over
insecure baseUrl "http://localhost:50051". Use an https:// baseUrl,
switch to bearerAuth, or pass insecure: true to opt in explicitly.
```

This is why the connect example above needs `insecure: true` at all: pairing `passwordAuth` with an `http://` `baseUrl` is exactly what this guard exists to catch.

The check has a real limit, stated plainly: it recognizes only the exact interceptor value `passwordAuth` returns, marked internally with a symbol property. Wrap that interceptor in your own logging or retry interceptor and the value `createClient` sees is a new function without the marker — the refusal is silently skipped, and the plaintext password still goes out over `http://`. The guard protects the direct, undecorated case; it is not a general plaintext-secret scanner.

4.21.6. Streaming queries and `retrievalMode`

```
import { createClient, passwordAuth, StreamQueryRequest_RetrievalMode } from
"@arcadedb/driver-grpc";

const grpc = createClient({
```

```
  baseUrl: "http://localhost:50051",
  auth: passwordAuth("root", "playwithdata", "mydb"),
  insecure: true,
});

for await (const row of grpc.streamQuery({
  database: "mydb",
  query: "SELECT FROM Person",
  language: "sql",
  retrievalMode: StreamQueryRequest_RetrievalMode.CURSOR,
  batchSize: 500,
})) {
  console.log(row.rid, row.properties);
}
```

```
#1:0 {
  name: {
    '$typeName': 'com.arcadedb.grpc.GrpcValue',
    kind: { case: 'stringValue', value: 'Alice' },
    logicalType: ''
  },
  age: {
    '$typeName': 'com.arcadedb.grpc.GrpcValue',
    kind: { case: 'int32Value', value: 30 },
    logicalType: ''
  },
  '@cat': {
    '$typeName': 'com.arcadedb.grpc.GrpcValue',
    kind: { case: 'stringValue', value: 'v' },
    logicalType: ''
  }
}
#1:1 {
  name: {
    '$typeName': 'com.arcadedb.grpc.GrpcValue',
    kind: { case: 'stringValue', value: 'Bob' },
    logicalType: ''
  },
  age: {
    '$typeName': 'com.arcadedb.grpc.GrpcValue',
    kind: { case: 'int32Value', value: 25 },
    logicalType: ''
  },
  '@cat': {
    '$typeName': 'com.arcadedb.grpc.GrpcValue',
    kind: { case: 'stringValue', value: 'v' },
    logicalType: ''
  }
}
```

`streamQuery` flattens the server's stream of row batches into one record at a time and does nothing else — each `properties` entry is still a `GrpcValue` needing the same `unwrap` shown above. Both `retrievalMode` and `batchSize` are deliberately the caller's own choice; the wrapper picks no default for either. There are three retrieval modes:

- `CURSOR` — the proto default (value `0`). The server runs the query once and streams rows as you iterate.
- `MATERIALIZE_ALL` — the server runs the query to completion first, then batches the already-materialized result set back to you.
- `PAGED` — same as `CURSOR` *(since v26.12.1)*: the query runs once and its rows are streamed in batches. Earlier versions re-ran a SQL query with `LIMIT`/`SKIP` for every batch, which made a large stream slow.

Leaving both fields out does not pick `CURSOR` and a sensible batch size as a considered choice — it sends whatever protobuf's zero values happen to be for those fields (`0`, which is `CURSOR`, and `0` rows per batch). Set both explicitly, as above.

4.21.7. Streaming inserts

`insertStream` owns the envelope bookkeeping the client-streaming `InsertStream` RPC needs — one session id for the stream, an incrementing chunk sequence, `database` on the first chunk, `last` on the final one — and leaves how rows are batched to you.

`database` goes on the first chunk only, and older servers insert nothing.

`insertStream` sends `database` on the first chunk only, as the `.proto` specifies, and passes your options through unchanged: it never sets `options.database`. Older servers read the database from `InsertOptions.database` alone (ArcadeData/arcadedb#6597), so against one the stream reports its rows as received, inserts none, and succeeds. Drivers up to 0.2.0 copied `database` into the options to cover those servers; 0.3.0 dropped the copy. Check `inserted`, not just that the call returned, and set `options.database` yourself if you must use such a server.

To make the inserted rows part of a transaction, call `tx.insertStream` on the transaction handle below: they then commit or roll back with it.

4.21.8. Transactions

```
import { createClient, passwordAuth } from "@arcadedb/driver-grpc";

const grpc = createClient({
  baseUrl: "http://localhost:50051",
  auth: passwordAuth("root", "playwithdata", "mydb"),
  insecure: true,
});

const total = await grpc.transaction("mydb", async (tx) => {
  await tx.executeCommand({
```

```
    command: "INSERT INTO Account SET balance = 100, owner = 'grpc-demo-js'",
    language: "sql",
  });
  const response = await tx.executeQuery({
    query: "SELECT sum(balance) AS total FROM Account WHERE owner = 'grpc-demo-js'",
    language: "sql",
  });
  return response.results[0].records[0].properties.total.kind.value;
});
console.log("total:", total);
```

```
total: 100
```

`transaction(database, fn)` begins a server-side transaction, hands `fn` a `TransactionHandle`, and ends the transaction on both paths: it commits when `fn` resolves and returns the value `fn` returned, and rolls back and re-throws the original error when `fn` throws or rejects. A rollback that fails on that path is attached as `cause` on the original error rather than replacing it, and a failing commit issues a best-effort rollback first so the server-side session is released. A `commitTransaction` that resolves with `committed: false` — what the server answers for a transaction it already reaped — throws rather than reporting a success whose writes were lost.

The handle exposes `executeQuery`, `executeCommand`, `createRecord`, `updateRecord`, `deleteRecord`, `lookupByRid`, `streamQuery` and `insertStream`, each with the transaction's `database` and `transaction` context bound in automatically, so you never set either yourself. Rows sent through `tx.insertStream` commit when the callback resolves and are discarded when it throws:

```
// rows() yields arrays of records, one array per wire chunk
await grpc.transaction("mydb", async (tx) => {
  await tx.insertStream({ database: "mydb", options: { targetClass: "Person" },
chunks: rows() });
  // throwing here rolls the streamed rows back with everything else
});
```

Older servers ran an insert stream outside any transaction (ArcadeData/arcadedb#6607), so its rows survived a rollback. `bulkInsert` and `graphBatchLoad` are not on the handle; they stay reachable through `grpc.raw` only, outside any transaction.

The trap is the same one the HTTP driver has: calls made through the outer client while a transaction is open do not join it. Only calls through the handle take part.

4.21.9. Errors

Every RPC rejects with a Connect `ConnectError` on failure — this driver adds no error type of its own. `err.code` is a Connect `Code` and `err.rawMessage` is the server's message. Both come from `@connectrpc/connect`, which the driver depends on; add it to your own dependencies to import it directly.

```
import { Code, ConnectError } from "@connectrpc/connect";
import { createClient, passwordAuth } from "@arcadedb/driver-grpc";

const grpc = createClient({
  baseUrl: "http://localhost:50051",
  auth: passwordAuth("root", "playwithdata", "mydb"),
  insecure: true,
});

try {
  await grpc.raw.executeQuery({ database: "mydb", query: "SELECT FROM NoSuchType",
language: "sql" });
} catch (err) {
  if (err instanceof ConnectError) console.log(Code[err.code], err.rawMessage);
}
```

```
NotFound Type with name 'NoSuchType' was not found
```

The code you are most likely to meet first is `Unauthenticated`. A stock ArcadeDB gRPC server refuses anonymous calls, so a client built without `auth` is created happily and then fails on the first RPC:

```
import { Code, ConnectError } from "@connectrpc/connect";
import { createClient } from "@arcadedb/driver-grpc";

const anonymous = createClient({ baseUrl: "http://localhost:50051" });

try {
  await anonymous.raw.executeQuery({ database: "mydb", query: "SELECT FROM Person",
language: "sql" });
} catch (err) {
  if (err instanceof ConnectError) console.log(err.code === Code.Unauthenticated, err
.rawMessage);
}
```

```
true Authentication required
```

`ResourceExhausted` is the other code worth branching on: it is what a result set larger than the server's row cap returns, as described above.

4.21.10. Next steps

- Native Drivers — how this driver compares to the others, and when to reach for gRPC instead of HTTP.
- JavaScript / TypeScript — HTTP Driver — the HTTP counterpart, and the only option for browsers.

- The package README for the full method list, including `insertStream` and `transaction`.

4.22. JavaScript / TypeScript

Connection options for JavaScript / TypeScript

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **Native HTTP driver** — `npm install @arcadedb/driver`, generated from ArcadeDB's OpenAPI contract. Start here for most applications.
- **Native gRPC driver** — `npm install @arcadedb/driver-grpc`, for streaming and bulk inserts.
- **Neo4j BOLT** — `npm install neo4j-driver`, run Cypher queries.
- **HTTP/JSON API** — use `fetch` or `axios`, no driver required.

This tutorial uses the PostgreSQL wire protocol, which is the right choice when you already run a PostgreSQL stack.

This tutorial shows how to connect to ArcadeDB from Node.js using the PostgreSQL wire protocol. You will create a graph, run queries, and use vector search — all from JavaScript.

4.22.1. Prerequisites

- ArcadeDB running with PostgreSQL protocol enabled (default port 5432)
- Node.js 18+
- `pg` library

```
npm install pg
```

4.22.2. Connect to ArcadeDB

ArcadeDB speaks the PostgreSQL wire protocol, so you can use the standard `pg` client. No special SDK is needed.

```
const { Client } = require('pg');

const client = new Client({
  host: 'localhost',
  port: 5432,
  database: 'mydb',
  user: 'root',
  password: 'arcadedb',
});
```

```
await client.connect();
console.log('Connected to ArcadeDB');
```

4.22.3. Create a Schema

Use SQL to create vertex and edge types:

```
// Create vertex types
await client.query('CREATE VERTEX TYPE Person IF NOT EXISTS');
await client.query('CREATE VERTEX TYPE Movie IF NOT EXISTS');

// Create edge type
await client.query('CREATE EDGE TYPE Acted IF NOT EXISTS');

// Insert data
await client.query("CREATE VERTEX Person SET name = 'Alice', age = 30");
await client.query("CREATE VERTEX Person SET name = 'Bob', age = 25");
await client.query("CREATE VERTEX Movie SET title = 'The Matrix', year = 1999");
await client.query(`
  CREATE EDGE Acted FROM (SELECT FROM Person WHERE name = 'Alice')
                    TO (SELECT FROM Movie WHERE title = 'The Matrix')
                    SET role = 'Trinity'
`);
```

4.22.4. Query with SQL

```
const result = await client.query(`
  MATCH {type: Person, as: person}.outE('Acted'){as: acted}.inV(){type: Movie, as:
movie}
  RETURN person.name AS name, movie.title AS title, acted.role AS role
`);

for (const row of result.rows) {
  console.log(`${row.name} acted in ${row.title} as ${row.role}`);
}
```

4.22.5. Query with Cypher

ArcadeDB supports Cypher queries through the same connection:

```
const result = await client.query(
  '{cypher} MATCH (p:Person)-[a:Acted]->(m:Movie) RETURN p.name, m.title, a.role'
);

for (const row of result.rows) {
  console.log(`${row['p.name']} acted in ${row['m.title']} as ${row['a.role']}`);
```

```
}
```

Prefix Cypher queries with `{cypher}` when using the PostgreSQL protocol.

4.22.6. Vector Search

Add embeddings to your data and query by similarity:

```
// Create a type with a vector property
await client.query('CREATE VERTEX TYPE Product IF NOT EXISTS');
await client.query('CREATE PROPERTY Product.embedding IF NOT EXISTS LIST OF FLOAT');
await client.query("CREATE INDEX IF NOT EXISTS ON Product (embedding) LSM_VECTOR
METADATA {dimensions: 4, similarity: 'COSINE'}");

// Insert products with embeddings
await client.query("CREATE VERTEX Product SET name = 'Laptop', embedding = [0.9, 0.1,
0.8, 0.2]");
await client.query("CREATE VERTEX Product SET name = 'Tablet', embedding = [0.8, 0.2,
0.7, 0.3]");
await client.query("CREATE VERTEX Product SET name = 'Phone',  embedding = [0.7, 0.3,
0.6, 0.4]");
await client.query("CREATE VERTEX Product SET name = 'Book',   embedding = [0.1, 0.9,
0.2, 0.8]");

// Find the 3 most similar products to a query vector
const similar = await client.query(`
  SELECT name, distance FROM (
    SELECT expand(vector.neighbors('Product[embedding]', [0.85, 0.15, 0.75, 0.25], 3))
  )
`);
console.log('Similar products:');
for (const row of similar.rows) {
  console.log(`  ${row.name}`);
}
```

4.22.7. Full Example

```
#!/usr/bin/env node
/**
 * ArcadeDB JavaScript quickstart -- graph + vector search.
 */

const { Client } = require('pg');

async function main() {
  const client = new Client({
    host: 'localhost', port: 5432, database: 'mydb',
    user: 'root', password: 'arcadedb',
```

```
  });
  await client.connect();

  // Create schema
  await client.query('CREATE VERTEX TYPE Person IF NOT EXISTS');
  await client.query('CREATE VERTEX TYPE Movie IF NOT EXISTS');
  await client.query('CREATE EDGE TYPE Acted IF NOT EXISTS');

  // Insert data
  await client.query("CREATE VERTEX Person SET name = 'Alice', age = 30");
  await client.query("CREATE VERTEX Movie SET title = 'The Matrix', year = 1999");
  await client.query(`
    CREATE EDGE Acted FROM (SELECT FROM Person WHERE name = 'Alice')
                      TO (SELECT FROM Movie WHERE title = 'The Matrix')
                      SET role = 'Trinity'
  `);

  // Query with SQL MATCH
  const result = await client.query(`
    MATCH {type: Person, as: person}.outE('Acted'){as: acted}.inV(){type: Movie, as:
movie}
    RETURN person.name AS name, movie.title AS title, acted.role AS role
  `);
  for (const row of result.rows) {
    console.log(`${row.name} acted in ${row.title} as ${row.role}`);
  }

  await client.end();
}

main().catch(console.error);
```

4.22.8. Next Steps

- Supply Chain Use Case — Full JavaScript implementation with 7 query patterns
- PostgreSQL Protocol Reference
- SQL Reference
- HTTP/JSON API — Alternative: use `fetch()` with ArcadeDB's REST API

4.23. HTTP / JSON from JavaScript

Connecting to ArcadeDB from JavaScript / TypeScript

- **Native HTTP driver** — `npm install @arcadedb/driver`, generated from ArcadeDB's OpenAPI contract. Start here for most applications.
- **Native gRPC driver** — `npm install @arcadedb/driver-grpc`, for streaming and bulk inserts.

- **PostgreSQL wire protocol** — `npm install pg`, run SQL through a driver you may already have.
- **Neo4j BOLT** — `npm install neo4j-driver`, run Cypher queries.

This page covers the remaining option: calling the HTTP/JSON API directly, with no driver at all.

ArcadeDB's HTTP/JSON API is reachable from any HTTP client, so JavaScript can talk to it with nothing installed. Prefer `@arcadedb/driver` when you can — it gives you typed results, transactions, and error handling for one dependency. Reach for raw HTTP when you cannot add a dependency at all: a serverless function you want to keep cold-start-free, a build script, or a quick probe.

See the HTTP/JSON API reference for every endpoint and parameter.

4.23.1. Using fetch

`fetch` is built into Node.js 18 and later, and into every current browser — so this needs no packages.

```
const base = 'http://localhost:2480/api/v1';
const auth = 'Basic ' + Buffer.from('root:playwithdata').toString('base64');

const response = await fetch(`${base}/command/mydb`, {
  method: 'POST',
  headers: {
    Authorization: auth,
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    language: 'sql',
    command: 'SELECT FROM Person LIMIT 10',
  }),
});

if (!response.ok) {
  throw new Error(`${response.status} ${await response.text()}`);
}

const data = await response.json();
console.log(data.result);
```

```
[
  {
    '@rid': '#1:0',
    '@type': 'Person',
    '@cat': 'v',
    name: 'Alice',
    age: 30
  }
```

```
]
```

`Buffer` is Node-only; in a browser, build the same header with `btoa('root:playwithdata')`.

Two things worth knowing. `fetch` does not throw on an HTTP error status — a 401 or a 500 resolves normally, and only a network failure rejects — so check `response.ok` yourself, as above. And `fetch` has no timeout at all: wrap the call with `signal: AbortSignal.timeout(5000)` if a hung connection would be worse than a failed one.

The response is the full envelope, not a bare array: `result` holds the rows, alongside `limit`, `returned`, and `truncated`. A `truncated` of `true` means the server stopped serializing before the result set ended, so `result` is a partial answer that looks exactly like a complete one — see Native Drivers for what to do about it.

4.23.2. Using axios

`axios` handles the base64 encoding and throws on non-2xx responses, which is the main reason to prefer it over `fetch` here.

```
npm install axios
```

```
import axios from 'axios';

const res = await axios.post(
  'http://localhost:2480/api/v1/command/mydb',
  { language: 'sql', command: 'SELECT FROM Person LIMIT 10' },
  { auth: { username: 'root', password: 'playwithdata' } },
);

console.log(res.data.result);
```

```
[
  {
    '@rid': '#1:0',
    '@type': 'Person',
    '@cat': 'v',
    name: 'Alice',
    age: 30
  }
]
```

4.23.3. TypeScript

Both examples are valid TypeScript as written, and need no type packages beyond the `@types/node` a Node project already has: `fetch` and `Buffer` come from it, and `axios` ships its own types. Annotate the response shape yourself if you want the rows typed:

```
interface QueryResponse<T> {
  result: T[];
  limit: number;
  returned: number;
  truncated: boolean;
}

const data = (await response.json()) as QueryResponse<{ name: string; age: number }>;
```

4.23.4. Further Reading

- Native HTTP driver — typed results, transactions, and error handling
- HTTP/JSON API reference — every endpoint and parameter
- Supply Chain Use Case — a full JavaScript implementation with 7 query patterns

4.24. Go HTTP Driver

`github.com/ArcadeData/arcadedb-drivers/go/arcadedb` is ArcadeDB's Go HTTP client, generated from the server's OpenAPI contract. A hand-written facade on top covers queries, commands, transactions, streaming, batch loading, insert sessions, vector search, time series, server administration, login sessions and API tokens, and listing and terminating running work.

4.24.1. Install

```
go get github.com/ArcadeData/arcadedb-drivers/go/arcadedb@latest
```

Requires Go 1.26 or later. Go has no package registry: the module is fetched through the Go module proxy like any other, and each release is the git tag `go/arcadedb/v<version>` in the arcadedb-drivers repository.

4.24.2. Connect and query

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ArcadeData/arcadedb-drivers/go/arcadedb"
)

func main() {
    ctx := context.Background()
```

```
    srv, err := arcadedb.NewServer("http://localhost:2480",
        arcadedb.WithBasicAuth("root", "playwithdata"))
    if err != nil {
        log.Fatal(err)
    }
    defer srv.Close()

    db := srv.DB("mydb")

    env, err := db.Query(ctx, arcadedb.SQL, "SELECT FROM Person WHERE age > ?", map
[string]any{"0": 21})
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(env.Result, env.Returned, env.Truncated)
}
```

```
[map[@cat:v @rid:#1:0 @type:Person age:30 name:Alice] map[@cat:v @rid:#1:1
@type:Person age:25 name:Bob]] 2 false
```

`Query` returns a `QueryEnvelope`. The rows are in `Result`, a `[]map[string]any`, and the envelope also carries `Limit`, `Returned` and `Truncated`. Check `Truncated` before you treat `Result` as the whole answer; Native Drivers explains why. Pass `arcadedb.WithLimit(n)` to `Query` to set the row cap for one call (`-1` for none). `Command` takes no limit: `WithLimit` is a `QueryOption` only, so passing it to `Command` does not compile.

An `EXPLAIN` or `PROFILE` statement answers with its execution plan instead of rows: the envelope's `Explain` holds it as indented text and `ExplainPlan` as the same plan in structured form, while `Result` is empty and `Returned` is `0`. An empty `Result` there is not "no rows matched".

Rows are decoded with `encoding/json`, so numbers arrive as `float64`. An integer larger than 2^{53} loses precision.

Positional parameters are zero-indexed, and `"1"` silently returns nothing.

Positional parameters are keyed by their zero-based index as a string. For the single placeholder above, the key is `"0"`, not `"1"`. Passing `map[string]any{"1": 21}` returns no error. The query runs with no value bound to `?` and comes back empty:

```
[] 0 false
```

Named parameters sidestep the problem: `"SELECT FROM Person WHERE age > :min"` with `map[string]any{"min": 21}`.

`NewServer` takes functional options:

- `WithBasicAuth(user, password)` or `WithBearerToken(token)`, for example a session token from

`srv.Login` (see Sessions and API tokens).

- `WithHeader(name, value)` adds a header to every request.
- `WithHTTPClient(client)` replaces the default `http.Client`.

A `*Server` is safe for concurrent use, and `srv.DB(name)` makes no request; it only returns a handle. Every call takes a `context.Context` as its first argument. There is no async variant of anything: run calls in goroutines when you want them concurrent.

There is no timeout by default.

Without `WithHTTPClient`, the server handle uses a bare `http.Client`, and Go's default client has no timeout. A call against a stalled connection can block forever. Bound each call with a context deadline:

```
ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
defer cancel()
env, err := db.Query(ctx, arcadedb.SQL, "SELECT FROM Person", nil)
```

You can also pass `arcadedb.WithHTTPClient(&http.Client{Timeout: 30 * time.Second})`. A client-level `Timeout` also cuts off a long streaming response halfway through, though, so for the streaming methods below prefer a per-call context.

`Close` releases idle connections, including those of a client you passed in with `WithHTTPClient`. Call it when you are done with the server handle, usually with `defer`.

4.24.3. Streaming queries

`QueryStream` and `CommandStream` read the result as newline-delimited JSON and return an `iter.Seq2[StreamEvent, error]`, so you range over them instead of holding the whole result in memory:

```
for ev, err := range db.QueryStream(ctx, arcadedb.SQL, "SELECT name FROM Person ORDER
BY name", nil) {
    if err != nil {
        log.Fatal(err)
    }
    if ev.Stats != nil {
        fmt.Println("returned", ev.Stats.Returned, "truncated", ev.Stats.Truncated)
        continue
    }
    fmt.Println(ev.Record["name"])
}
```

```
Alice
Bob
```

```
returned 2 truncated false
```

The iterator yields events. Each `StreamEvent` has either `Record` or `Stats` set, never both. The `Stats` event always comes last in a complete stream and carries the same `Limit`, `Returned` and `Truncated` values as the buffered envelope. If a stream ends without a `Stats` event, it was cut short (a server write timeout or a dropped connection, for example), and the rows you saw may be a partial answer.

Some behavior to know about:

- The request is sent on the first iteration and again on every later range over the same iterator. Treat the iterator as single-use.
- An error that happens after the server has sent its `200` status arrives in-band. The driver yields it once as an `*ArcadeDBError`, and the loop ends. Rows yielded before it stay delivered. Its `Status` is the event's own status (503 for a retryable conflict, 409 for a duplicated key, 500 for an unexpected failure), and only an event that carries none reads as 200. Up to 0.2.0 it was always 200.
- Leaving the loop early with `break` or `return` closes the response body.
- `CommandStream` only works for read-only statements. The server refuses a mutating statement with `400`, because the streamed rows would reach you before the commit, and that commit could still roll back. Use `Command` for writes.

4.24.4. Transactions

```
var total any
err := db.Transaction(ctx, func(tx *arcadedb.Database) error {
    if _, err := tx.Command(ctx, arcadedb.SQL, "INSERT INTO Account SET balance =
100", nil); err != nil {
        return err
    }
    env, err := tx.Query(ctx, arcadedb.SQL, "SELECT sum(balance) AS total FROM
Account", nil)
    if err != nil {
        return err
    }
    total = env.Result[0]["total"]
    return nil
})
if err != nil {
    log.Fatal(err)
}
fmt.Println("total:", total)
```

```
total: 100
```

`Transaction` begins a server-side transaction and calls your function with a second `*Database`, `tx`

above, that carries the transaction's session id. It takes a callback instead of returning a handle because Go has no `with` statement: a handle would rely on every caller remembering `defer tx.Rollback()`, while a callback cannot leak a session. The contract:

- If the function returns `nil`, the transaction commits.
- If the function returns an error or panics, the transaction rolls back and the error is returned (or the panic re-raised). If the rollback also fails, you get a `*arcadedb.TxError` whose `Unwrap` returns only your error, so `errors.Is` and `errors.As` match what your code produced. The rollback's error stays on the `RollbackErr` field.
- If the commit itself fails, a best-effort rollback is issued first, so the server-side session is not left open for `arcadedb.server.httpSessionExpireTimeout` to reap it.

Rollbacks run on `context.WithoutCancel(ctx)`, so a callback that failed because `ctx` was cancelled still gets its transaction rolled back.

Watch which handle you call. Calls made through the outer `db` while a transaction is open do not join it; each one auto-commits on its own. Only calls made through `tx` take part. Batch loads cannot join a transaction at all, so `tx.BatchLoad` returns `arcadedb.ErrBatchInTransaction` before sending anything.

4.24.5. Two error models

Facade methods return an `*arcadedb.ArcadeDBError` for any non-2xx response. `Ready` is the one exception: it answers a `503` with `false` and no error. Match the error with `errors.As`:

```
_, err := db.Query(ctx, arcadedb.SQL, "SELECT FROM NoSuchType", nil)
var aerr *arcadedb.ArcadeDBError
if errors.As(err, &aerr) {
    fmt.Println(aerr.Status, aerr.ErrorMessage, aerr.Detail, aerr.RequestID)
}
```

```
500 Internal error Type with name 'NoSuchType' was not found 44613703d0741a4c-11
```

`RequestID` is the server's `X-Request-Id` for that call and differs on every request; use it to find the failure in the server log. The body's `error` string is in `ErrorMessage` rather than `Error`, because Go does not allow a field and a method with the same name.

Two more fields come from the response's headers: `RetryAfter`, a `time.Duration` the server asked you to wait before retrying (on a 503, or a 409 for an identical request still in flight), and `CommitIndex`, the Raft index to send back as `X-ArcadeDB-Read-After` on a later read. Both are 0 when absent.

A 409 has two meanings. It is either "an identical request is still in flight", which is retryable after `RetryAfter`, or "query terminated", because an operator stopped the statement and its writes were rolled back. Retrying the second undoes the operator's decision. Tell them apart with `arcadedb.IsQueryTerminated(err)`, never

by `Status` alone.

A rejected time-series write returns a `*arcadedb.TimeSeriesWriteError`, which embeds `*ArcadeDBError` and says how many samples were written and dropped and why. A write is not atomic: the written samples landed, so re-sending the whole batch writes them twice.

`srv.Raw()` takes the opposite approach. It returns the generated `*generated.ClientWithResponses`, which never turns a status code into an error: `err` reports transport failures only, and you check the status yourself.

```
resp, err := srv.Raw().ListDatabasesWithResponse(ctx)
if err != nil {
    log.Fatal(err) // transport failure only
}
fmt.Println(resp.StatusCode(), resp.JSON200.Result)
```

```
200 [mydb]
```

Use `Raw()` for the endpoints the facade does not wrap: cluster, AI, MCP and metrics operations, and the two protobuf PromQL remote read and write routes. For server commands use `srv.ServerCommand` rather than `POST /api/v1/server` through `Raw()`, whose typed response misreads the server's `{"result": "ok"}` answer. Mixing up which of the two surfaces you are calling is the most common way to end up with an unhandled failure.

`srv.Exists(ctx, name)` returns `false` both when a database does not exist and when it exists but the caller is not allowed to see it. The server answers both cases identically, so `false` is not proof that a database is absent.

4.24.6. Beyond queries

The database handle also has:

- `db.BatchLoad` and `db.BatchLoadStream` for bulk loading vertices and edges through `/api/v1/batch`. A load commits every `commitEvery` records, so it is not atomic: a load that fails partway leaves earlier chunks committed, and retrying the whole payload duplicates them.
- `db.Vector()` for vector, hybrid and full-text search.
- `db.TS()`, `db.Grafana()` and `db.PromQL()` for time-series ingestion and querying, including `db.PromQL().LabelValues`.
- `db.InsertSession` for a duplex insert session, below.

The package README linked below covers each of them.

4.24.7. Labelling work with a query tag

`arcadedb.WithQueryTag("nightly-report")` sends the `X-ArcadeDB-Query-Tag` header, so you can list and

terminate the work by it. `Query`, `QueryStream`, `Command`, `CommandStream` and `Transaction` all take it; a batch load takes it as `generated.ExecuteBatchParams.XArcadeDBQueryTag`.

```
err := db.Transaction(ctx, func(tx *arcadedb.Database) error {
    _, err := tx.Command(ctx, arcadedb.SQL, "UPDATE Account SET flagged = true WHERE
balance < 0", nil)
    return err
}, arcadedb.WithQueryTag("audit"))
```

A transaction's tag is sent on the begin request only; the server applies it to every statement in the transaction that carries no tag of its own.

4.24.8. Duplex insert session

`db.InsertSession` opens an insert session on the server's `/ws` endpoint. Unlike `BatchLoad`, whose commit policy is fixed before the load starts, a session lets you read each chunk's acknowledgement before deciding what to send next, and whether to commit at all:

```
s, err := db.InsertSession(ctx, arcadedb.WithInsertTargetType("Person"))
if err != nil {
    return err
}
defer s.Close() // rolls back if the session is still open; idempotent

ack, err := s.SendChunk(ctx, []map[string]any{{"name": "Alice"}, {"name": "Bob"}})
if err != nil {
    return err
}
// ack.Inserted, ack.Failed, ack.Errors: decide here whether to go on. Nothing is
durable yet.
res, err := s.Commit(ctx) // res.Outcome == "commit"
```

A session is not safe for concurrent use: use one per goroutine.

4.24.9. Administration

`CreateDatabase`, `DropDatabase`, `CreateUser`, `UpdateUser`, `DropUser`, `ListUsers`, `SaveGroup`, `DeleteGroup` and `ListGroups` are on the `*Server`, and `ServerCommand` runs any other server command. All of them are root-only:

```
err := srv.CreateDatabase(ctx, "tenant_a")
err = srv.CreateUser(ctx, "alice", "a-long-password",
    map[string][]string{"tenant_a": {"admin"}}) // database -> groups
users, err := srv.ListUsers(ctx)
err = srv.DropUser(ctx, "alice")
err = srv.DropDatabase(ctx, "tenant_a")
```

A user or database that already exists is a 409 on ArcadeDB 26.10.2 and later. Older servers answer 403 for a duplicate user and 400 for a duplicate database.

4.24.10. Sessions and API tokens

```
// Log in as each user once, then keep a client that holds only that user's token.
login, err := srv.LoginAs(ctx, "alice", "a-long-password") // or srv.Login(ctx)
alice := srv.WithToken(login.Token)                        // srv is unchanged
dbs, err := alice.ListDatabases(ctx)                       // only what alice may see
err = alice.Logout(ctx)

// A long-lived machine credential (root only).
created, err := srv.CreateAPIToken(ctx, arcadedb.APITokenRequest{Name: "ci", Database:
"tenant_a"})
store(created.Token)                              // the only time the plaintext
exists
err = srv.DeleteAPIToken(ctx, created.TokenHash)  // by hash, never by token
```

`ListSessions` and `ListAPITokens` list what exists. Each user holds at most 100 sessions, and a further login silently evicts the oldest, so log in once per user and reuse the token rather than logging in per request. Wrong Basic credentials, like an invalid, expired or logged-out token, are answered 401; 403 is what an authenticated user gets for an operation it is not allowed (*since v26.11.1*: before, wrong Basic credentials were answered 403).

4.24.11. Running work

```
qs, err := srv.ListQueries(ctx, arcadedb.ListQueriesOptions{Tag: "nightly-report"}) //
or Database
rs, err := srv.TerminateQueriesByTag(ctx, "nightly-report", arcadedb.TerminateOptions
{Wait: 3 * time.Second})
r, err := srv.TerminateQuery(ctx, qs[0].ID, arcadedb.TerminateOptions{})
txs, err := srv.ListTransactions(ctx)
r, err = srv.TerminateTransaction(ctx, txs[0].ID, arcadedb.TerminateOptions{})
p, err := db.Progress(ctx) // long-running maintenance operations
```

Any authenticated user may call these: root sees and stops every statement, another user only their own. Each terminate answer carries a `Status`, and which one you get decides whether you have anything to undo:

`Status`	**Meaning**
`QueryTerminated`	It stopped, and what it wrote was rolled back.
`QueryCompleted`	It finished before it noticed, so what it did stands.
`QueryTerminating`	It had not stopped when the wait ended; the request's own outcome is the answer.
`QueryNotFound`	No such statement on this server, or another user's.

- **Each server sees only its own work.** These commands are not forwarded to the cluster leader, so behind a load balancer the server you reach may not be the one running the statement.
- **A commit after `TerminateTransaction` fails on ArcadeDB 26.11.1 and later.** The transaction's writes are rolled back, and the server answers its owner's commit with 404. Earlier servers answered 204, as if it had committed (ArcadeData/arcadedb#10164).

4.24.12. Next steps

- Native Drivers — the result envelope and why `truncated` matters, and how this driver compares to the others.
- Go — gRPC Driver — the gRPC counterpart for throughput-sensitive server-to-server work.
- The package README for the full API, including batch loading, vector search and time series.
- The API reference on pkg.go.dev.

4.25. Go gRPC Driver

`github.com/ArcadeData/arcadedb-drivers/go/arcadedbgrpc` is ArcadeDB's Go gRPC client, generated from the server's Protobuf contract. A small facade on top handles authentication, the streaming RPCs and transactions. It shares no code and no dependency with the Go HTTP driver, so using one never downloads the other.

4.25.1. Install

```
go get github.com/ArcadeData/arcadedb-drivers/go/arcadedbgrpc@latest
```

Requires Go 1.26 or later. The module depends only on `google.golang.org/grpc` and `google.golang.org/protobuf`.

The gRPC server is a plugin, and it is not started by default.

Register it in the server's plugin list before connecting:

```
-Darcadedb.server.plugins
=GRPC:com.arcadedb.server.grpc.GrpcServerPlugin
```

Once registered, it listens on port `50051`. See the gRPC API reference for the full option list, including TLS and message-size settings.

4.25.2. Connect and query

```
package main
```

```go
import (
    "context"
    "fmt"
    "log"

    "github.com/ArcadeData/arcadedb-drivers/go/arcadedbgrpc"
    "github.com/ArcadeData/arcadedb-drivers/go/arcadedbgrpc/generated"
)

func main() {
    ctx := context.Background()

    c, err := arcadedbgrpc.NewClient("localhost:50051",
        arcadedbgrpc.WithPasswordAuth("root", "playwithdata", "mydb"),
        arcadedbgrpc.WithInsecure()) // plaintext, opted into explicitly
    if err != nil {
        log.Fatal(err)
    }
    defer c.Close()

    resp, err := c.Raw().ExecuteQuery(ctx, &generated.ExecuteQueryRequest{
        Database: "mydb",
        Query:    "SELECT FROM Person WHERE age > 21",
        Language: "sql",
    })
    if err != nil {
        log.Fatal(err)
    }
    for _, result := range resp.GetResults() {
        for _, rec := range result.GetRecords() {
            fmt.Println(rec.GetRid(), rec.GetProperties()["name"].GetStringValue())
        }
    }
}
```

```
#1:0 Alice
#1:1 Bob
```

The target is grpc-go's native form (`localhost:50051`, `[::1]:50051`, or a resolver name such as `dns:///db.example.com:50051`), not a URL. `NewClient` refuses a target that starts with `http://` or `https://`. Like `grpc.NewClient`, it performs no I/O: the connection opens on the first call.

A `*Client` is safe for concurrent use. Every call takes a `context.Context` first, and there is no default timeout, so bound calls with a context deadline. Every facade method also accepts trailing `…grpc.CallOption` values and passes them to grpc-go unchanged.

4.25.3. `Raw()` and the facade

`c.Raw()` is the generated client for `ArcadeDbService`, the data plane, and every one of its 21 RPCs is reachable through it, as `ExecuteQuery` is above. The facade adds wrappers for the RPCs that the generated client alone handles badly:

- `StreamQuery` and `TimeSeriesQuery` turn server streams into Go iterators.
- `InsertStream` and `TimeSeriesWriteStream` feed client streams from an iterator you supply.
- `Transaction` pairs begin, commit and rollback around a callback.

Everything else goes through `Raw()` directly: the CRUD calls, `VectorSearch`, `HybridSearch`, `FullTextSearch`, `TimeSeriesWrite`, `TimeSeriesLatest`, `BulkInsert`, `InsertBidirectional` and `GraphBatchLoad`.

Values come back as the generated protobuf types. In a `GrpcRecord`, the record id from `GetRid()` is a plain string, but each entry in `GetProperties()` is a `*generated.GrpcValue`, a protobuf message whose `Kind` oneof holds the real value. Unwrap it with a type switch:

```
func unwrap(v *generated.GrpcValue) any {
    switch k := v.GetKind().(type) {
    case *generated.GrpcValue_LinkValue:
        return k.LinkValue.GetRid()
    case *generated.GrpcValue_Int32Value:
        return k.Int32Value
    case *generated.GrpcValue_Int64Value:
        return k.Int64Value
    case *generated.GrpcValue_DoubleValue:
        return k.DoubleValue
    case *generated.GrpcValue_StringValue:
        return k.StringValue
    case *generated.GrpcValue_BoolValue:
        return k.BoolValue
    case nil:
        return nil
    default:
        return k // lists, maps, embedded documents, decimals, timestamps, bytes
    }
}

props := make(map[string]any, len(rec.GetProperties()))
for k, v := range rec.GetProperties() {
    props[k] = unwrap(v)
}
fmt.Println(rec.GetRid(), props)
```

```
#1:0 map[@cat:v @rid:#1:0 @type:Person age:30 name:Alice]
#1:1 map[@cat:v @rid:#1:1 @type:Person age:25 name:Bob]
```

`LinkValue` is the one scalar-looking case that is itself a message (`GrpcLink`), so read its `GetRid()`. The driver has no helper that does this unwrapping for you.

A capped `ExecuteQuery` fails instead of truncating.

`ExecuteQuery` builds its whole answer into one gRPC message, so the server caps it: `server.grpcQueryMaxResultRows` (100000 by default) is a hard ceiling, and a larger result fails the call with `ResourceExhausted`. Unlike the HTTP drivers, there is no `truncated` flag to check. Use `StreamQuery` for result sets that large. See Settings for the streaming limits.

4.25.4. Authentication is attached to the connection

`WithPasswordAuth(user, password, database)` and `WithBearerToken(token)` do not attach credentials to individual calls. `NewClient` installs them as connection interceptors that cover all four gRPC call shapes, so `c.Raw().ExecuteCommand(…)` carries the same metadata as `c.StreamQuery(…)`. Most RPCs are reachable only through `Raw()`; per-call credentials on the facade methods would leave all of those calls anonymous. The metadata is appended to whatever you already set on the outgoing context, so your own metadata (a request id, for example) still reaches the server.

The insecure-channel guard.

`WithPasswordAuth` sends the password as plaintext call metadata. To keep that from happening by accident, `NewClient` refuses to pair it with a plaintext connection unless you pass TLS through `WithTransportCredentials` or opt in with `WithInsecure()`:

```
_, err := arcadedbgrpc.NewClient("localhost:50051",
    arcadedbgrpc.WithPasswordAuth("root", "playwithdata", "mydb"))
fmt.Println(errors.Is(err, arcadedbgrpc.ErrInsecureChannel), err)
```

```
true arcadedbgrpc: refusing to send credentials over a connection
without transport credentials: NewClient would send a plaintext
password to "localhost:50051"; pass WithTransportCredentials(creds),
switch to WithBearerToken, or pass WithInsecure() to opt in explicitly
(ArcadeData/arcadedb#5048)
```

For TLS, pass `arcadedbgrpc.WithTransportCredentials(credentials.NewTLS(&tls.Config{}))`. Do not pass TLS through `WithDialOptions`: it works, but the guard cannot see it there and still demands `WithTransportCredentials` or `WithInsecure`. A bearer token is not a password and never trips the guard.

4.25.5. Streaming queries

```
for rec, err := range c.StreamQuery(ctx, &generated.StreamQueryRequest{
```

```
    Database:      "mydb",
    Query:         "SELECT FROM Person",
    Language:      "sql",
    RetrievalMode: generated.StreamQueryRequest_CURSOR,
    BatchSize:     500,
}) {
    if err != nil {
        log.Fatal(err) // a grpc-go status error; records already yielded stay yielded
    }
    fmt.Println(rec.GetRid(), rec.GetProperties()["name"].GetStringValue())
}
```

```
#1:0 Alice
#1:1 Bob
```

`StreamQuery` returns an `iter.Seq2[*generated.GrpcRecord, error]` that flattens the server's batches into one record at a time. It sends your request as given and picks no default for `RetrievalMode` or `BatchSize`, so set both. There are three retrieval modes:

- `CURSOR`, the proto zero value: the server runs the query once and streams rows as you iterate.
- `MATERIALIZE_ALL`: the server runs the query to completion first, then sends the result in batches.
- `PAGED`: same as `CURSOR` *(since v26.12.1)*: the query runs once and its rows are streamed in batches. Earlier versions re-ran a SQL query with `LIMIT`/`SKIP` for every batch, which made a large stream slow.

The iterator is lazy and each range issues a new RPC, so treat it as single-use. Leaving the loop early cancels the server stream. A failure, including one in the middle of the stream, arrives once as a final `(nil, err)` pair.

4.25.6. Streaming inserts

`InsertStream` takes your rows as an `iter.Seq[[]*generated.GrpcRecord]`. Each batch becomes one wire chunk, so you decide how rows are batched. The driver handles the envelope: one session id for the stream, an incrementing chunk sequence, `Database` on the first chunk and `last` on the final one.

`InsertStream` never sets `options.database`. Older servers read the database from `InsertOptions.database` alone (ArcadeData/arcadedb#6597), so against one the stream reports its rows as received, inserts none, and returns no error. 0.2.0 copied `Database` into the options to cover those servers; 0.3.0 dropped the copy. Set `options.database` yourself if you must use such a server.

```
person := func(name string) *generated.GrpcRecord {
    return &generated.GrpcRecord{Type: "Person", Properties: map[string]*generated
.GrpcValue{
        "name": {Kind: &generated.GrpcValue_StringValue{StringValue: name}},
    }}
}
```

```
batches := func(yield func([]*generated.GrpcRecord) bool) {
    if !yield([]*generated.GrpcRecord{person("Carol")}) {
        return
    }
    yield([]*generated.GrpcRecord{person("Dave")})
}

sum, err := c.InsertStream(ctx, arcadedbgrpc.InsertStreamRequest{
    Database: "mydb",
    Options:  &generated.InsertOptions{TargetClass: "Person"},
    Chunks:   batches,
})
if err != nil {
    log.Fatal(err)
}
fmt.Println(sum.GetInserted(), sum.GetFailed())
```

```
2 0
```

Check the summary even when the error is `nil`. A server that ends the stream early with a summary makes the call return that summary and no error. Your sequence runs on the calling goroutine, so cancelling `ctx` does not interrupt a sequence that is blocked producing its next batch. If your sequence waits on a channel, a file or the network, have it watch `ctx` itself.

`TimeSeriesWriteStream` works the same way for time-series points. Its `Precision` field is a required pointer, because the enum's zero value is milliseconds, while the HTTP line-protocol endpoint treats a missing precision as nanoseconds.

4.25.7. Transactions

```
var total int32
err := c.Transaction(ctx, "mydb", func(tx *arcadedbgrpc.TxHandle) error {
    if _, err := tx.ExecuteCommand(ctx, &generated.ExecuteCommandRequest{
        Command:  "INSERT INTO Account SET balance = 100, owner = 'grpc-demo'",
        Language: "sql",
    }); err != nil {
        return err // rolls back
    }
    resp, err := tx.ExecuteQuery(ctx, &generated.ExecuteQueryRequest{
        Query:    "SELECT sum(balance) AS total FROM Account WHERE owner = 'grpc-
demo'",
        Language: "sql",
    })
    if err != nil {
        return err
    }
    total = resp.GetResults()[0].GetRecords()[0].GetProperties()["total"]
.GetInt32Value()
```

```
    return nil // commits
})
if err != nil {
    log.Fatal(err)
}
fmt.Println("total:", total)
```

```
total: 100
```

`Transaction` calls `BeginTransaction`, hands your function a `*TxHandle` bound to the new transaction, and then commits or rolls back with the same contract as the HTTP driver:

- If the function returns `nil`, the transaction commits.
- If it returns an error or panics, the transaction rolls back and the error is returned (or the panic re-raised). If the rollback also fails, you get a `*arcadedbgrpc.TxError` whose `Unwrap` returns only your error.
- If the commit fails, a best-effort rollback runs first, and the commit's error is returned.
- A commit that the server answers with `committed=false` (a transaction it has already reaped, for example) returns `arcadedbgrpc.ErrNotCommitted` instead of reporting success.

The handle has `ExecuteQuery`, `ExecuteCommand`, `CreateRecord`, `UpdateRecord`, `DeleteRecord`, `LookupByRid`, `VectorSearch`, `HybridSearch`, `FullTextSearch`, `TimeSeriesLatest`, `StreamQuery`, `TimeSeriesQuery` and `InsertStream`. Each sends a copy of your request with `Database` and the `Transaction` field set to this transaction, so you never set either yourself, and the request you passed in is left untouched for reuse. Rows sent through `tx.InsertStream` commit or roll back with the transaction; on older servers, which ran an insert stream outside any transaction (ArcadeData/arcadedb#6607), they survived a rollback. `TimeSeriesWriteStream` is not on the handle, because a time-series chunk has no transaction field on the wire, and `BulkInsert`, `InsertBidirectional` and `GraphBatchLoad` are raw-only.

As with the HTTP driver, calls made through `c` or `c.Raw()` while the function runs do not join the transaction. Do not keep the handle after `Transaction` returns: the server refuses its calls with `FailedPrecondition`.

4.25.8. Errors

A failed RPC returns grpc-go's status error unchanged, from `Raw()` and from every facade method. The driver adds no error type of its own for server failures. Match it with `status.Code(err)` or `status.FromError(err)`:

```
_, err := c.Raw().ExecuteQuery(ctx, &generated.ExecuteQueryRequest{
    Database: "mydb", Query: "SELECT FROM NoSuchType", Language: "sql",
})
if st, ok := status.FromError(err); ok && st.Code() != codes.OK {
    fmt.Println(st.Code(), st.Message())
}
```

```
NotFound Query execution failed: Type with name 'NoSuchType' was not found
```

The status you are most likely to meet first is `Unauthenticated`. A stock ArcadeDB gRPC server refuses anonymous calls, so a client built without an auth option connects fine and then fails on the first RPC with `Authentication required`.

The facade's own refusals are sentinel errors, matched with `errors.Is`: `ErrInsecureChannel`, `ErrNoTransactionID` (the server returned a blank transaction id, so your function never ran) and `ErrNotCommitted`. Match `*TxError` with `errors.As`.

4.25.9. Administration: `RawAdmin`

Server administration lives on a second service, `ArcadeDbAdminService`: 44 RPCs covering databases, users, groups, API tokens, settings, backups, the profiler and cluster operations. `Raw()` does not reach it. `c.RawAdmin()` returns its generated client on the same connection:

```
admin, err := c.RawAdmin()
if err != nil {
    log.Fatal(err) // ErrInsecureChannel: the client has neither TLS nor WithInsecure
}
resp, err := admin.ListDatabases(ctx, &generated.ListDatabasesRequest{
    Credentials: &generated.DatabaseCredentials{Username: "root", Password:
"playwithdata"},
})
if err != nil {
    log.Fatal(err)
}
fmt.Println(resp.GetDatabases())
```

```
[mydb]
```

Admin RPCs take their credentials inside the request message, not from the connection, so `WithPasswordAuth` and `WithBearerToken` do not authenticate them. Because those credentials travel in the request body, `RawAdmin` has its own guard: it returns `ErrInsecureChannel` unless the client was built with `WithTransportCredentials` or `WithInsecure`, whatever auth option you chose. A plaintext client without `WithInsecure` still works for the data plane; only `RawAdmin` fails.

4.25.10. Next steps

- Native Drivers — how this driver compares to the others, and when to reach for gRPC instead of HTTP.
- Go — HTTP Driver — the HTTP counterpart for general application traffic.
- The package README for the time-series streams, vector search and the full transaction contract.
- The API reference on pkg.go.dev.

4.26. ArcadeDB Go Quickstart

Connection options for Go

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **Native HTTP driver** — `go get github.com/ArcadeData/arcadedb-drivers/go/arcadedb`, generated from ArcadeDB's OpenAPI contract. Start here for most applications.
- **Native gRPC driver** — `go get github.com/ArcadeData/arcadedb-drivers/go/arcadedbgrpc`, for streaming and bulk inserts.
- **Neo4j BOLT** — `go get github.com/neo4j/neo4j-go-driver/v5`, run Cypher queries.

This tutorial uses the PostgreSQL wire protocol, which is the right choice when you already run a PostgreSQL stack.

This tutorial shows how to connect to ArcadeDB from Go using the PostgreSQL wire protocol and pgx. You will create a graph, query it with SQL and Cypher, and run a vector search.

4.26.1. Prerequisites

- ArcadeDB running with the PostgreSQL protocol enabled (default port 5432)
- Go 1.25+ (required by pgx v5)
- The `pgx` driver

```
go get github.com/jackc/pgx/v5
```

4.26.2. Connect to ArcadeDB

No ArcadeDB-specific SDK is needed. `pgx` connects with an ordinary PostgreSQL connection string:

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/jackc/pgx/v5"
)

func main() {
    ctx := context.Background()
```

```
    conn, err := pgx.Connect(ctx,

"postgres://root:arcadedb@localhost:5432/mydb?sslmode=disable&default_query_exec_mode=
exec")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close(ctx)

    fmt.Println("Connected to ArcadeDB")
}
```

Why `default_query_exec_mode=exec`?

By default, pgx prepares each statement and asks the server for its parameter types. ArcadeDB reports every parameter as text, so pgx then refuses to send a Go `int` or `float64` for it:

```
failed to encode args[0]: unable to encode 26 into text format for
varchar (OID 1043): cannot find encode plan
```

In `exec` mode, pgx sends parameters as text without asking the server first, and ArcadeDB converts them. `simple_protocol` works too. You can also choose the mode for a single call by passing `pgx.QueryExecModeExec` as the first argument after the SQL.

4.26.3. Create a Schema

Use SQL to create vertex and edge types, then insert some data:

```
for _, stmt := range []string{
    "CREATE VERTEX TYPE Person IF NOT EXISTS",
    "CREATE VERTEX TYPE Movie IF NOT EXISTS",
    "CREATE EDGE TYPE Acted IF NOT EXISTS",
    "CREATE VERTEX Person SET name = 'Alice', age = 30",
    "CREATE VERTEX Person SET name = 'Bob', age = 25",
    "CREATE VERTEX Movie SET title = 'The Matrix', year = 1999",
    `CREATE EDGE Acted FROM (SELECT FROM Person WHERE name = 'Alice')
                       TO (SELECT FROM Movie WHERE title = 'The Matrix')
                       SET role = 'Trinity'`,
} {
    if _, err := conn.Exec(ctx, stmt); err != nil {
        log.Fatalf("%s: %v", stmt, err)
    }
}
```

Parameters use PostgreSQL's `$1`, `$2` placeholders:

```
tag, err := conn.Exec(ctx, "CREATE VERTEX Person SET name = $1, age = $2", "Carol",
41)
if err != nil {
    log.Fatal(err)
}
fmt.Println(tag)
```

```
INSERT 0 1
```

4.26.4. Query with SQL

```
rows, err := conn.Query(ctx, `
    MATCH {type: Person, as: person}.outE('Acted'){as: acted}.inV(){type: Movie, as:
movie}
    RETURN person.name AS name, movie.title AS title, acted.role AS role`)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()

for rows.Next() {
    var name, title, role string
    if err := rows.Scan(&name, &title, &role); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s acted in %s as %s\n", name, title, role)
}
if err := rows.Err(); err != nil {
    log.Fatal(err)
}
```

```
Alice acted in The Matrix as Trinity
```

Always check `rows.Err()` after the loop: `rows.Next()` returns `false` both at the end of the result and on an error, and only `rows.Err()` tells the two apart.

4.26.5. Query with Cypher

ArcadeDB supports Cypher queries through the same connection:

```
rows, err := conn.Query(ctx,
    "{cypher} MATCH (p:Person)-[a:Acted]->(m:Movie) RETURN p.name AS name, m.title AS
title, a.role AS role")
if err != nil {
    log.Fatal(err)
```

```
}
defer rows.Close()

for rows.Next() {
    var name, title, role string
    if err := rows.Scan(&name, &title, &role); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s acted in %s as %s\n", name, title, role)
}
```

Prefix Cypher queries with {cypher} when using the PostgreSQL protocol.

4.26.6. Vector Search

Add embeddings to your data and query by similarity:

```
for _, stmt := range []string{
    // A type with a vector property and an index on it
    "CREATE VERTEX TYPE Product IF NOT EXISTS",
    "CREATE PROPERTY Product.embedding IF NOT EXISTS LIST OF FLOAT",
    "CREATE INDEX IF NOT EXISTS ON Product (embedding) LSM_VECTOR METADATA
{dimensions: 4, similarity: 'COSINE'}",

    // Products with embeddings
    "CREATE VERTEX Product SET name = 'Laptop', embedding = [0.9, 0.1, 0.8, 0.2]",
    "CREATE VERTEX Product SET name = 'Tablet', embedding = [0.8, 0.2, 0.7, 0.3]",
    "CREATE VERTEX Product SET name = 'Phone',  embedding = [0.7, 0.3, 0.6, 0.4]",
    "CREATE VERTEX Product SET name = 'Book',   embedding = [0.1, 0.9, 0.2, 0.8]",
} {
    if _, err := conn.Exec(ctx, stmt); err != nil {
        log.Fatalf("%s: %v", stmt, err)
    }
}

// Find the 3 products most similar to a query vector
rows, err := conn.Query(ctx, `
    SELECT name, distance FROM (
      SELECT expand(vector.neighbors('Product[embedding]', [0.85, 0.15, 0.75, 0.25],
3))
    )`)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()

fmt.Println("Similar products:")
for rows.Next() {
    var name string
```

```
    var distance float64
    if err := rows.Scan(&name, &distance); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("  %s (%.4f)\n", name, distance)
}
```

```
Similar products:
  Laptop (0.0025)
  Tablet (0.0029)
  Phone (0.0306)
```

See the vector search tutorial for the index options and similarity functions.

4.26.7. Full Example

```
// ArcadeDB Go quickstart: a small graph queried over the PostgreSQL protocol.
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/jackc/pgx/v5"
)

func main() {
    ctx := context.Background()

    conn, err := pgx.Connect(ctx,

"postgres://root:arcadedb@localhost:5432/mydb?sslmode=disable&default_query_exec_mode=
exec")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close(ctx)

    // Create schema and data
    for _, stmt := range []string{
        "CREATE VERTEX TYPE Person IF NOT EXISTS",
        "CREATE VERTEX TYPE Movie IF NOT EXISTS",
        "CREATE EDGE TYPE Acted IF NOT EXISTS",
        "CREATE VERTEX Person SET name = 'Alice', age = 30",
        "CREATE VERTEX Movie SET title = 'The Matrix', year = 1999",
        `CREATE EDGE Acted FROM (SELECT FROM Person WHERE name = 'Alice')
                           TO (SELECT FROM Movie WHERE title = 'The Matrix')
```

```
                        SET role = 'Trinity'`,
    ) {
        if _, err := conn.Exec(ctx, stmt); err != nil {
            log.Fatalf("%s: %v", stmt, err)
        }
    }

    // Query with SQL MATCH
    rows, err := conn.Query(ctx, `
        MATCH {type: Person, as: person}.outE('Acted'){as: acted}.inV(){type: Movie,
as: movie}
        RETURN person.name AS name, movie.title AS title, acted.role AS role`)
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    for rows.Next() {
        var name, title, role string
        if err := rows.Scan(&name, &title, &role); err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%s acted in %s as %s\n", name, title, role)
    }
    if err := rows.Err(); err != nil {
        log.Fatal(err)
    }
}
```

4.26.8. Next Steps

- PostgreSQL Protocol Reference
- SQL Reference
- Go HTTP driver — the native driver, which adds ArcadeDB's batch loading and vector search APIs

4.27. Connecting from C#

Connection options for C# / .NET

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **Neo4j BOLT** — `Install-Package Neo4j.Driver`, run Cypher queries.
- **PostgreSQL wire protocol** — `Install-Package Npgsql`, run SQL.
- **HTTP/JSON API** — use `HttpClient`, no driver required.

This page covers the HTTP/JSON approach. Per-protocol C# examples for BOLT and PostgreSQL are coming soon.

4.27.1. Sample Code

In C#/.NET 7.0/8.0, HTTP/JSON requests can be made using the `HTTPClient` class inside an async function.

In real situations, the `HTTPClient` object should not be created or discarded on each request. `HTTPClient` should be typically created once in the lifespan of an application, stored in a Singleton Instance or static reference/class, and reused for each request. Here it is created in the function just for simplicity.

The following example demonstrates a simple function which will add a record of the type `Profile` to a database named `mydb` with the name `Alexander`.

```
public async void addProfileName(){
    HttpClient httpClient = new(); //typically instantiate this only once over course
of application, then reuse
    HttpRequestMessage msg = new();
    msg.Method = HttpMethod.Post;
    string authString = "root:arcadedb-password"; //add your password here
    string base64AuthString = Convert.ToBase64String
(System.Text.ASCIIEncoding.ASCII.GetBytes(authString));
    msg.Headers.Authorization = new("Basic", base64AuthString);
    msg.RequestUri = new Uri("http://serveraddress:2480/api/v1/command/mydb"); //add
your server address (or localhost) and db name
    HttpContent httpContent = new StringContent("{ \"language\": \"sql\", \"command\":
\"INSERT into Profile set name = \'Alexander\'\" }", Encoding.UTF8,
"application/json"); //customize command here
    msg.Content = httpContent;
    HttpResponseMessage response = await httpClient.SendAsync(msg);
    string responseString = await response.Content.ReadAsStringAsync();
    Debug.WriteLine("SENT REQUEST, RESPONSE: " + responseString);
}
```

4.28. Connecting from C

Connection options for C

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **PostgreSQL wire protocol** — use `libpq`, run SQL.
- **HTTP/JSON API** — use `libcurl`, no driver required.

There is no actively maintained Neo4j BOLT driver for plain C. If you need Cypher from C, the easiest path is to call out to one of the BOLT-capable languages (Python,

> Go) over an FFI, or use the HTTP/JSON API and target the OpenCypher endpoint.

4.28.1. PostgreSQL Wire Protocol — `libpq`

```
#include <stdio.h>
#include <libpq-fe.h>

int main(void) {
    PGconn *conn = PQconnectdb(
        "host=localhost port=5432 dbname=mydatabase user=root password=playwithdata");
    if (PQstatus(conn) != CONNECTION_OK) {
        fprintf(stderr, "connection failed: %s", PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    PGresult *res = PQexec(conn, "SELECT FROM schema:types LIMIT 5");
    for (int i = 0; i < PQntuples(res); i++)
        printf("%s\n", PQgetvalue(res, i, 0));

    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

Build:

```
gcc example.c -lpq -o example
```

4.28.2. HTTP/JSON API — `libcurl`

```
#include <curl/curl.h>

int main(void) {
    CURL *curl = curl_easy_init();
    curl_easy_setopt(curl, CURLOPT_URL,
        "http://localhost:2480/api/v1/query/mydatabase");
    curl_easy_setopt(curl, CURLOPT_USERPWD, "root:playwithdata");

    struct curl_slist *headers = NULL;
    headers = curl_slist_append(headers, "Content-Type: application/json");
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

    const char *body =
        "{\"language\":\"sql\","
        " \"command\":\"SELECT FROM schema:types LIMIT 5\"}";
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, body);
```

```
    curl_easy_perform(curl);

    curl_slist_free_all(headers);
    curl_easy_cleanup(curl);
    return 0;
}
```

Build:

```
gcc example.c -lcurl -o example
```

4.29. C++

Connection options for C++

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **PostgreSQL wire protocol** — use `libpqxx`, run SQL.
- **HTTP/JSON API** — use `cpp-httplib`, `cpp-restclient`, or `libcurl`, no driver required.

There is no actively maintained Neo4j BOLT driver for C. If you need Cypher from C, the easiest path is to use the HTTP/JSON API or to embed a BOLT-capable runtime (e.g. Python via pybind11).

4.29.1. PostgreSQL Wire Protocol — `libpqxx`

```
#include <pqxx/pqxx>
#include <iostream>

int main() {
    pqxx::connection conn(
        "host=localhost port=5432 dbname=mydatabase "
        "user=root password=playwithdata");

    pqxx::work tx{conn};
    auto result = tx.exec("SELECT FROM schema:types LIMIT 5");

    for (auto const &row : result)
        std::cout << row[0].c_str() << "\n";

    tx.commit();
    return 0;
}
```

Build:

```
g++ -std=c++17 example.cpp -lpqxx -lpq -o example
```

4.29.2. HTTP/JSON API — `cpp-httplib`

```
#include <httplib.h>
#include <iostream>

int main() {
    httplib::Client cli("http://localhost:2480");
    cli.set_basic_auth("root", "playwithdata");

    auto res = cli.Post(
        "/api/v1/query/mydatabase",
        R"({"language":"sql","command":"SELECT FROM schema:types LIMIT 5"})",
        "application/json");

    if (res) std::cout << res->body << "\n";
    return 0;
}
```

Build (header-only library):

```
g++ -std=c++17 example.cpp -o example
```

4.30. Connecting from Rust

Connection options for Rust

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **Neo4j BOLT** — the community `neo4rs` crate, run Cypher queries.
- **PostgreSQL wire protocol** — the `postgres` crate (or the async `tokio-postgres`), run SQL.
- **HTTP/JSON API** — `reqwest` or any HTTP client, no driver required.

4.30.1. Neo4j BOLT — `neo4rs`

`Cargo.toml`:

```
[dependencies]
neo4rs = "0.7"
```

```
tokio = { version = "1", features = ["full"] }
```

```
use neo4rs::*;

#[tokio::main]
async fn main() -> Result<()> {
    let graph = Graph::new("bolt://localhost:7687", "root", "playwithdata").await?;
    let mut result = graph.execute(query("MATCH (n) RETURN n LIMIT 5")).await?;
    while let Ok(Some(row)) = result.next().await {
        println!("{:?}", row);
    }
    Ok(())
}
```

4.30.2. PostgreSQL Wire Protocol — postgres

Cargo.toml:

```
[dependencies]
postgres = "0.19"
```

```
use postgres::{Client, NoTls};

fn main() -> Result<(), postgres::Error> {
    let mut client = Client::connect(
        "host=localhost port=5432 dbname=mydatabase \
         user=root password=playwithdata",
        NoTls,
    )?;

    for row in client.query("SELECT FROM schema:types LIMIT 5", &[])? {
        let value: &str = row.get(0);
        println!("{}", value);
    }
    Ok(())
}
```

4.30.3. HTTP/JSON API — reqwest

Cargo.toml:

```
[dependencies]
reqwest = { version = "0.12", features = ["json", "blocking"] }
serde_json = "1"
```

```rust
use serde_json::json;

fn main() -> Result<(), reqwest::Error> {
    let body = json!({
        "language": "sql",
        "command":  "SELECT FROM schema:types LIMIT 5"
    });

    let res = reqwest::blocking::Client::new()
        .post("http://localhost:2480/api/v1/query/mydatabase")
        .basic_auth("root", Some("playwithdata"))
        .json(&body)
        .send()?
        .text()?;

    println!("{}", res);
    Ok(())
}
```

4.31. Connecting from Ruby

Connection options for Ruby

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **Neo4j BOLT** — the community `neo4j-ruby-driver` gem, run Cypher queries.
- **PostgreSQL wire protocol** — the `pg` gem, run SQL.
- **HTTP/JSON API** — standard library `Net::HTTP` or the `faraday` gem, no driver required.

4.31.1. Neo4j BOLT — `neo4j-ruby-driver`

```
gem install neo4j-ruby-driver
```

```ruby
require "neo4j/driver"

driver = Neo4j::Driver::GraphDatabase.driver(
  "bolt://localhost:7687",
  Neo4j::Driver::AuthTokens.basic("root", "playwithdata"),
  encryption: false
)

driver.session(database: "mydatabase") do |session|
  session.run("MATCH (n) RETURN n LIMIT 5").each do |record|
```

```
    puts record.values
  end
end

driver.close
```

4.31.2. PostgreSQL Wire Protocol — pg

```
gem install pg
```

```
require "pg"

conn = PG.connect(
  host:     "localhost",
  port:     5432,
  dbname:   "mydatabase",
  user:     "root",
  password: "playwithdata"
)

conn.exec("SELECT FROM schema:types LIMIT 5") do |result|
  result.each { |row| puts row.values.first }
end

conn.close
```

4.31.3. HTTP/JSON API — Net::HTTP

```
require "net/http"
require "json"
require "uri"

uri = URI("http://localhost:2480/api/v1/query/mydatabase")

req = Net::HTTP::Post.new(uri, "Content-Type" => "application/json")
req.basic_auth("root", "playwithdata")
req.body = JSON.dump(
  language: "sql",
  command:  "SELECT FROM schema:types LIMIT 5"
)

res = Net::HTTP.start(uri.host, uri.port) { |http| http.request(req) }
puts JSON.parse(res.body)
```

4.32. Connecting from R

Connection options for R

ArcadeDB exposes multiple wire protocols, so you can pick the one your stack already speaks:

- **PostgreSQL wire protocol** — the `RPostgreSQL` package (or the modern `RPostgres`), run SQL.
- **HTTP/JSON API** — the `httr2` or `httr` package, no driver required.

There is no Neo4j BOLT driver for R. The closest option is the community `neo4r` package, which talks to Neo4j over HTTP/Cypher rather than BOLT, so it does not work directly against ArcadeDB. Use the HTTP/JSON API for OpenCypher queries from R.

4.32.1. PostgreSQL Wire Protocol — `RPostgres`

```
install.packages("RPostgres")
library(DBI)

con <- dbConnect(
  RPostgres::Postgres(),
  host     = "localhost",
  port     = 5432,
  dbname   = "mydatabase",
  user     = "root",
  password = "playwithdata"
)

result <- dbGetQuery(con, "SELECT FROM schema:types LIMIT 5")
print(result)

dbDisconnect(con)
```

4.32.2. HTTP/JSON API — `httr2`

```
install.packages("httr2")
library(httr2)

response <- request("http://localhost:2480/api/v1/query/mydatabase") |>
  req_auth_basic("root", "playwithdata") |>
  req_body_json(list(
    language = "sql",
    command  = "SELECT FROM schema:types LIMIT 5"
  )) |>
  req_perform()
```

```
print(resp_body_json(response))
```

4.33. Connecting from Elixir

Connection options for Elixir

The community has not yet released a Neo4j BOLT driver for Elixir, and the PostgreSQL Elixir libraries (`postgrex` / Ecto adapter) introspect Postgres-specific catalogs that ArcadeDB does not implement. The recommended way to talk to ArcadeDB from Elixir is therefore the **HTTP/JSON API** using a library like HTTPoison or `Req`.

If your team has tried BOLT or PostgreSQL from Elixir successfully, we'd love to add an example — please open a discussion.

4.33.1. Configuration

Ensure the Phoenix Elixir framework is installed as per official instructions.

A test project named `testproject` can then be created in a given folder by running `mix phx.new testproject`. Various options during project creation are available.

A package such as HTTPoison must next be added to perform HTTP Requests.

Open the newly created `mix.exs` file and add the line `{:httpoison, "~> 2.2"}` (check current version of the package as indicated on the HTTPoison Package site):

```
defp deps do
    [
      {:phoenix, "~> 1.7.10"},
      # ... other packages
      {:httpoison, "~> 2.2"}
    ]
  end
```

Save and close `mix.exs`. Run `mix deps.get` to update the project and download `HTTPoison` into this project.

To start an interactive prompt enter `cd testserver` and then `iex -S mix phx.server`. This will begin an Interactive Elixir command prompt as indicated by `iex()>`.

4.33.2. Sample Code

A simple HTTP request can then be performed by running the following commands sequentially:

```
userPass = "root:arcadedb-password"
```

```
base64UserPass = Base.encode64(userPass)
authString = "Basic " <> base64UserPass
url = "http://serveraddress:2480/api/v1/command/mydb"
body = Jason.encode!(%{language: "sql", command: "SELECT from Profile"})
headers = [{"Authorization", authString}, {"Content-Type", "application/json"}]
HTTPoison.post(url, body, headers)
```

To process returned data, one can use the following approach:

```
case HTTPoison.post(url, body, headers) do
  {:ok, %{status_code: 200, body: body}} ->
    # do something with the body
    Jason.decode!(body)

  {:ok, %{status_code: 404}} ->
    # do something with a 404
    IO.puts("error404")

  {:error, %{reason: reason}} ->
    # do something with an error
    IO.puts(reason)

end
```

Chapter 5. API & Integration

5.1. API & Integration

5.2. HTTP/JSON API

Overview Endpoints

Action	Method	Endpoint
Get OpenAPI definition	GET	`/api/v1/openapi.json`
Get server status	GET, HEAD	`/api/v1/ready`
Check server liveness	GET, HEAD	`/api/v1/health`
Get server information	GET	`/api/v1/server`
Send server command	POST	`/api/v1/server`
List databases	GET	`/api/v1/databases`
Does database exist	GET	`/api/v1/exists/{database}`
Login (get auth token)	POST	`/api/v1/login`
Logout (invalidate token)	POST	`/api/v1/logout`
List active sessions	GET	`/api/v1/sessions`
Execute a query	GET	`/api/v1/query/{database}/{language}/{query}`
Execute a query	POST	`/api/v1/query/{database}`
Execute database command	POST	`/api/v1/command/{database}`
Begin a new transaction	POST	`/api/v1/begin/{database}`
Commit a transaction	POST	`/api/v1/commit/{database}`
Rollback a transaction	POST	`/api/v1/rollback/{database}`
Batch import vertices and edges	POST	`/api/v1/batch/{database}`

Overview Query & Command Parameters

Parameter	Type	Values
language	Required	`sql`, `sqlscript`, `graphql`, `cypher`, `gremlin`, `mongo`, and others
command	Required	encoded command string
awaitResponse	Optional	set synchronous (`true`, default) or asynchronous (`false`) command
limit	Optional	maximum number of results

Parameter	Type	Values
params	Optional	map of parameters
serializer	Optional	`graph`, `record`, `studio`
autoCommit	Optional	Boolean. `true` to force atomic transaction, `false` to disable, omit for automatic behavior

5.2.1. Introduction

The ArcadeDB Server is accessible from the remote through the HTTP/JSON protocol. The protocol is very simple. For this reason, you don't need a driver, because every modern programming language provides an easy way to execute HTTP requests and parse JSON.

For the examples in this chapter we're going to use `curl`. Every request must be authenticated by passing user and password as HTTP basic authentication (in HTTP Headers). In the examples below we're going to always use "root" user with password "arcadedb-password".

Under Windows (Powershell) single and double quotes inside a single or double quoted string need to be replaced with their Unicode entity representations `\u0022` (double quote) and `\u0027` (single quote). This is for example the case in the data argument (`-d`) of POST requests.

5.2.2. Token-Based Authentication

In addition to HTTP Basic Authentication, ArcadeDB supports token-based authentication. Wrong Basic credentials, an unknown user and a user locked out after too many failed attempts are answered `401` "Authentication failed", the same status the bearer-token and API-token paths use. So are a malformed Basic header and an unsupported `Authorization` scheme. `403` is the answer for an authenticated user who lacks permission. These answers carry no `WWW-Authenticate` header, so browsers do not open their native login dialog over Studio's login form. *(Since v26.11.1: before, these cases answered `403` "Security error".)* A request with no `Authorization` header at all is also answered `401` "Authentication failed" *(since v26.12.1: before, its error message was empty)*. That one answer carries a `WWW-Authenticate: Basic` challenge: it lets a browser tab sign in to the Swagger UI page (`/api/v1/docs`) through its native login dialog, and lets clients that authenticate by answering a challenge (such as the JDK `Authenticator` or `curl --anyauth`) send their credentials.

This allows you to login once with credentials and receive a token that can be used for subsequent requests, avoiding the need to send username and password with every request.

This is particularly useful when:

- Building applications that make multiple API calls
- You want to avoid storing credentials in your application
- You need to manage user sessions with explicit logout capability

Authentication Flow:

1. Call `POST /api/v1/login` with Basic Auth credentials to obtain a token

2. Use the token in subsequent requests via `Authorization: Bearer <token>` header
3. Call `POST /api/v1/logout` to invalidate the token when done

Example workflow:

```
# 1. Login and get a token
$ curl -X POST http://localhost:2480/api/v1/login \
       --user root:arcadedb-password

# Response: {"token":"AU-ArcadeDB_0-2af64e60-8455-423a-bc64-
ed0e19729f04","user":"root"}

# 2. Use the token for subsequent requests
$ curl http://localhost:2480/api/v1/query/mydb/sql/select%201 \
       -H "Authorization: Bearer AU-ArcadeDB_0-2af64e60-8455-423a-bc64-ed0e19729f04"

# 3. Logout when done
$ curl -X POST http://localhost:2480/api/v1/logout \
       -H "Authorization: Bearer AU-ArcadeDB_0-2af64e60-8455-423a-bc64-ed0e19729f04"
```

Authentication tokens expire after a configurable period of inactivity (default is 30 minutes). See `server.httpAuthSessionExpireTimeout` to configure this timeout.

Token revocation:

Deleting a user or changing its password immediately invalidates every token already issued to that user, on every server of a cluster. Previously such a token kept working until it expired on its own. This applies however the change is made: the `/api/v1/server/users` endpoints, the `create user` / `drop user` server commands, the Cypher `CREATE USER` / `ALTER USER … SET PASSWORD` / `DROP USER` statements, or an edit to `server-users.jsonl`.

Tokens in a cluster:

A token issued by one server of a cluster is honoured by every other server, so a load balancer in front of the cluster needs no sticky sessions. The token names the server that issued it, `AU-<server name>-<uuid>`, where the server name is the value of `arcadedb.server.name` (any character outside letters, digits, `.`, `_` and `-` is replaced with `_`). A server that receives a token it has never seen asks the issuer once, through the cluster-internal channel protected by the cluster token, and keeps a local copy for the following requests. The copy is confirmed with the issuer periodically (a third of `server.httpAuthSessionExpireTimeout`, at most every 60 seconds), which also counts as activity on the issuer, so a token in use anywhere in the cluster does not expire at its source. A logout served by any server drops the copies on every other server. If the issuer no longer holds the session (it expired there, or the server restarted) the copies are dropped at their next confirmation; if the issuer cannot be reached, a copy is served for at most one idle timeout and then dropped.

Every server must be able to map the name in the token to a member of the cluster. This is the same rule a server uses to find itself in `arcadedb.ha.serverList`: a `name@host:port` entry, a host equal to the server name, or a `-N` / `_N` suffix giving the position in the list (the Kubernetes Helm chart satisfies it as-is). A token whose name matches no member is refused without asking anyone. Tokens in the previous `AU-<uuid>` form are still accepted, but only by the server that issued them.

API tokens (`at-` prefix) work the same way on every server: they are replicated to the whole cluster, like users and groups (see Cluster security documents).

Number of concurrent tokens:

The server bounds how many authentication tokens it keeps. A single user may hold up to `server.httpAuthSessionMaxPerUser` tokens (default `100`); past that, its own oldest token is invalidated to make room for the new one, and no other user is ever affected. Across all users the server keeps up to `server.httpAuthSessionMax` tokens (default `10000`); expired tokens are reclaimed first, and a login that still finds no room is answered `503`. Set either setting to `0` for the previous unlimited behaviour (not recommended: an unbounded number of sessions is a memory-exhaustion risk).

The client metadata stored with a session (`User-Agent`, `CF-Connecting-IP`, `X-Forwarded-For`, `CF-IPCountry`, `CF-IPCity`) is truncated to 256 characters.

5.2.3. Server-Side Transactions

ArcadeDB implements server-side transaction over HTTP stateless protocol by using sessions. A session is created with the /begin request and returns a session id in the response header (example `arcadedb-session-id: AS-ee056170-dc9b-4956-8d71-d7cfa01900d4`). Use the session id in the request header of further commands you want to execute in the same transaction and request /commit to commit the server side transaction or /rollback to rollback the changes. After a period of inactivity (default is 5 seconds, see `server.httpSessionExpireTimeout`), the server automatically rolls back and purges expired transactions.

If a request names a session id the server can no longer resolve - because it expired, or was already committed or rolled back - a **write** is refused with `404`, while a **read** still answers, running outside any transaction. A read that degraded this way says so: the response carries an `arcadedb-session-expired` header holding the id that could not be resolved, so a client can tell it apart from a read that ran inside its transaction. This applies to `GET /query`, `/begin`, `/commit`, `/rollback` and the time-series and Prometheus read endpoints.

The time-series and observability endpoints accept a session id so they run under that session's user and keep it from expiring while you use it, but they do **not** take part in its transaction: samples are durable as soon as they are written, and the reads only read. An error on one of them therefore leaves your open transaction exactly as it was. Before, it rolled the transaction back while leaving the session alive, so a later `/commit` reported success and the work was silently gone. The endpoints this covers:

- the three time-series endpoints, `/api/v1/ts/{database}/write`, `/query` and `/latest`;

- the PromQL endpoints under `/api/v1/ts/{database}/prom`, and the Prometheus `remote_read` and `remote_write` endpoints;
- the Grafana query, metadata and health endpoints.

The two errors easiest to hit on the observability routes are a malformed `start`/`end` timestamp and a read whose answer exceeds `arcadedb.server.maxResultRows`; neither touches your transaction any more.

Since v26.10.2 the same holds for `GET /api/v1/query` and `POST /api/v1/query`. They still run inside the session's transaction, so they see what you wrote in it, but they only read, so an error on them leaves your open transaction as it was: a `413` for an answer over `arcadedb.server.maxResultRows`, or a `400` for a request refused before the query ran (for example `Accept: application/x-ndjson` on a statement that cannot be streamed). `POST /api/v1/command` also leaves the transaction alone for a `400` of this kind, which includes a malformed request body: a missing `command`, a field of the wrong type (such as a numeric `language`), a `limit` that is not an integer, or a typed parameter marker that cannot be decoded. An error raised by the statement itself on `/api/v1/command` still rolls the transaction back, as described under Commit a transaction.

5.2.4. Transaction Scripts

In case SQL (`sql`) is supposed to be used as language for a transactions, the language variant SQL Script (`sqlscript`) is also available. A `sqlscript` can consist of one or multiple SQL statements, which is collectively treated as a transaction. Hence, for such a batch of SQL statements, no `begin` and `commit` commands are necessary, since `begin` and `commit` implicitly enclose any `sqlscript` command.

5.2.5. Streaming Change Events

This feature only works on the server where changes are executed. In a replicated environment, the changes executed on other servers would not fire events on all the servers in the cluster, but only on the local server. The cluster support is coming soon.

The Java API supports real-time change notifications, which the HTTP API implements via a websocket. You can opt into notifications for all changes that occur on a database, or filter by the operation (i.e. create, update, delete) or underlying entity type.

To connect, point your favorite WebSocket client to the `ws://SERVER:PORT/ws` endpoint. You will need to authenticate with HTTP Basic, which for some clients (like most browsers) is only possible via the URI, like this: `ws://USERNAME:PASSWORD@SERVER:PORT/ws`. Others will require that you set the Authorization header directly. Check the documentation for your client of choice for details.

To subscribe/unsubscribe to change events, send JSON messages using the following structure:

Property	Required	Description
action	**Required**	`subscribe` or `unsubscribe`.

Property	Required	Description
database	**Required**	The database name.
type	**Optional**	The entity type to filter by.
changeTypes	**Optional**	Array of change types you'd like to receive. Must be `create`, `update`, or `delete`.

Example: to subscribe to all changes (create, update, delete) for the type `Movie` in the database `movies`, use:

```
{"action": "subscribe", "database": "movies", "type": "Movie"}
```

If instead, you only want updates, send:

```
{"action": "subscribe", "database": "movies", "type": "Movie", "changeTypes":
["update"]}
```

If you want every change on the database (use with caution!):

```
{"action": "subscribe", "database": "movies"}
```

Once subscribed, you will get JSON messages for any matching changes with the following properties:

Property	Description
database	The source database.
changeType	create, update or delete.
record	The full record that generated the change event.

A subscription is not permanent. The server re-checks on every event that the connected user is still allowed to read the database: if that access is revoked, or the user is deleted, the subscription is dropped and the connection closed instead of continuing to stream.

A subscription is to the database NAME. Starting from v26.11.1, it follows the database the server serves under that name: after `close database` and `open database`, a restore, an HA snapshot install, or a drop followed by a create of the same name, the existing subscribers receive the changes of the database opened again, with no new `subscribe`. While the database is closed, nothing is sent. Before v26.11.1 a subscription made before a `close database` stayed silent after the `open database`, and so

did every new one, until the last subscriber of the name had left.

A subscriber is also expected to keep reading. Events are sent asynchronously, so a client that stops consuming them accumulates unsent frames in the server's memory. Once more than `server.eventBusMaxPendingBytes` (16 MB by default) is outstanding towards one connection, that subscription is dropped and the connection closed. Set it to `0` to disable the limit.

In `production` server mode an `error` frame answering a failed action carries "The request failed. Check the server log for the details" instead of the engine's message, which is written to the server log (since v26.10.2). See Error details in production mode.

5.2.6. Streaming Inserts over WebSocket

The same `ws://SERVER:PORT/ws` connection can run a duplex insert session: the client sends records in chunks and gets one acknowledgement per chunk while it is still sending the next, and it decides when the session commits. This is the WebSocket counterpart of the gRPC `InsertBidirectional` RPC. Each frame is a JSON object whose `action` is one of:

Action	Description
start	Opens a session on `database`. Optional `sessionId` (the server generates one when absent) and `options`, see below. Answered with `started`, which echoes the modes the session runs under.
chunk	`sessionId`, `chunkSeq` (contiguous from 1; re-sending an applied chunk is acknowledged as a replay and not applied again) and `records`, an array of JSON objects. A record takes its type from `@class` or from the session's `targetType`; an edge names its endpoints with `@from` and `@to` (`out` / `in` are accepted too). Answered with `batchAck`.
commit, rollback	Ends the session. Answered with `committed`, whose `outcome` says which of the two it was and whose `summary` carries the totals of the whole session.

The `options` object of `start` accepts:

Option	Description
targetType	Default type of the records, overridable per record with `@class`.
transactionMode	`per_stream` (default, also spelled `per_request`): one transaction for the whole session, committed or rolled back by the client. `per_batch`: one transaction per chunk, committed before its acknowledgement. `per_row`: one transaction per record. Under the last two, `committed` reports `partialCommit: true` because a `rollback` cannot take back what was already committed.
conflictMode	What to do with a record whose key is already taken: `error` (default) reports it as a failed row with the error code `CONFLICT`; `update` rewrites the existing record with the incoming values; `ignore` drops it; `abort` is accepted for parity with gRPC and behaves as `error`. The gRPC spellings (`CONFLICT_UPDATE`, …) are accepted as well.

Option	Description
keyColumns	The properties an existing record is matched on. Required for `update`. When absent, a duplicate is found by the engine where the transaction commits: per row under `per_row`, per chunk under `per_batch`, and on the `commit` frame under `per_stream`, which is then answered with an error and closes the session.
updateColumnsOnConflict	With `update`, the only properties overwritten. When absent, every non-key property sent is merged; a null value never overwrites, and an edge's endpoints are never changed.
validateOnly	`true` to parse and count every record without writing anything.

The `batchAck` frame carries `chunkSeq`, the counters `received`, `inserted`, `updated`, `ignored`, `failed`, and an `errors` array with `rowIndex`, `code`, `message` and `exception` for each record that could not be applied; the rest of the chunk still goes in. An entry with `rowIndex -1` reports a failure of the chunk's transaction rather than of one record: nothing in the chunk was applied. A session the client walks away from - the connection drops, or no frame arrives for `server.wsInsertSessionExpireTimeout` milliseconds - is rolled back, and the idle case is reported with an unsolicited `error` frame.

The `code` of an `errors` entry is one of:

Code	Meaning
CONFLICT	The record's key is already taken (`conflictMode error` or `abort`), or the commit of the chunk found a duplicated key.
NOT_LEADER	*(Since v26.10.2.)* The node is a follower of a replicated cluster and refused the write. The entry carries a `leaderAddress` field with the address of the leader to send the load to.
DB_ERROR	Any other failure.

`leaderAddress` is a field of its own, outside `message`, so production mode does not hide it. A `per_stream` session whose `commit` frame is refused by a follower is answered with an `error` frame whose `error` is "Not the leader" and which carries `leaderAddress` too (since v26.10.2). Since v26.12.1 a `start` or `commit` frame refused for a reason that clears by itself (on an HA server, a database being replaced from the leader's snapshot or quarantined) is answered with `"error": "Retry later"` and the reason in `detail`, and a load that committed cluster-wide but failed to apply locally with an `error` that ends in "do not retry".

Terminating a chunk. *(Since v26.10.2.)* While a chunk runs it is listed by the `list queries` server command, in every transaction mode, and `terminate query` stops it. The termination undoes what the session's mode allows: under `per_stream` the whole session's transaction is rolled back, chunks acknowledged before included; under `per_batch` the running chunk's transaction is rolled back and chunks acknowledged before stay committed; under `per_row` the rows committed before the termination stay and the rest are not applied. A session joined to an HTTP transaction (`transactionMode none`) rolls that transaction back and ends it. In every case the session is closed and the server sends an `error` frame that says what became of the chunk; the connection stays open and can `start` a new session.

Security. *(Since v26.10.2.)* The user that opened the connection is checked again on every frame, not only at the handshake:

- a user that was dropped or given a new password has its session rolled back, receives an `error` frame, and the connection is closed (close code `1008`, "Principal is no longer valid");
- a user whose grant on the session's database was revoked has its `chunk` and `commit` frames refused and the session rolled back; a `rollback` frame is always accepted;
- a session joined to an HTTP transaction (`transactionId`) that a failed command has rolled back is refused, instead of writing its records outside the transaction, where they would autocommit.

The counters are what the engine wrote, not what the session handed it. A record that a `BeforeRecordCreateListener` or a `BeforeRecordUpdateListener` vetoed was refused on purpose, so it is neither `inserted` or `updated` nor `failed`: it is counted as `ignored`, the same way the gRPC loader counts it. Under `conflictMode: update` a vetoed update does not fall through to inserting a second record for the key. A client that resumes a load from these counters can therefore trust `inserted + updated` to be the number of records that are durable.

Before v26.11.1 a record a listener vetoed was counted as `inserted` or `updated`, and a vetoed edge was answered as a `DB_ERROR` row carrying the text of a `NullPointerException`. A load that was driven off the counters may have marked a chunk as done although the engine had refused some of its records; compare `inserted + updated` with the number of records the chunk was expected to write.

In `production` server mode the engine's text in an `error` frame and in the `message` of an `errors` entry (for a conflict, the key values the row carried) is replaced by "The request failed. Check the server log for the details", and the detail is written to the server log. The `code` and the exception class are kept. Text the protocol writes about the request itself, such as a missing `keyColumns`, an invalid option or an expired session, stays visible. Before v26.10.2 the message was sent as is in every mode. See Error details in production mode.

A `/ws` connection is allowed `server.wsMaxControlFrameSize` (64 KB by default) per text frame until it has an insert session, and `server.wsMaxInsertFrameSize` (16 MB) while it has one, so a `chunk` can carry a whole batch while a connection that never inserts stays on the small budget. A frame over the budget in force closes the connection with a `1009` close code. A separate cap, `server.wsMaxInsertChunkRows` (100,000), bounds the number of records in one `chunk` after parsing; a chunk over it is refused with an `error` frame and leaves the session open, so the client can split the batch and resend it under the same `chunkSeq`.

Example, upserting people by `email`:

```
{"action": "start", "database": "crm", "options": {"targetType": "Person",
"conflictMode": "update", "keyColumns": ["email"]}}
{"action": "chunk", "sessionId": "...", "chunkSeq": 1, "records": [{"email": "a@x.io",
"name": "Ann"}, {"email": "b@x.io", "name": "Bob"}]}
{"action": "commit", "sessionId": "..."}
```

5.2.7. Responses

The server answers a HTTP request with a response. This response can have a body, which will always be in the JSON format. Generally, a successful response (HTTP status codes 2xx) contains a `result` field, while an erroneous request (HTTP status code 4xx) has an `error` field, and a server error (HTTP status code 5xx), in addition to the `error` field, provides a `detail` and an `exception` field.

A request body that is sent but cannot be parsed as JSON (a truncated object, a scalar such as `123`, text that is not JSON) is refused with a `400` and `"error": "Invalid JSON payload"`, whichever route it was sent to. The `detail` names where the parser stopped, for example `End of input at line 1 column 13`; it never repeats the body. *(Since v26.11.1: before, a body that could not be parsed was treated by most routes as if no body had been sent, and `POST /api/v1/server` answered a `500`. The cluster routes that took a body as optional, such as the security seed and the bootstrap state, now refuse an unparsable one too.)* A blank body is no body, and is not refused. The routes whose body is not JSON keep their format: the time series line protocol write, the Prometheus `remote_read` and `remote_write`, and the batch import. The MCP endpoint is the exception to the refusal: it answers the JSON-RPC parse error `-32700` over HTTP `200`.

A route that answers `GET` also answers `HEAD`, with the status and the headers of the `GET` and no body, as RFC 9110 requires, so a container healthcheck such as `wget --spider` works against any of them. A `HEAD` runs the `GET` handler in full and the connector discards the body, so it costs what the `GET` costs: a probe should target a cheap route such as `/api/v1/ready`. A request with a method the route does not answer is refused with a `405` whose `Allow` header names the methods it does answer, for example `Allow: GET, HEAD` on `/api/v1/databases`. *(Since v26.11.1: before, only `/api/v1/ready` and `/api/v1/health` answered `HEAD`, and a `405` from a core route carried no `Allow` header, issue #10531.)*

A request to a path the server does not serve, such as `/api/v1/nosuchroute`, is answered `404` with `"error": "The requested resource does not exist"`, a `requestId` field and the `X-Request-Id` and `X-ArcadeDB-Replay-Protection` headers every other answer carries. The message never repeats the path that was asked for. *(Since v26.12.1: before, such a request was answered `404` with an empty body and none of these headers, issue #12410.)*

5.2.8. Retrying Requests Safely (`X-Request-Id`)

Every request can carry an `X-Request-Id` header. The server echoes it on the response and logs it with the request (it generates one when the request has none), which is what correlates a failure with the server log.

On a `POST` it also makes a retry safe to send verbatim. A successful (`2xx`) response is kept for `arcadedb.ha.idempotencyCacheTtlMs` milliseconds, keyed by the id together with the method, path, database and body, and bound to the authenticated user. An identical retry is answered from it instead of executing the request a second time, so a client that lost the answer to a write, a restore or an import can simply send it again with the same id.

- A failed request is not kept: its retry executes afresh.
- While the first request is still executing, an identical retry waits briefly for it and then answers `409 Conflict` with a `Retry-After` header, naming `RequestStillInFlightException`. The request was not executed again: retry it later with the same id to receive the result of the execution in progress.

- A `restore database`, `restore backup` or `import database` asked for as an SSE progress stream (`Accept: text/event-stream`) is replayed as a one-event stream carrying its `completed` event, since a finished stream cannot be sent again. A retry without that header receives the JSON answer instead, whichever way the first request asked.
- Not replayed: a request inside a client-managed transaction (it carries `arcadedb-session-id`), a request asking for an NDJSON stream, a response larger than `arcadedb.ha.idempotencyCacheMaxBodyBytes` bytes, and the bulk-load route `POST /api/v1/batch/{database}`, whose body is streamed rather than buffered and so cannot be part of the key.

Every response carries the header `X-ArcadeDB-Replay-Protection: true`, which tells a client that this server answers a repeated id from that cache. The remote Java driver (`RemoteDatabase`, `RemoteServer`) sends one `X-Request-Id` per `command` call, outside a transaction, and once it has seen that header it sends the same call again, with the same id, to the same server when the response was lost, instead of failing with "the server may already have applied it". The retry names the server process in `X-ArcadeDB-Replay-Instance`: a server that restarted, or that no longer holds the answer (too large to keep, expired or evicted), answers `412` and does not run the request again.

Use a new id for every distinct request. The OpenAPI document declares the header and the `409` on every `POST` operation that honours them.

5.2.9. Tutorial

Let's first create an empty database "school" on the server:

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "create database school"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Now let's create the type "Class":

```
$ curl -X POST http://localhost:2480/api/v1/command/school \
       -d '{"language": "sql", "command": "create document type Class"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

We could insert our first Class by using SQL:

```
$ curl -X POST http://localhost:2480/api/v1/command/school \
       -d '{"language": "sql", "command": "insert into Class set name = '\''English'
\'', location =  '\''3rd floor'\''"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Or better, using parameters with SQL:

```
$ curl -X POST http://localhost:2480/api/v1/command/school \
       -d '{"language": "sql", "command": "insert into Class set name = :name,
location = :location", "params": {"name": "English", "location": "3rd floor"}}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

5.2.10. Reference

Get OpenAPI Definition (GET)

Returns a JSON document containing the HTTP API's OpenAPI definition.

URL Syntax: `/api/v1/openapi.json`

Response:

- `200` OK

Example:

```
$ curl -X GET "http://localhost:2480/api/v1/openapi.json"
```

Return:

```
{"openapi":"3.0.3", ... }
```

Check if server is ready (GET, HEAD)

Returns a header-only (no content) status about if the ArcadeDB server is ready.

URL Syntax: `/api/v1/ready`

This endpoint accepts GET or HEAD requests without authentication, and is useful for remote monitoring of server readiness. Use it as a Kubernetes **readiness** probe. HEAD is supported so tools that check health with a HEAD request (e.g. `wget --spider`) work out of the box.

Responses:

- `204` OK — server is `ONLINE`
- `503` server not started yet

When `arcadedb.server.readinessRequiresHA=true` and HA is active, readiness additionally requires the node to have joined the Raft group: it returns `503` (`Node has not yet joined the Raft group`) until a leader has been elected. Once a leader is known, a second gate requires the node to be a member of the current Raft configuration and, for a follower, to have replayed the committed log to within `arcadedb.server.readinessHAMaxLag` entries of the commit index; until then it returns `503` (`Node is not yet in the Raft configuration or has not caught up`). This keeps a (re)joined follower with a

wiped or lagging log out of traffic during a rolling restart, so the write quorum is not dropped. The default (`false` for `readinessRequiresHA`) preserves the behavior above. See Health probes.

Example:

```
$ curl -I -X GET "http://localhost:2480/api/v1/ready"
```

Return:

```
HTTP/1.1 204 OK
```

Check server liveness (GET, HEAD)

Returns a header-only (no content) liveness status. Use it as a Kubernetes **liveness** probe.

URL Syntax: `/api/v1/health`

This endpoint accepts GET or HEAD requests without authentication and performs no database I/O. It returns `204` whenever the server process and HTTP layer are up; unlike readiness it never reports a busy state, so a warming-up node is not restarted. The one exception: on a High Availability cluster, a node that has given up recovering from a crash loop returns `503`, so the orchestrator restarts it. See Health probes.

Response:

- `204` OK — server process and HTTP layer are up
- `503` Service Unavailable — (HA only) the node is stuck in an unrecoverable crash loop and needs a restart

Example:

```
$ curl -I -X GET "http://localhost:2480/api/v1/health"
```

Return:

```
HTTP/1.1 204 OK
```

Get server information (GET)

Returns the current configuration.

URL Syntax: `/api/v1/server`

The following `mode` query parameter values are available:

- `basic` returns minimal server information.

- `default` returns full server configuration (default value when no parameter is given).
- `cluster` returns cluster layout.

In `default` mode the `settings` array reports, for each server-level setting, the value this server is actually running on in `value` and the built-in default in `default`, with `overridden` telling you whether the two differ because the setting was given an explicit value - in the server configuration file, by an embedding application, or at runtime with `set server setting`. Secrets are masked: settings that are secrets in their entirety (the HA cluster token, any key containing `password`) report `*****`, and `arcadedb.server.defaultDatabases`, whose value merely **embeds** credentials, keeps its database names, user names and groups while each password is replaced.

Responses:

- `200` OK
- `401` invalid credentials

Example:

```
$ curl -X GET "http://localhost:2480/api/v1/server?mode=basic" \
       --user root:arcadedb-password
```

Return:

```
{"version": "26.11.1", "serverName": "ArcadeDB_0"}
```

Send server command (POST)

Sends control commands to server.

URL Syntax: `/api/v1/server`

The following commands are available:

- `list databases` returns the list of databases installed in the server
- `create database <dbname>` creates database with name `dbname`
- `drop database <dbname>` deletes database with name `dbname`
- `open database <dbname>` opens database with name `dbname`
- `close database <dbname>` closes database with name `dbname`. Refused while a backup, restore, import, export or another close of the same database is running, so it cannot pull the database out from under one; retry once that operation finishes
- `create user { "name": "<username>", "password": "<password>", "databases": { "<dbname>": "admin", "<dbname>": "admin" } }` creates user credentials `username` and `password` and admin access to databases `dbname`.
- `drop user <username>` deletes user `username`

- `get server events [<filename>]` returns a list of server events, optionally a filename of the form `server-event-log-yyyymmdd-HHMMSS.INDEX.jsonl` (where `INDEX` is a integer, i.e. `0`) can be given to retrieve older event logs
- `shutdown [<serverName>]` kills a server gracefully: this one when no name is given, otherwise the named peer of the HA cluster. See Shutdown server.
- `set server setting <key> <value>` sets the server setting with `key` to `value`, see the list of server-level settings
- `set database setting <dbname> <key> <value>` sets the database's <dbname> with `key` to `value`, see the list of database-level settings
- `connect cluster <address>` connects this server to a cluster with `address`
- `disconnect cluster` disconnects this server from a cluster
- `align database <dbname>` aligns database `<dbname>`, see the associated SQL command
- `restore database <dbname> <url>` restores a database from a backup file. The URL can be a local path (`file://`), HTTP/HTTPS URL, or classpath resource. The database must not already exist. See how-to/operations/restore.pdf.
- `import database <dbname> <url>` creates a new database and imports data from the given URL (OrientDB export, GraphML, GraphSON, CSV, JSON). Combines `create database` and `IMPORT DATABASE` in a single command. See how-to/migration/importer.pdf.
- `list queries [database <dbname>] [tag <label>]` lists the statements the server is running. See List and terminate running queries.
- `terminate query <id>` and `terminate queries tag <label>` stop running statements and say whether they stopped. See List and terminate running queries.
- `list transactions` lists the open transaction sessions, and `terminate transaction <sessionId>` rolls one back and ends it. See List and terminate running queries.

Both `restore database` and `import database` support **SSE progress streaming**: send `Accept: text/event-stream` in the request headers to receive real-time progress events instead of waiting for the response. With an `X-Request-Id`, a retry of a restore or import that completed is replayed rather than run again, in either encoding (see Retrying Requests Safely (`X-Request-Id`)).

Only **root** users can run these command, except the `list databases` command, which every user can run, and this user's accessible databases are listed, and the commands on running work (`list queries`, `terminate query`, `terminate queries`, `list transactions`, `terminate transaction`), which every user can run on their own work.

Responses:

- `200` OK
- `400` invalid command
- `401` invalid credentials
- `404` the command names a database this server does not have

- `503` the database is there and cannot serve the command now: closed by `close database`, being restored, being dropped, or being recovered by HA (see a database that is not available)
- `409` the name is taken: `create database` on an existing database (`"error": "Database already exists"`) or `create user` on an existing user (`"error": "User already exists"`) *(Since v26.10.2)*
- `500` invalid JSON request body

A command naming a database that does not exist - `open database nosuchdb`, for example - answers `404` instead of `500`. Naming a database the server has not got is a mistake in the request, not a server fault. Adjust any monitoring that alerts on `5xx` from this endpoint. Since 26.11.1 a database that is there and held - closed, being restored or dropped - answers `503`, not `404` (see a database that is not available).

Databases opened on demand. Since v26.11.1 a database that exists on disk but is not open on the server, because `arcadedb.server.databaseLoadAtStartup` is `false`, is opened by the first HTTP request that names it (query, command, batch load, WebSocket insert session), as the Postgres, BOLT and gRPC protocols already did. Until v26.10.2 the HTTP API answered `Database '<dbname>' is not available` until the database was opened explicitly with `open database`. Only a directory that really holds a database (one with a `schema.json` or `schema.prev.json` file) is opened: any other name is refused as before, and a request never creates a directory. The user must still be authorized on the database. This makes `databaseLoadAtStartup=false` practical for a server that hosts many databases: it starts without opening any of them, and each one is opened by its first request (issue #10234).

A database closed with the `close database` server command is the exception: it was closed on purpose, for a restore of its files or a cold backup, so an HTTP request does not reopen it. It stays closed until `open database` (or a restore, an import or a drop of that name). The server remembers this in memory only and for the node that received the command: after a restart the database is opened again, at start or on its first request.

The same rule holds for the other protocols. A database that a `close database` command closed, or that an `overwrite` restore of its name is about to replace, is not opened by a Postgres, BOLT, gRPC, Redis or MongoDB connection, or by an MCP or Gremlin request, that names it: the client is told that the database is not available, as an HTTP request is (`503` for HTTP, SQLSTATE `57P03` for Postgres, status `UNAVAILABLE` for gRPC, the transient `Neo.TransientError.Database.DatabaseUnavailable` for BOLT, a `LOADING` error reply for Redis, error `365 TemporarilyUnavailable` for MongoDB; an engine error category of its own, `UNAVAILABLE`, so that none of them reports it as a server fault). A Redis key `<database>.<key>` whose database is held is refused the same way, and not kept as a plain key of the connection. *(Since v26.11.1: before, these protocols opened it, so a cold backup copied a database that was open and being written to, and the writes a client made during an `overwrite` restore were acknowledged and then discarded when the restored copy replaced the database, issue #10533.)* A database that is already open keeps serving reads while an `overwrite` restore replaces it, and refuses every write until the restore ends, on every protocol and also on a connection opened before the restore started: the write is answered as "not available", with the codes above, and a database setting change is refused the same way. The restore first waits for the commits already running. *(Since v26.11.1: before, those writes were acknowledged and then lost with the database the restore replaced, issue #10689.)* A name that is not a database at all is the client's to correct, and every protocol says so rather than reporting a server fault: SQLSTATE `3D000` for Postgres, status `NOT_FOUND` for gRPC, `Neo.ClientError.Database.DatabaseNotFound` for BOLT, error `26`

`NamespaceNotFound` for MongoDB, `ERR Database '<name>' not found` for Redis, `404` for HTTP (an engine error category of its own, `DATABASE_NOT_FOUND`, since v26.11.1, issue #10795). A connection that keeps naming a database across a `close database` + `open database`, a `drop database` + `create database` or an `overwrite` restore is served by the new database as soon as it is there (issue #10794).

Since v26.11.1, HTTP tells the two apart as every other protocol does (issue #10872): a name that is no database answers `404` with `"error": "Database not found"` and the exception `com.arcadedb.exception.DatabaseNotFoundException`; a database that is there and cannot serve the request now answers `503` with the exception `com.arcadedb.exception.DatabaseNotAvailableException`, an `error` naming the state, and the state in `exceptionArgs`, so a client is told it in production mode too, where the `detail` is not sent:

exceptionArgs	error	when
CLOSED	Database is closed	a `close database` command closed it; it stays closed until `open database`
RESTORING	Database is being restored	an `overwrite` restore holds its name and the database is not open
WRITES_REFUSED	Database does not accept writes	an `overwrite` restore holds it while it is open: reads still answer `200`
DROPPING	Database is being dropped	a `drop database` is deleting its files
RECOVERING	Database is being recovered	on a replicated cluster, a snapshot install or a resync has to reconcile this node's copy first
UNAVAILABLE	Database is not available	any other reason it is not open for this request, such as a server that is starting or stopping

No `Retry-After` is sent: how long an operator keeps a database closed is not something the server knows. The Java remote client rebuilds both exceptions, with the state as `DatabaseNotAvailableException.getReason()`, and does not retry either of them.

Before v26.11.1 these states answered `404 Database not found`, the answer of a name that is no database, with the same exception class, so the two bodies were identical in production mode. A client or a monitor that treated a `404` from a database endpoint as "the database does not exist" now sees `503` for a database that exists and is held.

By default an opened database stays open until the server stops or `close database` closes it, and there is no limit on how many the requests can open: each one holds its data files, its caches and, once written, a WAL file (since v26.11.1 a database holds no thread of its own and no WAL file until it is written, and a WAL file nobody writes for `arcadedb.txWalIdleTimeout` is closed). Size the memory and the file descriptor limit of a server that hosts thousands of databases for the ones in use at the same time. Since v26.11.1 the server can also close the databases that fall idle, and cap how many stay open, with `arcadedb.server.databaseIdleTimeout` and

> `arcadedb.server.maxOpenDatabases`: a database closed that way is reopened by its next request of any protocol, unlike one closed with `close database`. See Closing idle databases.

Examples:

List databases

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "list databases"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": ["school", "mydatabase"]}
```

Create database

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "create database mydatabase"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

If a database with that name already exists, the command fails with `409` and `"error": "Database already exists"`; the label stays readable in production mode, where `detail` is concealed. *(Since v26.10.2)* Restoring a backup into a name that is taken (`restore database`, or `restore backup` without overwrite) still answers `400`.

Drop database

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "drop database mydatabase"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Open database

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "open database mydatabase"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Close database

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "close database mydatabase"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Restore database

Restores a database from a backup file (local or remote). This is the fastest way to provision a database from a backup archive.

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "restore database mydatabase
https://example.com/backups/mydatabase.zip"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Restoring from a local file:

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "restore database mydatabase file:///path/to/mydatabase-
backup.zip"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Create user

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "create user {\"name\": \"myuser\", \"password\":
\"mypassword\", \"databases\": {\"mydatabase\": \"admin\"}}"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

If a user with that name already exists, the command fails with `409` and `"error": "User already exists"`, on a single server and on a cluster alike; `POST /api/v1/server/users` answers the same. *(Since v26.10.2)*

Drop user

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "drop user myuser"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Shutdown server

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "shutdown"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Shutting down another node of an HA cluster

Naming a server after the command stops that peer instead of the one receiving the request. The

command is relayed over the cluster's own transport — encrypted when the cluster has TLS configured for its side channels, see TLS/SSL for peer-to-peer transfers:

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "shutdown arcadedb-1"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

The name is matched as a substring against each peer's identifier and its HTTP address, so a partial name is enough:

- **No peer matches** it — the request fails with `Cannot find server '<name>' in the cluster`.
- **More than one peer matches** it — the request is refused rather than resolved to an arbitrary one. `arcadedb-1` matches `arcadedb-10` as well, so on a cluster holding both, name the one you mean exactly.
- **The name is this server** — this server shuts down, exactly as the plain `shutdown` command does.

In a Kubernetes StatefulSet each pod's hostname is the name to use (`arcadedb-0`, `arcadedb-1`, ...). Prefer `POST /api/v1/cluster/leave` when the intent is to take a node out of the cluster rather than to stop its process.

Get server events

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "get server events"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": [{"time": "2023-06-18 15:37:40.378", "type": "INFO", "component": "Server",
"message": "ArcadeDB Server started in \u0027development\u0027 mode (CPUs\u003d8
MAXRAM\u003d4,00GB)"}]}
```

Set server setting

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "set server setting arcadedb.server.name player0"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Set database setting

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "set database setting mydb arcadedb.typeDefaultBuckets 4"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Connect cluster

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "connect cluster 192.168.0.1"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

This adds the given server to **this** server's cluster; it does not make this server join another cluster. An address that resolves to this server itself is refused with `400` instead of being silently accepted as a no-op.

Disconnect cluster

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "disconnect cluster"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

Align database

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "align database mydb"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{"result": "ok"}
```

List and terminate running queries

Since 26.10.2 a client can see what the server is running, stop one statement, and verify afterwards that the server really stopped working on it. A benchmark harness or a coordinator that gives up on a query (its own deadline, a killed client process) uses this to free the server before it runs the next one.

The list covers every protocol, not only HTTP: statements received over PostgreSQL, Bolt, gRPC, MongoDB, Redis, the Gremlin Server and MCP (see reference/mcp/mcp.pdf) are listed and stopped the same way, and the same registry answers the Cypher `SHOW TRANSACTIONS` and `TERMINATE TRANSACTIONS` commands and the Studio Running Queries tab.

Label your work. Send the `X-ArcadeDB-Query-Tag` header with a query, a command, a GET query or a batch load. The value is an opaque label of at most 256 printable characters; the statement is listed under it. Sent to `/api/v1/begin`, it labels every statement of that transaction session that carries no label of its own. A value that breaks these rules is refused with `400 Bad Request` on every endpoint, before anything runs or a transaction session is created. *(Since v26.10.2: a refused tag on `/api/v1/begin` still opened a transaction session that no client could end.)*

```
$ curl -X POST http://localhost:2480/api/v1/command/mydb \
       -H "X-ArcadeDB-Query-Tag: bench-run-42" \
       -d '{"language": "opencypher", "command": "UNWIND range(1, 10000) AS i UNWIND
range(1, 10000) AS j RETURN sum(sin(toFloat(i * j))) AS total"}' \
       --user root:arcadedb-password
```

List. `list queries` returns the statements this server is running. `database <dbname>` and `tag <label>` narrow the list (the label runs to the end of the command):

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "list queries tag bench-run-42"}' \
       --user root:arcadedb-password
```

```
{"result": [{"id": "q118-5e1f0a9c", "server": "ArcadeDB_0", "database": "mydb",
"user": "root", "protocol": "http",
  "language": "opencypher", "text": "UNWIND range(1, 10000) AS i UNWIND ...", "tag":
"bench-run-42",
```

```
  "startedAt": 1791620000000, "elapsedMs": 2311, "terminating": false}]}
```

An id is `q<n>-<instance>`: the instance part belongs to the server that issued it, so a `terminate query` that a load balancer sends to another server answers `not found` there rather than stopping a different statement. The text is truncated to 1024 characters, and the parameters are never listed. `protocol` names how the statement reached the server: `http`, `postgres`, `bolt`, `grpc`, `mongodb`, `redis`, `gremlin` (the Gremlin Server), `mcp` (a tool call of the MCP server) or `ws` (a chunk of a WebSocket insert session). Some entries carry more fields:

Field	Present when
`sessionId`	The statement runs in an HTTP transaction session.
`connectionId`	The protocol keeps connections of its own: the backend process id of a PostgreSQL connection (what `pg_backend_pid()` answers, see Cancelling a statement), `bolt-<n>` for a Bolt connection, the channel id of a MongoDB or Gremlin Server connection, the connection name of a Redis connection, the session id of a WebSocket insert session.
`clientAddress`	The address of the client the statement came from, on every protocol that has one: `/host:port`. An MCP tool call made over the stdio transport lists `stdio`. Absent when the transport cannot report the client, for example a Gremlin Server session or a custom Gremlin channelizer.
`forwardedFrom`	On an HA leader, for a statement a replica forwarded to it: the id the statement has on that replica.
`forwardedTo`	On an HA replica, for a statement it forwarded to the leader: the name of the leader running it.

Terminate. `terminate query <id>` stops a statement, and `terminate queries tag <label>` stops every statement carrying the label that you may see - for **root**, every user's statement carrying it. The command waits for the work to end before it answers, up to `wait` milliseconds in the request (5000 when absent, 10000 at most; poll `list queries` for longer), and says what happened:

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "terminate query q118-5e1f0a9c", "wait": 2000}' \
       --user root:arcadedb-password
```

```
{"result": {"id": "q118-5e1f0a9c", "status": "terminated"}}
```

Status	Meaning
`terminated`	The statement stopped, and what it wrote was rolled back.
`completed`	The statement ended on its own before it noticed the request: what it did stands, a write it committed included.
`terminating`	The statement was asked to stop and has not reached a check yet. Poll `list queries` until it is gone.

Status	Meaning
`not found`	No such statement is running: it already ended, or it is not one you may see.

A statement that ran in a transaction session (`/api/v1/begin`) answers two more fields, in the answer of `terminate query` and in each entry of the answer of `terminate queries tag` *(since v26.10.2)*: `sessionId`, the session it ran in, and `sessionOpen`, whether that session is still open now that the statement has ended. A terminated command ends its session (`false`). A terminated read leaves the session to its client (`true`), as described under *Transaction sessions* below: a caller that wants the session gone, not only the statement, ends it with `terminate transaction <sessionId>`.

```
{"result": {"id": "q121-5e1f0a9c", "status": "terminated", "sessionId": "AS-ee056170-dc9b-4956-8d71-d7cfa01900d4", "sessionOpen": true}}
```

The terminated request fails with HTTP `409` and `"error": "Query terminated"`. It is not a transient error: do not retry it. The Java `RemoteDatabase` raises it as `com.arcadedb.exception.QueryTerminatedException`, which its `transaction()` does not retry. Once a statement is gone from `list queries`, its rollback and its cleanup are over: its absence is the evidence that the server no longer works on it.

Each protocol reports the termination to its client in its own terms, never as a transient error:

Protocol	The terminated statement fails with
HTTP	Status `409`, `"error": "Query terminated"`.
PostgreSQL	SQLSTATE `57014` (`query_canceled`). The connection stays open.
Bolt	`Neo.ClientError.Transaction.Terminated`, which drivers do not retry.
gRPC	Status `CANCELLED` with the trailer `arcadedb-exception-class` set to `com.arcadedb.exception.QueryTerminatedException`. The gRPC Java client raises that exception.
MongoDB	Error code `11601` (`Interrupted`), what MongoDB answers for an operation `killOp` interrupted.
Redis	An `ERR` error whose message says the statement was terminated.
`/ws` insert session	*(Since v26.10.2.)* An `error` frame (`"error": "Insert session error"`) saying what became of the chunk. The session is closed and the connection stays open, see Streaming Inserts over WebSocket.

Termination is cooperative: the statement stops at its next check, made by the SQL and openCypher operators that loop over data (scans, expansions, UNWIND, aggregations, sorts, FOREACH, script lines), and its commit is refused if it reaches one. A Gremlin traversal checks inside its adjacency steps (`out()`, `bothE()`, ...) and while its results are iterated. A JavaScript script (`command("js", …)`), a JavaScript function called from SQL (`DEFINE FUNCTION … LANGUAGE js`) and a JavaScript trigger are interrupted through GraalVM, so even an endless loop in user code stops; the SQL such a script runs is stopped as any other.

Clients that go away. A client that disconnects while its statement runs (a closed socket, a killed process, `curl --max-time`) terminates the statement, instead of leaving the server busy with it until it completes or reaches `arcadedb.command.timeout`. This applies to plain HTTP/1.x connections once the request body has been read, not to TLS connections, to HTTP/2, or to a request whose client has already pipelined the next one. A statement that runs in a transaction session and is terminated this way follows the transaction session rule below. Keep-alive and pipelined requests are not affected. While the statement runs, the connection's read timeout (`arcadedb.network.socketTimeout`, 30 seconds by default) is suspended, so a client waiting for a statement that runs longer than that still receives its answer; the timeout is restored when the statement ends, and it still closes idle keep-alive connections. HTTP connections use TCP keepalive, so a client that vanishes without closing its connection (a host crash, a network partition) is eventually detected by the operating system and its statement terminated; the operating system's keepalive timing applies, so set `arcadedb.command.timeout` to bound such statements sooner. A client that half-closes its side of the connection after sending the request and still waits for the response cannot be told from one that left: for such clients, set `arcadedb.server.httpTerminateOnClientDisconnect` to `false`.

Transaction sessions. Terminating a command that writes in a transaction session (`/api/v1/command`, `/api/v1/batch`) also ends the session and rolls back everything the session wrote, exactly as a failure of that command does. Terminating a read (`GET` or `POST /api/v1/query`) or a time-series, vector, PromQL or Grafana request stops only that request and leaves the session and its transaction as they were, so a client whose read timed out can retry it and still commit its writes *(since v26.10.2)*. The answer of `terminate query` says which case applied through `sessionOpen`. `list transactions` lists the open sessions (id, user, database, tag, idle time, whether a request is running in it, and the ids of its running statements), and `terminate transaction <sessionId>` terminates what runs in it, rolls it back and ends it. That includes a `/ws` insert session writing into it, whose current chunk is listed under the session. The command never waits longer than its time limit (at most 10 seconds): when the session is still busy at that point it answers `terminating`, and the transaction is rolled back as soon as the running work ends. A later `/api/v1/commit` on that session publishes nothing and answers `204` with the `arcadedb-session-expired` header, as for any session that no longer exists.

Who can do what. Every authenticated user can run these commands. **root** sees and stops every statement and session; any other user only their own, and someone else's is answered `not found`, so its existence is not given away. The commands are never held back by the query admission gate (`arcadedb.queryMaxConcurrent`), so a saturated server can still be relieved, and they are never forwarded to the leader: each server lists and stops its own work.

On an HA cluster a write that a replica forwards to the leader runs on the leader. It is listed on both: on the replica with `forwardedTo` (the leader's name), on the leader with `forwardedFrom` (the id it has on the replica) and the client's tag. `terminate query <replica id>` sent to the replica stops the work on the leader too, and the same command sent to the leader finds the statement by the replica's id. See Running statements on a cluster.

Credentials. The listed text masks the value after `IDENTIFIED BY`, `PASSWORD` (`PASSWD`, `PWD`), `TOKEN`, `SECRET`, `API_KEY` and `AUTHORIZATION` (also after a `Bearer` or `Basic` scheme) with `*` (best effort).

Cluster authentication session lookup (POST, internal)

`POST /api/v1/cluster/auth-session` is a cluster-internal route that lets servers share authentication tokens (see *Tokens in a cluster* under Token-Based Authentication). It is not meant for clients and is listed here only so it is not mistaken for a public API. The request must arrive through the cluster-token channel (`X-ArcadeDB-Cluster-Token` header) on behalf of the root user: a request without that header is refused with `403`, even with root credentials, and an invalid cluster token gets `401`.

The body is `{"token": "<AU-··· token>", "action": "validate" | "revoke"}` (`action` defaults to `validate`). `validate` answers `200` with `{"user": ···, "createdAt": ···}` when this server issued the session, and `404` otherwise (including for a copy held on behalf of another server); `revoke` drops the session and answers `204`. A missing token or an unknown action answers `400`.

List Databases (GET)

Returns a list of available databases for the requesting user.

URL Syntax: `/api/v1/databases`

Responses:

- `200` OK
- `401` invalid credentials

Example:

```
$ curl -X GET http://localhost:2480/api/v1/databases \
       --user root:arcadedb-password
```

Return:

```
{"result": ["school", "mydatabase"]}
```

Since v26.11.1 the list includes the databases that are on disk but not open: those closed for idleness and those not opened at startup because `arcadedb.server.databaseLoadAtStartup` is `false`. The first request that names one opens it. A database closed with `close database`, being restored or being dropped is not listed. The `list databases` server command answers the same way. See Databases on disk that are not open.

Does database exist (GET)

Returns boolean answering if database exists.

URL Syntax: `/api/v1/exists/{database}`

Responses:

- `200` OK
- `400` no database passed: the name is empty or blank (`"error": "Database parameter is null"`, as `GET /api/v1/progress/{database}` answers). *(Since v26.12.1: before, `GET /api/v1/exists/` answered `200 {"result": false}`.)*

Example:

```
$ curl -X GET http://localhost:2480/api/v1/exists/school \
       --user root:arcadedb-password
```

Return:

```
{"result": true}
```

Since v26.11.1 a database that is on disk but not open (closed for idleness, or not opened at startup because `arcadedb.server.databaseLoadAtStartup` is `false`) answers `true`, since the first request that names it opens it. Before, it answered `false` until it was opened. Since v26.12.1 a database closed with `close database`, being restored or being dropped answers `true` as well: it exists, and `create database` refuses its name. It is still left out of the list of databases, and a request on it is answered `503` with the reason until the operator gives it back. `drop database` drops a database closed with `close database`, and refuses one a restore or a drop holds with the reason (`409`). *(Issue #11310. Before, such a database answered `false` here, and `drop database` answered that it did not exist, while `create database` of the same name was refused with `409`.)*

Get operation progress (GET)

Returns the long-running maintenance operations currently in progress on this server for one database, with their step-by-step progress. `CHECK DATABASE`, `REBUILD INDEX`, `COMPACT INDEX`, `BACKUP DATABASE` and `IMPORT DATABASE` publish their progress here while they run. The endpoint answers from an in-memory snapshot without touching the database, so it is safe to poll at any frequency (the console and Studio poll it to render a live progress bar). In a cluster each server reports its own operations.

URL Syntax: `/api/v1/progress/{database}`

Responses:

- `200` OK
- `400` no database passed
- `403` the user is not authorized on the database

Example:

```
$ curl -X GET http://localhost:2480/api/v1/progress/school \
       --user root:arcadedb-password
```

Return (one entry per running operation, empty array when nothing is running):

```
{"result": [{
  "id": 3,
  "database": "school",
  "operation": "check database fix",
  "stepName": "Checking vertices 'Account'",
  "stepIndex": 3,
  "totalSteps": 9,
  "done": 421000,
  "total": 1000000,
  "percentage": 42,
  "startedOn": 1786370000000,
  "elapsedMs": 154000
}]}
```

`percentage` is `-1` when the current step's total is unknown (for example the backup or the import, whose sizes cannot be computed upfront).

Login - Get Authentication Token (POST)

Authenticates a user and returns an authentication token that can be used for subsequent requests instead of sending credentials each time. See Token-Based Authentication for more details.

URL Syntax: `/api/v1/login`

This endpoint requires HTTP Basic Authentication with valid credentials. On successful authentication, it returns a JSON response containing the authentication token.

Responses:

- `200` OK - returns token and username
- `401` invalid credentials
- `503` the server holds the maximum number of concurrent authentication sessions and none could be reclaimed - see `server.httpAuthSessionMax`

Example:

```
$ curl -X POST http://localhost:2480/api/v1/login \
       --user root:arcadedb-password
```

Return:

```
{"token": "AU-ArcadeDB_0-2af64e60-8455-423a-bc64-ed0e19729f04", "user": "root"}
```

The returned token can be used in subsequent requests via the `Authorization: Bearer` header:

```
$ curl http://localhost:2480/api/v1/databases \
       -H "Authorization: Bearer AU-ArcadeDB_0-2af64e60-8455-423a-bc64-ed0e19729f04"
```

Logout - Invalidate Authentication Token (POST)

Invalidates an authentication token so it can no longer be used for requests. The token must be provided via the `Authorization: Bearer` header.

URL Syntax: `/api/v1/logout`

Responses:

- `204` OK - token invalidated (or was already invalid)

Example:

```
$ curl -X POST http://localhost:2480/api/v1/logout \
       -H "Authorization: Bearer AU-ArcadeDB_0-2af64e60-8455-423a-bc64-ed0e19729f04"
```

After logout, any attempt to use the invalidated token will result in a `401 Unauthorized` response.

List Active Authentication Sessions (GET)

Returns a list of all active authentication sessions. This endpoint is restricted to **root** users only for security reasons.

URL Syntax: `/api/v1/sessions`

Responses:

- `200` OK - returns list of active sessions
- `403` forbidden (non-root user)

Example:

```
$ curl http://localhost:2480/api/v1/sessions \
       --user root:arcadedb-password
```

Return:

```
{
  "result": [
```

```
    {
      "token": "AU-ArcadeDB_0-2af64e60-8455-423a-bc64-ed0e19729f04",
      "user": "root",
      "elapsedMs": 12345
    },
    {
      "token": "AU-6fa459ea-ee8a-3ca4-894e-db77e160355e",
      "user": "admin",
      "elapsedMs": 5678
    }
  ],
  "count": 2
}
```

The `elapsedMs` field indicates how many milliseconds have passed since the last request using that token.

Create an API token (POST)

Mints a long-lived API token (`at-` prefix) for a script or service. Root only. The plaintext token is returned once, in `result.token`, and is never shown again: the server keeps only its hash. On a cluster the request is forwarded to the leader and the token is replicated to every node. `GET /api/v1/server/api-tokens` lists the tokens and `DELETE /api/v1/server/api-tokens?token=<hash>` revokes one. Studio's API Tokens panel uses the same routes.

URL Syntax: `/api/v1/server/api-tokens`

Body: `name` (required), `database` (default `*`, every database), `expiresAt` (epoch milliseconds, `0` for no expiry) and `permissions` (the same structure as a group).

Responses:

- `201` created - the token is in `result.token`
- `400` missing name or malformed permissions
- `403` forbidden (non-root user)
- `409` a token with that name already exists. *(Since v26.10.2)* the body also names the exception in `exception`, like the other error bodies
- `412` the request did not arrive over a protected transport and `arcadedb.server.apiTokenRequireSecureTransport` is on

```
$ curl -X POST https://localhost:2490/api/v1/server/api-tokens \
       --user root:arcadedb-password \
       -d '{"name":"etl-job","database":"mydb"}'
```

Transport check. The token authenticates whoever holds it, so the server checks how the mint reached it. The transport counts as protected when the request came over HTTPS, from a loopback address, or from a reverse proxy listed in `arcadedb.server.apiTokenTrustedProxies` that reports

`https` for every hop through `X-Forwarded-Proto` or the RFC 7239 `Forwarded` header. What happens otherwise depends on `arcadedb.server.apiTokenRequireSecureTransport`:

- `false` (the default): the token is minted and the server logs `Minting an API token over an unprotected transport (scheme=…, peer=…)` at WARNING.
- `true`: the mint is refused with `412` and a body explaining the three ways out: connect over TLS, list the proxy, or turn the check off.

On a cluster the check also covers the hop between nodes: a mint forwarded to the leader over cleartext inter-node HTTP is refused with `412` too, even when the client itself used HTTPS.

The default of `arcadedb.server.apiTokenRequireSecureTransport` is scheduled to change to `true` in 27.1.1. From that release an unprotected mint is refused unless you set the value back to `false`. It is `false` today only because Studio's own **Create Token** button uses this route, and enforcing it would break Studio served over plain HTTP from a remote host. If you mint tokens over plain HTTP, plan for HTTPS (or a trusted proxy) before upgrading.

Behind a TLS-terminating reverse proxy. When nginx, Envoy or a Kubernetes ingress terminates TLS and forwards to ArcadeDB over HTTP, the server sees a cleartext request from the proxy's address. List that address (or its subnet) in `arcadedb.server.apiTokenTrustedProxies`, for example `-Darcadedb.server.apiTokenTrustedProxies=10.0.0.5,10.1.0.0/16`, and make sure the proxy sets `X-Forwarded-Proto` (nginx: `proxy_set_header X-Forwarded-Proto $scheme;`). The rules are strict on purpose:

- A peer that is not on the list gains nothing by sending the same headers. They are ignored, because the client asking for a token is the one that would forge them.
- Entries must be literal addresses or CIDR ranges. A hostname is rejected rather than resolved.
- A list that does not parse is treated as empty, so a typo denies instead of opening the gate. The server logs the discarded list at SEVERE (`Ignoring the whole of arcadedb.server.apiTokenTrustedProxies (…)`).
- List only an L7 proxy that writes the header itself. An L4 balancer (nginx `stream`, HAProxy `mode tcp`) passes the client's own headers through, so listing one lets any client vouch for itself.
- If the request carries a `Forwarded` header, every element must include `proto=https`; an element without `proto=` refuses the mint. A proxy that does not use `Forwarded` should strip any it receives.

The gRPC `CreateApiToken` RPC applies the same list, reading the `x-forwarded-proto` and `forwarded` metadata keys. Unlike HTTP it always refuses an unprotected mint, whatever `apiTokenRequireSecureTransport` says.

Execute a query (GET|POST)

This command allows executing idempotent commands, like `SELECT` and `MATCH`:

URL Syntax GET: `/api/v1/query/{database}/{language}/{command}`

URL Syntax POST: `/api/v1/query/{database}`

Where:

- `database` is the database name
- `language` is the query language used. is the query language used, between "sql", "sqlscript", "graphql", "cypher", "gremlin", "mongo" and any other language supported by ArcadeDB and available at runtime.
- `command` the command to execute in encoded format
- `params` (optional), is the map of parameters to pass to the query engine via the POST body, where parameters are introduced with a colon `:`, or a JSON array of positional parameters. Parameter values may include the typed-marker objects described in Typed parameter markers to send `byte[]` — e.g. INT8-encoded vectors — without a float32 round trip.

When using the `GET` variant the query needs to be URL encoded.

Due to security reasons (encoded) slashes `/` (`%2F`) which are used for divisions or block comments, cannot be used in queries via the GET method with the `query/` endpoint.

Question marks (`?`) cause the server to stop reading the query string when sent via GET. To use question marks (inside strings) one can use `format('%c',63)`; in this case make sure to replace all percent symbols (`%`) in the format string with `%%`.

These restrictions do not apply to the `POST` variant, where the `language` and `command` are send in the body.

Even though a `POST` method is used, the query in `command` has to be idempotent.

(Since v26.10.2.) A SQL statement is classified by everything it runs, not only by its outermost statement. A write nested in a `LET`, a projection, a `FROM` subquery, a `CASE`, a `RETURN` or a script block (`IF`, `FOREACH`, `WHILE`) makes the whole statement a write, so `/api/v1/query` refuses it with `400` (the query is not idempotent) and writes nothing, for example `SELECT FROM Account LET $x = (INSERT INTO Log SET a = 1)` or `SELECT FROM (INSERT INTO Log SET a = 1)`. Send such a statement to `/api/v1/command`. A nested read, such as `SELECT $c FROM Account LET $c = (SELECT count(*) FROM Account)`, still runs.

`POST /api/v1/query` runs without an implicit transaction, like `GET /api/v1/query`, so a full scan of a large type can use several threads and is as fast as with `GET`. Inside a session (`/begin`) a scan stays sequential to keep it consistent with the transaction. `POST /api/v1/command` still runs in a transaction.

`EXPLAIN` is answered the same way by the `GET` and the `POST` variants and by `/api/v1/command`: the plan arrives in the `explain` (indented text) and `explainPlan` (structured) members of the response envelope, and `result` is empty — the plan is the answer, not a row. Requesting it with `Accept: application/x-ndjson` is refused with `400` ("EXPLAIN produces a plan, not a row stream"), because a stream of rows plus a stats trailer has nowhere to carry one.

To receive the rows one at a time as they are produced, instead of one JSON document at the end,

send `Accept: application/x-ndjson`. See Stream query results (NDJSON).

Responses:

- `200` OK
- `400` invalid language, invalid query
- `401` invalid credentials
- `404` database does not exist (`Database not found`)
- `503` the database is there and cannot serve the request now: closed, being restored, dropped or recovered (see a database that is not available)
- `500` cannot execute query
- `503` the server is busy and the query can be retried as it is: the running queries hold the heap budget (`QueryHeapBudgetExceededException`, see `queryMaxHeapRAM`), or the query admission gate did not start it, because it waited longer than `queryQueueTimeout` or found the queue full (`QueryAdmissionException`, see `queryMaxConcurrent`). The answer carries `Retry-After: 1` *(since v26.12.1)*
- `503` the database was closed while the request was running because it is being reinstalled from a snapshot on an HA replica (`DatabaseIsReopeningException`, with `Retry-After: 0` since v26.12.1), or the server is shutting down (no `Retry-After`): retry as it is
- `503` the database was closed (`CLOSE DATABASE`, a restore) while the request was running and is still there (`DatabaseNotAvailableException`, `exceptionArgs CLOSED`) *(since v26.11.1, before it was `404`)*, or `404` it was dropped (`DatabaseIsClosedException`)

Example:

```
$ curl -X GET http://localhost:2480/api/v1/query/school/sql/select%20from%20Class \
       --user root:arcadedb-password
```

The `query` endpoint may also be used via the POST method, which has no character restrictions such as `/` or `?`:

```
$ curl -X POST http://localhost:2480/api/v1/query/school \
       -d '{"language": "sql", "command": "select from Class"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

The POST variant accepts additional parameters in the JSON body:

- `autoCommit` (optional) controls transaction behavior (same as in Execute database command):
  - `true`: Forces the query to execute within an atomic transaction
  - `false`: Disables transaction wrapping (query executes directly)
  - not specified: Uses automatic behavior (no transaction for read-only queries)

Example of forcing a transaction for a query to ensure consistent reads:

```
$ curl -X POST http://localhost:2480/api/v1/query/mydb \
       -d '{"language": "sql", "command": "SELECT FROM Account WHERE balance > 1000",
"autoCommit": true}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Execute database command (POST)

Executes a non-idempotent command (as an implicit transaction).

URL Syntax: `/api/v1/command/{database}`

Where:

- `database` is the database name

Example to create the new document type "Class":

```
$ curl -X POST http://localhost:2480/api/v1/command/school \
       -d '{"language": "sql", "command": "create document type Class"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

The payload, as a JSON, accepts the following parameters:

- `language` is the query language used, between "sql", "sqlscript", "graphql", "cypher", "gremlin", "mongo" and any other language supported by ArcadeDB and available at runtime.
- `command` the command to execute in encoded format
- `awaitResponse` (optional) a boolean which is by default "true", if set to "false" the command will be executed asynchronously and only acknowledgement of receiving the command is responded. The completion of the command is noted in the log, yet no results of the command can be returned.
- `limit` (optional) is the maximum number of results to return. *(Since v26.12.1)* this cap, or `arcadedb.server.httpQueryDefaultLimit` when neither the request nor the statement sets one, is no longer added to the text of a SQL `SELECT` or `MATCH` as a `LIMIT`: the statement is planned and cached exactly as an embedded or Postgres client would send it, so a `SELECT DISTINCT` keeps its parallel plan and a `GROUP BY` its columnar plan over HTTP too. The engine still stops early where the cap allows it, for example by keeping only the first rows of an `ORDER BY` sort. A `LIMIT` written in the statement decides the rows returned, as before.
- `params` (optional), is the map of parameters to pass to the query engine, where parameters are prefixed with a colon `:`, or a JSON array of positional parameters. Parameter values may include the typed-marker objects described in Typed parameter markers to send `byte[]` — e.g. INT8-encoded vectors — without a float32 round trip.
- `retries` (optional), is the number of times the command (transaction) is retried. When omitted it

follows the `arcadedb.txRetries` database setting (3 by default), the same attempt count the embedded and the remote Java APIs take it from. Only a concurrent-modification conflict is retried (`NeedRetryException` and `DuplicatedKeyException`, the latter capped at a single retry), and always after a full rollback, so a retried attempt starts from committed state with nothing of the failed one surviving. Note the consequence for side effects **outside** the database: a SQL or JavaScript function that calls another system can now run more than once per request. Send `"retries": 1` to keep the previous single-attempt behavior for such a command.

- `autoCommit` (optional) controls transaction behavior:
  - `true`: Forces the command to execute within an atomic transaction that auto-commits on success
  - `false`: Disables automatic transaction wrapping (command executes directly without transaction)
  - not specified (default): Uses automatic behavior (commands are wrapped in atomic transactions unless a session is active)
  - NOTE: This parameter is ignored when using session-based transactions (see Begin a transaction). If you provide `autoCommit: true` with an active session, a warning will be logged and the session transaction will be used instead.
- `serializer` (optional) specify the serializer used for the result:
  - `graph`: returns a graph separating vertices from edges
  - `record`: returns everything as records
  - `studio`: by default it's like record but with additional metadata for vertex records, such as the number of outgoing edges in `@out` property and total incoming edges in `@in` property. This serializer is used by Studio.

A read-only command can stream its rows as newline-delimited JSON with `Accept: application/x-ndjson`, exactly like a query. See Stream query results (NDJSON).

Asynchronous commands (using `"awaitResponse": false`) are queued, and their processing is configurable via `async*` database settings.

Responses:

- `200` OK
- `202` Accepted
- `400` invalid language, invalid command
- `400` *(since v26.12.1)* a SQL or openCypher function called with an argument it is not defined for: a value of the wrong kind (`toIntegerList(1)`, `geo.lineString(1)`, `point.distance(1, 2)`; `exception` is `com.arcadedb.exception.TypeMismatchException`), or a value of the right kind it does not accept (a timestamp text that does not parse in `ts.rate()`, an unknown unit in `duration()`, a percentile outside `[0, 1]`, a `null` where a value is required; `exception` is `com.arcadedb.exception.InvalidArgumentException`). Before v26.12.1 these answered `500`, some with a raw `NullPointerException`, `ClassCastException` or `DateTimeParseException`. Over Bolt they are `Neo.ClientError.Statement.TypeError` and `Neo.ClientError.Statement.ArgumentError`, over the

Postgres protocol `42804` and `22023`

- `400` *(since v26.12.1)* an openCypher `LOAD CSV` of a URL that cannot be opened or read, such as a file that does not exist (`exception` is `com.arcadedb.exception.ExternalResourceException`). Before v26.12.1 this answered `500`. Over Bolt it is `Neo.ClientError.Statement.ExternalResourceFailed`
- `401` invalid credentials
- `409` *(since v26.11.1)* the command creates, or renames onto, a schema object name that is already taken: a type, property, index, constraint, bucket, trigger, view or function. `error` names the kind (for example `"Type already exists"`, kept in production mode), `exception` is `com.arcadedb.exception.AlreadyExistsException` and `exceptionArgs` the kind (`TYPE`, `PROPERTY`, `INDEX`, ...). Not retryable. Before v26.11.1 these answered `500` (`"Cannot execute command"` or `"Internal error"`), and a rename onto a taken type name `400`
- `409` *(since v26.12.1)* the command is refused for what the database holds now: `TRUNCATE TYPE`, `TRUNCATE BUCKET` or `DROP TYPE` of a vertex or edge type that still has records (without `UNSAFE`), `DROP PROPERTY` of a property an index uses (without `FORCE`), an openCypher `DELETE` of a node that still has relationships (use `DETACH DELETE`). `error` is `"Object in use"` and `exception` `com.arcadedb.exception.ObjectInUseException`; the detail says how to resolve it. Not retryable. Before v26.12.1 these answered `500`. In the same release a `LIMIT`/`SKIP` that is not a number, a `CREATE EDGE`/`CREATE VERTEX` on a type of the other kind, a `CREATE EDGE` endpoint that is not a vertex, a SQL `MATCH rid:` that is not a RID, a bare `ORDER BY @rid` on the rows of a SQL `MATCH` that returns columns, `REBUILD INDEX` of an index that does not exist, `ALTER PROPERTY` of a setting it cannot change (the type of a property included) and `IMPORT DATABASE` of a source that does not exist answer `400` instead of `500`
- `503` the command's transaction conflicted with a concurrent commit and the server's own retries did not get past it (`ConcurrentModificationException`), or another retryable refusal (`NeedRetryException`, a write refused by a leader that is stepping down included *(since v26.12.1)*, see Writes during a leadership transfer): retry as it is. The answer carries `Retry-After: 0` *(since v26.12.1)*: the conflict clears as soon as the competing commit has landed
- `503` the server is busy and the command can be retried as it is: the running queries hold the heap budget (`QueryHeapBudgetExceededException`, see `queryMaxHeapRAM`), or the query admission gate did not start it, because it waited longer than `queryQueueTimeout` or found the queue full (`QueryAdmissionException`, see `queryMaxConcurrent`). The answer carries `Retry-After: 1` *(since v26.12.1)*
- `503` the database was closed while the request was running because it is being reinstalled from a snapshot on an HA replica (`DatabaseIsReopeningException`, with `Retry-After: 0` since v26.12.1), or the server is shutting down (no `Retry-After`): retry as it is
- `503` the database was closed (`CLOSE DATABASE`, a restore) while the request was running and is still there (`DatabaseNotAvailableException`, `exceptionArgs CLOSED`) *(since v26.11.1, before it was `404`)*, or `404` it was dropped (`DatabaseIsClosedException`)
- `404` *(since v26.12.1)* an openCypher statement reads, sets or links a node it deleted earlier in the same statement (`MATCH (n:A) DELETE n RETURN n.x`). `error` is `"Entity deleted"` and `exception` is `com.arcadedb.exception.DeletedEntityException`. Not retryable. Before v26.12.1 this answered `500`. Over Bolt it is `Neo.ClientError.Statement.EntityNotFound`
- `404` the database does not exist (`Database not found`), or `503` it is there and cannot serve the command now: closed, being restored (a write while a restore holds an open database is refused

this way), dropped or recovered (see a database that is not available)

- `504` *(since v26.12.1)* in an HA cluster, the write was handed to the replication, or forwarded by a follower to a leader that died or did not answer, and its outcome is unknown: it may or may not have been applied. `error` is `"Transaction outcome unknown: the write may have been applied, check before retrying it"` (kept in production mode) and `exception` is `com.arcadedb.exception.TransactionOutcomeUnknownException` (`com.arcadedb.server.ha.raft.ReplicationDispatchedTimeoutException` on the node that handed the write to the replication). No `Retry-After`: check whether the write landed before sending it again, see Outcome-unknown response contract. Before v26.12.1 these answered `500`

Example of insertion of a new Client by using parameters:

```
$ curl -X POST http://localhost:2480/api/v1/command/company \
       -d '{"language": "sql", "command": "create vertex Client set firstName =
:firstName, lastName =  :lastName", params: {"firstName": "Jay", "lastName":
"Miner"}}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Example of forcing an atomic transaction with `autoCommit`:

```
$ curl -X POST http://localhost:2480/api/v1/command/mydb \
       -d '{"language": "sql", "command": "INSERT INTO Account SET balance = 1000",
"autoCommit": true}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Example of disabling automatic transaction wrapping:

```
$ curl -X POST http://localhost:2480/api/v1/command/mydb \
       -d '{"language": "sql", "command": "SELECT FROM Account LIMIT 1", "autoCommit":
false}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Stream query results (NDJSON)

`GET /api/v1/query`, `POST /api/v1/query` and `POST /api/v1/command` can stream the result as newline-delimited JSON (NDJSON, also called JSON Lines) instead of building one JSON document. Send the request header `Accept: application/x-ndjson`; the response comes back with `Content-Type: application/x-ndjson` and `Transfer-Encoding: chunked`. The server holds one row in memory at a time, and the client can process each row as soon as it arrives, so a result larger than the server heap, or the client's, can still be read.

The answer is one HTTP response, not a series of pages: there is nothing to request after it, and no cursor. Every line that is not empty is a JSON object with exactly one key, which says what the line

is:

Line	Meaning
`{"record": {…}}`	One result row, in the same shape as an item of the `result` array of the buffered (`application/json`) response.
`{"stats": {"limit": n, "returned": n, "truncated": b}}`	The trailer: the last line of a stream that completed. Nothing follows it.
`{"error": {…}}`	The statement failed after the `200` status had already been sent. The last line of a stream that failed. No `stats` line follows it.
an empty line	Keep-alive, written while the server is still producing the next row. Skip it.

How a client knows there is more to read. Until it has read a `stats` or an `error` line, the stream is not over, however long the pause between two rows. Read it line by line:

1. skip empty lines;
2. on `record`, hand the row to the application and read the next line;
3. on `stats`, the result is complete. Check `truncated` (see below);
4. on `error`, the result is incomplete, and the line says why;
5. if the connection closes before a `stats` or an `error` line, the result is incomplete (a dropped connection, a proxy timeout). Treat it as a failure, never as a short result.

Example:

```
$ curl -X POST http://localhost:2480/api/v1/query/school \
       -d '{"language": "sql", "command": "select name from Student order by name",
"limit": 2}' \
       -H "Content-Type: application/json" \
       -H "Accept: application/x-ndjson" \
       --user root:arcadedb-password
```

```
{"record":{"name":"Ann"}}
{"record":{"name":"Bob"}}
{"stats":{"limit":2,"returned":2,"truncated":true}}
```

The `GET` variant takes the same header, and the limit as the `limit` query parameter: `curl -H "Accept: application/x-ndjson" "http://localhost:2480/api/v1/query/school/sql/select%20name%20from%20Student?limit=2"`.

The `stats` trailer and row limits

The trailer carries the same three values the buffered response reports at its top level:

- `limit`: the row cap applied to this response, `-1` when there was none;

- `returned`: how many `record` lines were sent;
- `truncated`: `true` when the cap stopped the stream **and** at least one more row exists. The server checks for that extra row before it says so, so a result that has exactly `limit` rows ends with `"truncated": false`.

The row limits of the buffered response apply to a stream too:

- With no `limit` in the request and no `LIMIT` in the statement (a SQL `LIMIT`, or the `LIMIT` of an openCypher `RETURN`), the stream stops at `arcadedb.server.httpQueryDefaultLimit` rows (20,000 by default) and ends with `"truncated": true`. To stream the whole result, send `"limit": -1` (`?limit=-1` on the `GET` variant).
- Whatever the request asks for, one response carries at most `arcadedb.server.httpQueryMaxResultRows` rows (1,000,000 by default). Where the buffered response answers a result above that ceiling with `413`, a stream has already sent its `200`: it sends the rows up to the ceiling and then ends with an `error` line whose `status` is `413` and whose `exception` is `com.arcadedb.server.http.ResultSetTooLargeException`, instead of the `stats` trailer.

A stream that ended with `"truncated": true` cannot be resumed. To read the rows after it, send a new query for the next page. Prefer a filter on the last value you read over `SKIP`, which reads the skipped rows again on every page, for example `SELECT FROM Student WHERE name > :last ORDER BY name LIMIT 10000` on an indexed `name`.

Errors in a stream

A request the stream cannot express is refused before the statement runs, with an ordinary `400` JSON response:

- a statement that writes (`INSERT`, `UPDATE`, `DELETE`, `CREATE …`, `BACKUP DATABASE` and any other DDL), also on `/api/v1/command`: the rows would reach the client before the statement finished, and it could still roll back. Send it with `Accept: application/json`;
- `EXPLAIN` and `PROFILE`, whose answer is a plan, not rows;
- the `graph` and `studio` serializers, which build one document out of the whole result. Use the `record` serializer;
- `profileExecution`. The buffered envelope members `explain`, `explainPlan`, `profile` and the write statistics have no place in a stream.

A failure after the stream has started cannot change the `200` status line, so it is reported in-band as the last line, `{"error": {…}}`, which carries what the buffered response would have answered the same failure with:

- `message`: the failure's own message outside production mode. *(Since v26.10.2)* in `production` server mode it is the classified label, as the buffered response conceals the raw text;
- `status`: the HTTP status the buffered response would have used, and `exception` / `exceptionArgs` (the latter only when the failure has structured arguments, such as a duplicated key's index, keys and RID);
- *(since v26.10.2)* `error`, the classified label (for example `Cannot execute command`), `requestId` when the request carried one, `detail`, the cause chain, outside production mode only, and `retryAfter`,

the integer number of seconds the buffered response would have sent as `Retry-After`, present only on a retryable refusal (a `503`, or a `409` for a request with the same `X-Request-Id` still in flight).

Key on `status`, `exception` and `retryAfter` rather than parsing `message`.

Slow queries, slow clients and proxies

- While the server is still producing the next row it writes an empty line every `arcadedb.server.httpStreamingKeepAliveInterval` milliseconds (default 5000, `0` disables), so a client or proxy that bounds silence does not mistake a slow query for a dead server.
- A client that stops reading holds a server thread. When one write makes no progress for `arcadedb.server.httpStreamingWriteTimeout` milliseconds (default 60000) the server closes the connection, and the client sees a stream with no trailer.
- The response carries `Cache-Control: no-cache` and `X-Accel-Buffering: no`, so nginx passes the rows through as they are written. A proxy that buffers whole responses defeats the streaming without breaking it: the rows arrive, all at the end.

Clients

The Java remote client streams with `RemoteDatabase.queryStream()` and `commandStream()` (see Streaming results), the Go HTTP driver with `QueryStream` and `CommandStream`, and the JavaScript HTTP driver with `queryStream` and `commandStream`. The Java client skips the keep-alive lines, raises the `error` line as an exception, and raises a `RemoteException` when the stream ends with no trailer.

Write statistics (`stats`)

When a command **modifies** data, the `POST /api/v1/command/{database}` response includes a `stats` object alongside `result`, reporting how many entities the command created, deleted or changed. The object is **omitted for read-only queries** (a command that changes nothing has no `stats` field). It is populated for OpenCypher writes — including counts aggregated across `UNION` branches and `CALL { … }` subqueries — and mirrors the counters the Bolt driver exposes through its result summary.

The shape is an ArcadeDB-native camelCase object with a stable set of keys (each counter is always present, `0` when unused):

Key	Meaning
`nodesCreated` / `nodesDeleted`	Vertices created / deleted
`relationshipsCreated` / `relationshipsDeleted`	Edges created / deleted
`propertiesSet`	Properties assigned a value
`labelsAdded` / `labelsRemoved`	Type/label memberships added / removed
`indexesAdded` / `indexesRemoved`	Indexes created / dropped

Key	Meaning
`constraintsAdded` / `constraintsRemoved`	Constraints (`UNIQUE`, `NOT_NULL`, `KEY`, `TYPED`) added / removed
`containsUpdates`	`true` when any counter above is non-zero

Example — a Cypher write returns the counters it produced:

```
$ curl -X POST http://localhost:2480/api/v1/command/school \
       -d '{"language": "cypher", "command": "CREATE (a:Person {name:$n})-[:KNOWS]-
>(b:Person)", "params": {"n": "Ada"}}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

```
{
  "user": "root",
  "result": [],
  "stats": {
    "nodesCreated": 2,
    "nodesDeleted": 0,
    "relationshipsCreated": 1,
    "relationshipsDeleted": 0,
    "propertiesSet": 1,
    "labelsAdded": 2,
    "labelsRemoved": 0,
    "indexesAdded": 0,
    "indexesRemoved": 0,
    "constraintsAdded": 0,
    "constraintsRemoved": 0,
    "containsUpdates": true
  }
}
```

The gRPC service carries the same counters in the write-result summary of `ExecuteCommand`; see gRPC API.

5.2.11. Numbers and arrays in `params`

Parameters keep the numbers as written: an integer beyond the `long` range, or a decimal with more than 17 significant digits, is read as a `DECIMAL` (`BigDecimal`) instead of being wrapped or rounded to a double. A number token with more than 1000 characters or an exponent beyond 1000 is refused with `400`. A JSON array given as a parameter is a list, as when the statement runs embedded: written to a `LIST` property it is stored as that list, and `[0.1, 3.141592653589793]` keeps both doubles.

A list of numbers nested inside a map parameter (`INSERT … CONTENT :map`, openCypher `SET n = $map`, `UNWIND $rows`) is kept in the compact array form vectors use, so an undeclared property stores it as `ARRAY_OF_LONGS` or `ARRAY_OF_DOUBLES`. The SQL list operators and functions read such a value as a list (see `UPDATE`). Written to a property declared `STRING`, it is stored as the text of its elements, `"[1,`

2]", the text the same list gets. *(Since v26.12.1, issue #12896; before, a `STRING` property refused it with "Cannot convert type 'class [J' to 'STRING'".)*

Positional parameters

`params` given as a JSON array (or as an object keyed `"0"`, `"1"`, ...) binds its values by position. With SQL their number must match the placeholders of the statement: each `?`, and each `:name` bound positionally, takes exactly one value (send `null` explicitly to bind a null), and in a `sqlscript` the placeholders of all its statements are counted together. A Postgres-style `$N` reads the value at position `N` and may leave lower positions unread. A request that does not match is refused with `400` before the statement runs (`CommandSQLParsingException`, the message names both counts). A list meant for one placeholder is one value, a nested array:

```
$ curl -X POST http://localhost:2480/api/v1/query/school \
       -d '{"language": "sql", "command": "select from Class where name in ?",
"params": [["English", "Math"]]}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Named parameters (a JSON object with names as keys) are not counted. See also the Java API.

(since v26.11.1) Before, a surplus value was silently dropped (`"params": ["English", "Math"]` with `where name in ?` answered only the `English` rows) and a missing value was bound to `null`.

Typed parameter markers (`$bytes`, `$int8`, `$rid`, `$decimal`, dates)

JSON has no native type for binary data, links, exact decimals or dates, so HTTP/JSON clients cannot send these parameters with the standard scalar/array shapes: a record id would arrive as a string, a decimal as a double and a date as a number. The `query` and `command` endpoints recognise reserved single-key marker objects in any parameter value (named or positional) and decode them to the Java type before the parameter binder sees them. One use case is sending an INT8-encoded vector to a `LSM_VECTOR` index without the float32 round trip — see Vector Encoding.

Marker	Decodes to
`{"$bytes": "<base64>"}`	`byte[]` (BINARY) from a base64 string. Both standard (RFC 4648 section 4) and URL-safe (section 5, with `-` and `_` in place of `+` and `/`) alphabets are accepted.
`{"$int8": [v0, v1, ⋯]}`	`byte[]` from an array of integers. Each element must round to an integer in `[-128, 127]`; fractional or out-of-range values are rejected with HTTP 400.
`{"$date": "2024-01-15"}`	`LocalDate` (DATE) from an ISO-8601 date.
`{"$rid": "#12:0"}`	a link to the record (LINK), rather than its text. *(Since v26.12.1)*

Marker	Decodes to
`{"$decimal": "0.10"}`	`BigDecimal` (DECIMAL) from its text, scale kept, rather than a double. The text is held to the limits of a JSON number (1000 characters, an exponent up to 1000). *(Since v26.12.1)*
`{"$datetime": 1700000000123}`	`java.util.Date` (DATETIME) from epoch milliseconds, as a JSON integer. *(Since v26.12.1)*
`{"$localDateTime": "2024-02-29T13:14:15"}`	`LocalDateTime` (DATETIME) from ISO-8601 text, nanoseconds kept. *(Since v26.12.1)*
`{"$instant": "2023-11-14T22:13:20.123Z"}`	`Instant` (DATETIME) from ISO-8601 text. *(Since v26.12.1)*
`{"$zonedDateTime": "2024-01-01T10:00:00+01:00[Europe/Rome]"}`	`ZonedDateTime` (ZONED_DATETIME) from ISO-8601 text, zone kept. *(Since v26.12.1)*

The Java remote client sends its parameters, and the values of a record it saves, with these markers. A server older than a marker receives it as a one-key map.

A single-key map whose key is one of these is decoded; multi-key maps that happen to contain such a key are passed through unchanged so user data with leading-`$` keys is not silently transformed. The decoder also recurses into nested arrays and maps, so a list of typed markers (e.g. a batch of int8 vectors as a single parameter) works.

Examples:

```
# Base64-encoded byte array
curl -X POST http://localhost:2480/api/v1/query/mydb \
     -H "Content-Type: application/json" \
     --user root:arcadedb-password \
     -d '{
       "language": "sql",
       "command": "SELECT expand(`vector.neighbors`(\"Doc[embedding]\", :q, 5))",
       "params": {
         "q": {"$bytes": "AwIB/38="}
       }
     }'

# Integer array form (handy for debugging)
curl -X POST http://localhost:2480/api/v1/query/mydb \
     -H "Content-Type: application/json" \
     --user root:arcadedb-password \
     -d '{
       "language": "sql",
       "command": "SELECT expand(`vector.neighbors`(\"Doc[embedding]\", :q, 5))",
       "params": {
         "q": {"$int8": [1, 2, 3, -1, 127, -128]}
       }
     }'
```

Malformed markers (invalid base64, non-numeric `$int8` element, value out of `[-128, 127]`, a `$rid` that is not a record id, a `$decimal` that is not a number or is too large, a date that is not valid ISO-8601, `null` payload) return `400 Bad Request` with a message naming the marker and the offending value, before the statement runs.

Vector search (POST)

Returns the nearest neighbors of a query vector in a dense `LSM_VECTOR` or sparse `LSM_SPARSE_VECTOR` index, without writing the `vector.neighbors` SQL by hand. ArcadeDB does not compute embeddings: the caller supplies the vector. The same request body and limits apply to the MCP `vector_search` tool and the gRPC `VectorSearch` RPC.

URL Syntax: `/api/v1/vector/{database}/search`

The JSON body accepts:

- `indexName` (required) name of the vector index, for example `Doc[embedding]`
- `queryVector` (required) array of numbers, not empty, finite values only. On a dense index its length must match the index dimensions and at least one value must be non-zero. With `sparse: true` it holds the weights matching `queryIndices`, or the full vector when `queryIndices` is omitted (zero entries are dropped)
- `k` (optional, default `10`) maximum number of results, between `1` and `1000`
- `sparse` (optional, default `false`) search an `LSM_SPARSE_VECTOR` index instead of a dense one. The index type must match the flag
- `queryIndices` (optional, sparse only) dimension ids matching the `queryVector` weights: non-negative integers, no duplicates, same length as `queryVector`. Refused without `sparse: true`
- `efSearch` (optional, dense only) HNSW search beam width, between `1` and `10000`. Refused with `sparse: true`
- `filter` (optional) read-only SQL `WHERE` predicate, at most 4096 characters, applied to each neighbor row (record properties are flattened, and `@rid`, `@type`, `record` plus `distance` or `score` are available). With a filter the search inspects a candidate window of `k * 8` rows, at most 8000, reported as `candidateLimit`

The response carries:

- `indexName`, `sparse`
- `scoring`: the ranking direction, for example `distance_lower_is_better:COSINE` for a dense index, or `score_higher_is_better:dot_product` (`score_higher_is_better:idf_weighted_dot_product` with the IDF modifier) for a sparse one
- `candidateLimit`: the candidate window inspected
- `truncated`: `true` when `k` results were returned, so more matches may exist
- `count`, `results`: each hit has `rid`, `properties` and either `distance` (dense, lower is better) or `score` (sparse, higher is better). Records deleted between the index scan and the load are skipped, so a search can return fewer than `k` hits

Responses:

- `200` OK
- `400` missing body, missing or out-of-range argument, unknown index or wrong index type (the message lists the available indexes), invalid filter expression
- `401` missing or invalid credentials
- `403` the user has no access to the database
- `404` the database does not exist (`{"error": "Database not found"}`). A user without a grant on that name gets the `403` instead, so the answer does not reveal which databases exist

The route runs without a transaction. A `400` caused by a bad argument does not roll back a transaction the client opened with `/begin`.

Example:

```
$ curl -X POST http://localhost:2480/api/v1/vector/mydb/search \
       -d '{"indexName": "Doc[embedding]", "queryVector": [0.12, 0.85, 0.33], "k": 2,
"filter": "lang = '\''en'\''"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{
  "indexName": "Doc[embedding]",
  "sparse": false,
  "scoring": "distance_lower_is_better:COSINE",
  "candidateLimit": 16,
  "truncated": true,
  "count": 2,
  "results": [
    { "rid": "#12:4", "distance": 0.018, "properties": { "@rid": "#12:4", "@type":
"Doc", "title": "Graphs 101", "lang": "en" } },
    { "rid": "#12:9", "distance": 0.041, "properties": { "@rid": "#12:9", "@type":
"Doc", "title": "Vector search", "lang": "en" } }
  ]
}
```

Hybrid search (POST)

Fuses a vector leg, an optional full-text leg and an optional graph expansion leg into one ranked list, using the engine's `vector.fuse` function. The same request body and limits apply to the MCP `hybrid_search` tool and the gRPC `HybridSearch` RPC.

URL Syntax: `/api/v1/vector/{database}/hybrid`

The JSON body accepts:

- `vectorIndexName` (required) name of the vector index

- `queryVector`, `queryIndices`, `sparse`, `efSearch`, `filter` (vector leg): same meaning and limits as in Vector search. The `filter` applies to the vector leg only
- `k` (optional, default `10`) maximum number of fused results, between `1` and `1000`. Each retrieval leg fetches `k * 4` rows, at most 1000, before fusion
- `fulltextIndexName` and `fulltextQuery` (optional) add the full-text leg: a `FULL_TEXT` index and a Lucene-syntax query. They must be given together, otherwise the request is refused. Note the field is `fulltextQuery`, not `queryText` as in Full-text search
- `fusionStrategy` (optional, default `RRF`) one of `RRF`, `DBSF`, `LINEAR`, case-insensitive
- `weights` (optional) object with per-leg weights: `vector` (default `1.0`), `fulltext` (default `1.0`), `expand` (default `0.5`). Each must be a finite number, not negative. Any other key, or a weight for a leg the request does not run, is refused
- `expand` (optional) object that adds the graph expansion leg, seeded from the results of the retrieval legs (at most 256 seeds) and ranked by breadth-first discovery order:
  - `edgeTypes` (optional) edge type names to walk; each must exist. Empty or omitted walks every edge type
  - `direction` (optional, default `out`) one of `out`, `in`, `both`
  - `maxDepth` (optional, default `1`) between `1` and `3`

  The expansion returns at most 2000 records. Both the vector index and the full-text index must be declared on vertex types. Since the expansion leg has no score, it can only be used with `RRF`.

The response carries `vectorIndexName`, `sparse`, `scoring`, `fused`, `truncated`, `count`, `results`, plus:

- `fulltextIndexName` when the full-text leg ran
- `fusionStrategy` (upper-cased) when `fused` is `true`
- `legs`: per-leg accounting. `vector.count` always; `fulltext` with `indexName`, `similarity` (`BM25` or `CLASSIC`) and `count` when that leg ran; `expand` with `direction`, `edgeTypes`, `maxDepth`, `truncated` (fan-out cap reached), `seedCount`, `seedsTruncated` and `count` when `expand` was given

Fusion needs at least two legs that returned rows. When only the vector leg did, `fused` is `false` and each hit carries the native `distance` (dense) or `score` (sparse) instead of a fused score. Otherwise each hit has `rid`, `fusedScore` (higher is better), `sources` (the legs that produced it: `vector`, `fulltext`, `expand`) and `properties`, plus `depth` and `path` (RIDs from the seed, seed included) for a hit found by the expansion.

Responses:

- `200` OK
- `400` missing body, missing or out-of-range argument, unknown index, edge type or strategy, half a full-text leg, `expand` with a strategy other than `RRF` or on a non-vertex type, invalid filter or full-text query
- `401` missing or invalid credentials
- `403` the user has no access to the database

- `404` the database does not exist (`{"error": "Database not found"}`). A user without a grant on that name gets the `403` instead, so the answer does not reveal which databases exist

Example:

```
$ curl -X POST http://localhost:2480/api/v1/vector/mydb/hybrid \
       -d '{
             "vectorIndexName": "Doc[embedding]",
             "queryVector": [0.12, 0.85, 0.33],
             "fulltextIndexName": "Doc[content]",
             "fulltextQuery": "+graph database",
             "k": 5,
             "expand": { "edgeTypes": ["Cites"], "direction": "out", "maxDepth": 1 }
           }' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return (abridged):

```
{
  "vectorIndexName": "Doc[embedding]",
  "sparse": false,
  "scoring": "distance_lower_is_better:COSINE",
  "legs": {
    "vector": { "count": 20 },
    "fulltext": { "indexName": "Doc[content]", "similarity": "BM25", "count": 7 },
    "expand": { "direction": "out", "edgeTypes": ["Cites"], "maxDepth": 1,
"truncated": false,
                "seedCount": 24, "seedsTruncated": false, "count": 11 }
  },
  "fused": true,
  "fusionStrategy": "RRF",
  "fulltextIndexName": "Doc[content]",
  "truncated": true,
  "count": 5,
  "results": [
    { "rid": "#12:4", "fusedScore": 0.0325, "sources": ["vector", "fulltext"],
"properties": { "@rid": "#12:4", "@type": "Doc", "title": "Graphs 101" } },
    { "rid": "#12:17", "fusedScore": 0.0081, "sources": ["expand"], "depth": 1,
"path": ["#12:4", "#12:17"], "properties": { "@rid": "#12:17", "@type": "Doc",
"title": "Property graphs" } }
  ]
}
```

Full-text search (POST)

Runs a Lucene-syntax query against a `FULL_TEXT` index and returns the matching documents, highest score first (ties ordered by RID). The same request body and limits apply to the MCP

`full_text_search` tool and the gRPC `FullTextSearch` RPC.

URL Syntax: `/api/v1/vector/{database}/fulltext`

The JSON body accepts:

- `queryText` (required) Lucene-syntax query, for example `java` or `+java -python`. Must not be blank
- `indexName` (optional) the full-text index to search. Wins over `typeName` when both are given
- `typeName` (optional) the type whose full-text index to search. Alone, it works only when the type has exactly one full-text index. An index declared on a supertype is named after the supertype
- `properties` (optional) array of indexed property names, to pick one of several full-text indexes on `typeName`
- `limit` (optional, default `10`) maximum number of results, between `1` and `1000`

One of `indexName` or `typeName` is required. The response carries `indexName`, `similarity` (`BM25` or `CLASSIC`), `count` and `results`, where each hit has `rid`, `score` and `properties`. The limit is applied inside the index (across all buckets for a query of OR-ed terms, *since v26.12.1*, per bucket otherwise), so a record deleted between the index scan and the load is skipped rather than replaced, and a search can return fewer than `limit` hits.

Responses:

- `200` OK
- `400` missing body, blank `queryText`, `limit` out of range, no index addressed, unknown or ambiguous index (the message lists the available full-text indexes), invalid Lucene syntax
- `401` missing or invalid credentials
- `403` the user has no access to the database
- `404` the database does not exist (`{"error": "Database not found"}`). A user without a grant on that name gets the `403` instead, so the answer does not reveal which databases exist

Example:

```
$ curl -X POST http://localhost:2480/api/v1/vector/mydb/fulltext \
       -d '{"typeName": "Doc", "queryText": "+graph -relational", "limit": 3}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Return:

```
{
  "indexName": "Doc[content]",
  "similarity": "BM25",
  "count": 1,
  "results": [
    { "rid": "#12:4", "score": 2.71, "properties": { "@rid": "#12:4", "@type": "Doc",
"title": "Graphs 101" } }
```

```
  ]
}
```

Begin a transaction (POST)

Begins a transaction on the server managed as a session. The response header contains the session id. Set this id in the following requests to execute them in the same transaction scope. See also /commit and /rollback.

URL Syntax: `/api/v1/begin/{database}`

Where:

- `database` is the database name

The payload, optional as a JSON object, accepts the following parameters:

- `isolationLevel` is the isolation level for the current transaction, either `READ_COMMITTED` (default) or `REPEATABLE_READ`.

A request without a body, or with a blank one, begins the transaction at the database default isolation level. A body that is not a valid JSON object (truncated, garbled, a scalar or a top-level array) is refused with `400` and no transaction is started: it is never taken as "no body", so a client that meant to ask for `REPEATABLE_READ` cannot silently get `READ_COMMITTED`.

Responses:

- `204` OK
- `400` invalid `isolationLevel` value, or a body that is not a valid JSON object
- `401` invalid credentials
- `409` transaction already started
- `500` invalid database

Example:

```
$ curl -I -X POST http://localhost:2480/api/v1/begin/school \
       --user root:arcadedb-password
```

Returns the Session Id in the response header, example:

```
arcadedb-session-id: AS-ee056170-dc9b-4956-8d71-d7cfa01900d4
```

Use the session id in the request header of further commands you want to execute in the same transaction and execute /commit to commit the server side transaction or /rollback to rollback the changes. After a period of inactivity (default is 30 seconds), the server automatically rollback and purge expired transactions.

Commit a transaction (POST)

Commits a transaction on the server. Set the session id obtained with the /begin command as a header of the request. See also /begin and /rollback.

URL Syntax: `/api/v1/commit/{database}`

Where:

- `database` is the database name

Set the session id returned from the /begin command in the request header. If the session (and therefore the server side transaction) is unknown, expired or already ended, `404` is returned: nothing was committed.

The commit ends the session, whether it succeeds or fails: a commit that fails rolls the transaction back, and the answer carries the header `arcadedb-session-closed: true`, so there is no need to send a /rollback afterwards.

A command that fails inside a session rolls the session's transaction back. When the transaction held changes, they are lost, so the session is marked as rolled back: the next command naming it is refused with `404`, and `/commit` answers `404` with a message saying that the transaction was rolled back, instead of `204`. Send a /rollback (or the `/commit`) to end it. When nothing was lost (a failed read, an error before the first write) the session goes on in a fresh transaction.

A `/commit` retried while the first one is still running (a client that timed out and sent it again) waits up to 5 seconds for the first one to finish, then answers `204` if that commit succeeded and `404` if it failed. A retry that reaches another server, or arrives after a restart, cannot know the outcome and answers `404`: send an `X-Request-Id` to make a retry safe in every case. *(Since v26.11.1: before, the retry answered `404` even when the first commit succeeded.)* The server remembers the sessions it committed for the session timeout, at most 10,000 of them, per server and in memory, which is what lets the retry of a commit that did succeed answer `204`. A `/commit` that names a session the server does not hold for the caller (never issued, already rolled back, ended by `terminate transaction`, expired, or owned by another user) answers `404` "Remote transaction session not found or expired". *(Since v26.11.1: before, it answered `204`, the answer of a real commit, although nothing was committed.)* `POST /api/v1/rollback` still answers `204` for a stale session id.

Response:

- `204` OK
- `401` invalid credentials
- `404` the session is unknown, expired or already ended, or its transaction was rolled back by a failed command
- `503` the commit conflicted with a concurrent commit that changed the same records (`ConcurrentModificationException`), or met another retryable refusal: nothing was committed and the session is closed. Begin a new transaction, read the records again and re-apply the change. The answer carries `Retry-After: 0` *(since v26.12.1, before it carried none)*: the conflict clears as soon as the competing commit has landed
- `504` *(since v26.12.1)* in an HA cluster, the commit was handed to the replication and its outcome

is unknown: it may or may not have been applied. No `Retry-After`: read the records to see whether it landed before sending the work again, see Outcome-unknown response contract

A conflict surfaces at the commit, not at the write: the write of the transaction that loses the race is answered `200`, and the conflict is detected when it commits, which is where the server can see the competing commit. A transaction at `REPEATABLE_READ` that read a record a concurrent transaction then changed and committed is the typical case (see Transactions). The Java remote client's `transaction()` runs the whole block again on it; any other client reads the `503` and the `Retry-After` header. The header is a floor, not a schedule: a client that retries in a loop still backs off, with jitter, when the same conflict comes back, as it does on a record many writers update.

Example:

```
$ curl -I -X POST http://localhost:2480/api/v1/commit/school \
       -H "arcadedb-session-id: AS-ee056170-dc9b-4956-8d71-d7cfa01900d4" \
       --user root:arcadedb-password
```

Rollback a transaction (POST)

Rollbacks a transaction on the server. Set the session id obtained with the /begin command as a header of the request. See also /begin and /commit.

URL Syntax: `/api/v1/rollback/{database}`

Where:

- `database` is the database name

Set the session id returned from the /begin command in the request header. If the session (and therefore the server side transaction) is expired, then an internal server error is returned.

The commit ends the session, whether it succeeds or fails: a commit that fails rolls the transaction back, and the answer carries the header `arcadedb-session-closed: true`, so there is no need to send a /rollback afterwards.

Response:

- `204` OK
- `401` invalid credentials
- `500` transaction expired, not found, not begun

Example:

```
$ curl -I -X POST http://localhost:2480/api/v1/rollback/school \
       -H "arcadedb-session-id: AS-ee056170-dc9b-4956-8d71-d7cfa01900d4" \
       --user root:arcadedb-password
```

Batch import vertices and edges (POST)

High-performance bulk import of vertices and edges using the GraphBatch engine under the hood. Supports JSONL and CSV input formats with streaming parsing — the server never loads the entire request body into memory, so you can push millions of records in a single request.

URL Syntax: `/api/v1/batch/{database}`

Where:

- `database` is the database name

The `Content-Type` header determines the input format:

- `application/x-ndjson` or `application/jsonl` — JSONL (newline-delimited JSON)
- `text/csv` — CSV

All vertices must appear before edges in the request body. Edges reference vertices either by temporary ID or by existing database RID.

JSONL Format

Each line is a JSON object with the following meta fields:

Field	Required	Description
`@type`	Yes	`"vertex"` (or `"v"`) for vertices, `"edge"` (or `"e"`) for edges
`@class`	Yes	The vertex or edge type name (must exist in schema)
`@id`	No	Client-assigned temporary ID for vertices. Used by edges to reference newly created vertices. Each ID names one vertex: a vertex declaring an ID an earlier vertex of the same payload already declared is refused with `400`, naming both lines (since 26.12.1; before, the later vertex silently took the ID over and the edges naming it attached to it).
`@from`	Edges only	Source vertex reference: a temporary ID (e.g., `"t1"`) or an existing RID (e.g., `"#12:0"`)
`@to`	Edges only	Destination vertex reference: a temporary ID or an existing RID

An endpoint written as a RID must be an existing vertex. A RID that does not exist, or that names a document or an edge, is refused with `400` on the line that names it (`Edge endpoint '#99:0' at line 4: Vertex #99:0 does not exist`), with the `verticesCreated`, `edgesCreated` and `partialCommit` counters, and the edges loaded before that line stay reachable (since 26.12.1; before, the load failed only at its end with `404` or `500`, without the counters, and left every edge of the load unreachable by traversal).

All other fields are treated as properties.

A control key sent on the kind of line that cannot use it is refused with `400`, naming

> the line: `@id` on an edge, `@from` or `@to` on a vertex. A key carrying **nothing** is still accepted, so JSON `null` and the empty string are ignored.

Example:

```
{"@type":"vertex","@class":"Person","@id":"t1","name":"Alice","age":30}
{"@type":"vertex","@class":"Person","@id":"t2","name":"Bob","age":25}
{"@type":"edge","@class":"KNOWS","@from":"t1","@to":"t2","since":2020}
{"@type":"edge","@class":"KNOWS","@from":"#12:0","@to":"t2","weight":0.8}
```

CSV Format

The first line is the column header. A `---` sentinel row separates the vertex section from the edge section, followed by a new header row for edges.

Example:

```
@type,@class,@id,name,age
vertex,Person,t1,Alice,30
vertex,Person,t2,Bob,25
---
@type,@class,@from,@to,since
edge,KNOWS,t1,t2,2020
```

CSV parsing supports RFC 4180 quoting (double-quote escaping). A CSV field is text, and the declared type of the property it is written to decides how it is read (since 26.12.1):

- a `STRING` or `DECIMAL` property stores the text exactly as sent: `007`, `1.50` and `False` stay `007`, `1.50` (scale 2 for a `DECIMAL`) and `False`;
- any other declared type reads the text as leniently as before and converts it: `007` and `+5` for an `INTEGER`, `TRUE` for a `BOOLEAN`, `1e3` for a `DOUBLE`;
- a property the schema does not declare is read as the JSON literal the text spells, if it spells one: `true` and `false` become booleans, a JSON number (`42`, `-2.5E-3`, `1e3`) a `Long` when it is an integer that fits one and a `Double` otherwise. Anything else stays text, so `007`, `+5`, `5.`, `TRUE` and an integer beyond the `Long` range are not rewritten.

Before 26.12.1 every field was converted by its shape before the declared type was applied, so a `STRING` property stored `7` for `007` and `1000.0` for `1e3`.

The same rule as JSONL applies to a control **value**, not to the column: a row carrying a value in a control column its kind cannot use — `@id` on an edge row, `@from`/`@to` on a vertex row — is refused with `400`, while an empty one is ignored. That keeps a single header naming all five control columns across both sections working, as long as the inapplicable columns are left empty on each row.

Query Parameters

All parameters are optional and map directly to GraphBatch builder options:

Parameter	Default	Description
`batchSize`	100000	Maximum edges buffered before auto-flush
`lightEdges`	false	If true, property-less edges are stored as connectivity only (saves ~33% I/O)
`wal`	false	Enable Write-Ahead Logging for crash safety
`parallelFlush`	true	Parallelize edge connection across async threads
`preAllocateEdgeChunks`	true	Pre-allocate edge segments on vertex creation
`edgeListInitialSize`	2048	Initial segment size in bytes (64-8192)
`bidirectional`	true	Connect the incoming side of the edges whose type is declared bidirectional; a `UNIDIRECTIONAL` type never gets it. Set to `false` only to load unidirectional edge types: an edge of a bidirectional type is then refused with HTTP 400
`commitEvery`	50000	Edges per sub-transaction within a flush
`expectedEdgeCount`	0	Hint for auto-tuning batch size

Response

The response is a JSON object with import statistics:

```
{
  "verticesCreated": 2,
  "edgesCreated": 1,
  "elapsedMs": 42,
  "idMapping": {
    "t1": "#9:0",
    "t2": "#9:1"
  }
}
```

The `idMapping` field is included only when temporary IDs (`@id`) were used. It maps each temporary ID to the actual RID assigned by the database.

Because this response is built in full before anything is sent, the mapping of a very large load is dropped rather than risk exhausting memory: past 10,000 entries the server replies with `idMappingOmitted: true` and `idMappingSize` instead. Send `idMapping=true` to demand it anyway, or use the streaming response below, which has no such limit.

Streaming response

Send `Accept: application/x-ndjson` to be answered as the load runs instead of once at the end. The response is one JSON object per line:

```
{"progress":{"phase":"vertices","verticesCreated":10000,"edgesCreated":0,"idMapping":{
```

```
"t1":"#9:0","...":"..."}}}
{"progress":{"phase":"vertices","verticesCreated":20000,"edgesCreated":0,"idMapping":{
"...":"..."}}}
{"summary":{"verticesCreated":20000,"edgesCreated":0,"elapsedMs":1234,"idMappingStream
ed":true,"idMappingSize":20000}}
```

Each `progress` line acknowledges a committed chunk and carries the temporary-ID mapping **for that chunk only**. Concatenate them to obtain the whole mapping, and check the total against `idMappingSize` on the last line. Because neither side ever holds more than one chunk, the 10,000-entry limit above does not apply here: a load of any size gets its complete mapping.

The stream always ends with either a `summary` line (success) or an `error` line (failure); a stream that ends with neither did not arrive complete. On failure the `error` line carries the same counters the buffered response would have (`verticesCreated`, `edgesCreated`, `partialCommit`), plus `status`, the status code it would have been sent under, since the status line can no longer be changed once the stream has started, and the members of the buffered error body (`error`, `exception`, `exceptionArgs`, `requestId`, and `detail` outside production mode). *(Since v26.10.2)* a retryable refusal also carries `retryAfter`, the integer seconds the buffered response would have sent as `Retry-After` (a `503`, or a `409` for a request still in flight).

Sending no `Accept` header, or any other value, returns the single buffered object described above, unchanged.

The Java driver's `RemoteGraphBatch` uses this encoding automatically, so a bulk load through it is not limited by the mapping size either. Call `withProgressListener(…)` on the builder to be notified of each chunk.

Responses:

- `200` OK
- `400` invalid input format, missing required fields, unknown temporary ID, an edge endpoint RID that does not exist or is not a vertex, vertices after edges
- `401` invalid credentials
- `409` *(since v26.12.1)* another batch load is already running on the database (`"error": "A GraphBatch load is already in progress on this database"`, `"exception": "com.arcadedb.exception.GraphBatchInProgressException"`). A database accepts one batch load at a time, and nothing of the refused load ran. No `Retry-After` is sent, because the server cannot know when the other load ends: send the load again once it has finished. Before v26.12.1 the same refusal answered `500 "Internal error"`
- `413` the body exceeds `server.httpBodyContentMaxSize`; the load is cut short and the records loaded before the cut **stay loaded**
- `503` / `504` the request could not be forwarded (for example from a follower to the leader); the records loaded before the failure **stay loaded**

A `413`, `503` or `504` does NOT mean that nothing was applied. A batch that is cut short may be partially applied: the server log carries the counts, but the HTTP answer

does not. A client that reads such an error as "rejected, nothing applied" and resends the whole payload after raising the limit or splitting it creates duplicates, unless the payload is idempotent (for example `@id`-keyed upserts). Deduplicate on the client side, or reload into a clean target database.

This endpoint is NOT atomic by design. The underlying GraphBatch commits internally in chunks for maximum throughput. The response tells you exactly how many records were committed. Treat it as a bulk-loading operation, not a transactional one. If the server dies during the load, the edges already committed are not traversable until `CHECK DATABASE FIX` relinks them *(since v26.11.1)*, see GraphBatch.

For maximum throughput, group vertices by type in the input. The endpoint batches consecutive same-type vertices into a single `createVertices()` call. Interleaving types forces smaller, less efficient batches.

Examples

JSONL import with light edges:

```
$ curl -X POST "http://localhost:2480/api/v1/batch/mydb?lightEdges=true" \
       -H "Content-Type: application/x-ndjson" \
       --data-binary @graph-data.jsonl \
       --user root:arcadedb-password
```

CSV import:

```
$ curl -X POST http://localhost:2480/api/v1/batch/mydb \
       -H "Content-Type: text/csv" \
       --data-binary @graph-data.csv \
       --user root:arcadedb-password
```

Inline JSONL with WAL enabled:

```
$ curl -X POST "http://localhost:2480/api/v1/batch/mydb?wal=true" \
       -H "Content-Type: application/x-ndjson" \
       --user root:arcadedb-password \
       -d '{"@type":"vertex","@class":"Person","@id":"p1","name":"Alice"}
{"@type":"vertex","@class":"Person","@id":"p2","name":"Bob"}
{"@type":"edge","@class":"KNOWS","@from":"p1","@to":"p2"}'
```

Python example:

```
import requests
```

```
data = (
    '{"@type":"vertex","@class":"Person","@id":"p1","name":"Alice"}\n'
    '{"@type":"vertex","@class":"Person","@id":"p2","name":"Bob"}\n'
    '{"@type":"edge","@class":"KNOWS","@from":"p1","@to":"p2"}\n'
)

resp = requests.post(
    "http://localhost:2480/api/v1/batch/mydb?lightEdges=true",
    auth=("root", "arcadedb-password"),
    headers={"Content-Type": "application/x-ndjson"},
    data=data,
)
print(resp.json())
# {'verticesCreated': 2, 'edgesCreated': 1, 'elapsedMs': 15, 'idMapping': {'p1':
'#9:0', 'p2': '#9:1'}}
```

JavaScript (Node.js) example:

```
const resp = await fetch("http://localhost:2480/api/v1/batch/mydb", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-ndjson",
    "Authorization": "Basic " + btoa("root:arcadedb-password"),
  },
  body: [
    '{"@type":"vertex","@class":"Person","@id":"p1","name":"Alice"}',
    '{"@type":"vertex","@class":"Person","@id":"p2","name":"Bob"}',
    '{"@type":"edge","@class":"KNOWS","@from":"p1","@to":"p2"}',
  ].join("\n"),
});
console.log(await resp.json());
```

5.3. gRPC API

ArcadeDB exposes a gRPC interface for high-performance, strongly-typed access from any language with gRPC support (Java, Python, Go, C++, Node.js, Rust, C#, and more). The gRPC API provides two services with 67 RPCs covering queries, record operations, transactions, graph bulk loading, vector and full-text search, time series, and server administration.

Proto definition: arcadedb-server.proto

5.3.1. Enabling the gRPC server

The gRPC module is bundled in the `full` distribution, but the server is implemented as a plugin (`com.arcadedb.server.grpc.GrpcServerPlugin`) that is **not started by default**. If the server is not listening on any gRPC port, it is because the plugin has not been registered yet.

Enable it by adding the plugin to the `arcadedb.server.plugins` setting, then start the server:

```
-Darcadedb.server.plugins=gRPC:com.arcadedb.server.grpc.GrpcServerPlugin
```

The format is `<pluginName>:<pluginFullClass>` (comma-separated when registering several plugins). Once enabled, the server listens on port `50051` by default. Confirm it started by looking for a `gRPC server started` line in the server log, or with `grpcurl -plaintext localhost:50051 list`.

Host, port, TLS, message size and the other gRPC options are the `arcadedb.grpc.*` settings - see gRPC settings for the full list. Since v26.10.2 every one of them is a registered server setting, read like `arcadedb.server.plugins` from the server configuration file, from environment variables and from JVM system properties (`-Darcadedb.grpc.*`). *(Since v26.10.2.)* `arcadedb.grpc.host` binds the standard gRPC server to every local address the name resolves to; the xDS server (`grpc.mode xds` or `both`) has no host setting and still listens on every interface.

5.3.2. Services

ArcadeDbService

The main service for database operations: querying, record CRUD, bulk inserts, transactions, graph bulk loading, search, and time series.

RPC	Request	Response	Description
`StreamQuery`	`StreamQueryRequest`	stream `QueryResult`	Execute a query and stream results back row by row
`ExecuteCommand`	`ExecuteCommandRequest`	`ExecuteCommandResponse`	Execute a DDL/DML command (CREATE, INSERT, UPDATE, DELETE)
`ExecuteQuery`	`ExecuteQueryRequest`	`ExecuteQueryResponse`	Execute a query and return all results at once
`CreateRecord`	`CreateRecordRequest`	`CreateRecordResponse`	Create a new record (vertex, edge, or document)
`UpdateRecord`	`UpdateRecordRequest`	`UpdateRecordResponse`	Update an existing record by RID
`LookupByRid`	`LookupByRidRequest`	`LookupByRidResponse`	Look up a record by its Record ID
`DeleteRecord`	`DeleteRecordRequest`	`DeleteRecordResponse`	Delete a record by RID
`BulkInsert`	`BulkInsertRequest`	`InsertSummary`	Insert multiple records in a single request
`InsertStream`	stream `InsertChunk`	`InsertSummary`	Client-streaming insert for large datasets
`InsertBidirectional`	stream `InsertRequest`	stream `InsertResponse`	Bidirectional streaming for insert with per-record feedback
`BeginTransaction`	`BeginTransactionRequest`	`BeginTransactionResponse`	Begin an explicit transaction

RPC	Request	Response	Description
CommitTransaction	CommitTransactionRequest	CommitTransactionResponse	Commit the current transaction
RollbackTransaction	RollbackTransactionRequest	RollbackTransactionResponse	Roll back the current transaction
GraphBatchLoad	stream GraphBatchChunk	GraphBatchResult	Client-streaming bulk load of vertices and edges (see below)
VectorSearch	VectorSearchRequest	VectorSearchResponse	k-nearest-neighbor search over a dense or sparse vector index
HybridSearch	HybridSearchRequest	HybridSearchResponse	Fused vector, full-text, and graph-expansion search
FullTextSearch	FullTextSearchRequest	FullTextSearchResponse	Lucene-syntax search over a FULL_TEXT index
TimeSeriesWrite	TimeSeriesWriteRequest	TimeSeriesWriteSummary	Write a batch of time-series points
TimeSeriesWriteStream	stream TimeSeriesWriteChunk	TimeSeriesWriteSummary	Client-streaming time-series ingest
TimeSeriesQuery	TimeSeriesQueryRequest	stream TimeSeriesQueryResult	Range query, raw or aggregated into buckets, streamed back
TimeSeriesLatest	TimeSeriesLatestRequest	TimeSeriesLatestResponse	Latest sample of a time-series type, optionally filtered by tags

A record that a BeforeRecordCreateListener or a BeforeRecordUpdateListener vetoed was refused on purpose, and the replies say so instead of reporting it as written. The bulk inserts (BulkInsert, InsertStream, InsertBidirectional) count it as ignored, and a vetoed update under CONFLICT_UPDATE does not insert a second record for the key. UpdateRecord answers success = false and updated = false, which is what the drivers return to the caller as "the record was updated". CreateRecord has to answer a RID, so it fails with INVALID_ARGUMENT naming the refusal.

Before v26.11.1 UpdateRecord answered success = true and updated = true for an update a listener vetoed, and CreateRecord of a vetoed edge failed with INTERNAL carrying the text of a NullPointerException.

Graph batch load

GraphBatchLoad streams GraphBatchChunk messages, each carrying GraphBatchRecord entries of kind VERTEX or EDGE. The first chunk must name the database and carry the GraphBatchOptions (batch size, light edges, WAL, commit frequency, vertex_batch_size, commit retries, and return_id_mapping); options on later chunks are ignored. Vertices declare a temp_id, and edges reference their endpoints by temp_id or by an existing RID (#bucket:pos) in from_ref / to_ref. An endpoint given as a RID must be an existing vertex: a RID that does not exist, or that names a document or an edge, fails the load with INVALID_ARGUMENT naming the RID when the record is read (since 26.12.1; before, it failed only at the end of the load with NOT_FOUND or INTERNAL). All VERTEX records must precede every EDGE record across the whole stream: interleaving is an error. On a replicated cluster the load must run on the

leader; a follower refuses it.

`GraphBatchResult` reports `vertices_created`, `edges_created`, `elapsed_ms` and, depending on `return_id_mapping`, the `temp_id` to RID map in `id_mapping` (when the map is not returned, `id_mapping_omitted` is set and `id_mapping_size` still reports its size). The load commits incrementally, so a failure is not a rollback: the counts already durable at the point of failure travel in the `arcadedb-graph-batch-result-bin` trailer of the failed call, with `partial_commit` set.

Vector, hybrid, and full-text search

These three unary RPCs run the same server-side implementation as the HTTP `/api/v1/vector/*` routes and the MCP search tools, so a request that is legal on one protocol is legal on all of them and is rejected with the same message. Every hit is a `SearchHit` with `rid`, `record`, and either `distance` (dense index, lower is better) or `score` (sparse, full-text, or fused, higher is better); the response's `scoring` field says which. The hit's `record` properties are encoded like the values of a query result, so a list, set or array property is a `list_value` and an embedded map is a `map_value` (since 26.11.1).

RPC	Key request fields
`VectorSearch`	`index_name` (an `LSM_VECTOR` index, or an `LSM_SPARSE_VECTOR` index when `sparse` is set), `query_vector`, `query_indices` (sparse dimension ids), `k` (1 to 1000, default 10), `ef_search` (dense only), `filter` (read-only SQL `WHERE` predicate applied to a bounded candidate window). ArcadeDB does not generate embeddings.
`HybridSearch`	The vector fields above (`vector_index_name` instead of `index_name`), plus an optional full-text leg (`fulltext_query` and `fulltext_index_name`, which must be given together), `fusion_strategy` (`RRF` by default, `DBSF`, or `LINEAR`), per-leg `weights` (keys `vector`, `fulltext`, `expand`), and an optional `expand` graph leg (`edge_types`, `direction` `out`/`in`/`both`, `max_depth` 1 to 3). The `expand` leg requires `RRF`.
`FullTextSearch`	`query_text` (Lucene syntax, not blank), the index addressed by `index_name` or by `type_name` plus optional `properties` (`index_name` wins when both are set), and `limit` (1 to 1000, default 10).

All three accept an optional `transaction`: when it names a live transaction the search reads exactly what that transaction reads, including its uncommitted changes. A transaction id the server no longer knows is refused with `FAILED_PRECONDITION`. See also Full-text index.

Time series

The time-series RPCs read and write `TIMESERIES` types (see Time series).

- **`TimeSeriesWrite` / `TimeSeriesWriteStream`**: each `TimeSeriesPoint` carries `type` (the measurement, defaulting to the request or chunk `type`), `timestamp`, and `tags` / `fields` maps of `GrpcValue`. `precision` defaults to `TS_PRECISION_MILLISECONDS` (unlike the HTTP line-protocol endpoint, which defaults to nanoseconds). The streaming variant needs `database` on the first chunk only and applies back-pressure, pulling one chunk at a time. A write is not atomic: each measurement's batch commits as it is appended. `TimeSeriesWriteSummary` reports `received`, `written`, and `dropped`

(`written + dropped == received`), and lists dropped points' types under `unknown_types`, `non_time_series_types`, or `unavailable_types`. *(Since v26.10.2.)* `undeclared_keys` (field 8) lists, as `<measurement>.<key> (tag|field)`, the tag or field keys a measurement's `TIMESERIES` type does not declare in that role: the points carrying them are dropped unless the database sets `arcadedb.timeSeriesUndeclaredKeys=ignore`. `unreadable_values` (field 9) lists, as `<measurement>.<column> (tag|field): <reason>`, values a column cannot store. Both lists are capped at 100 entries, while `dropped` still counts every point. The HTTP line-protocol endpoint reports the same lists as `undeclaredKeys` and `unreadableValues` in its `400` partial-write body, described by the `TimeSeriesWriteError` schema of the OpenAPI definition.

- **`TimeSeriesQuery`**: `type`, inclusive `from_timestamp` / `to_timestamp` (unset means unbounded), `fields` projection (an unknown name is refused with `INVALID_ARGUMENT`), `tags` (`TimeSeriesTagFilter`, a conjunction of equality predicates on TAG columns; a non-TAG name is refused with `INVALID_ARGUMENT`), `limit`, `batch_size` (rows per streamed message, default 1000), and optional `aggregation` (`bucket_interval_ms` > 0, at least one `TimeSeriesAggregationRequest` of `TS_AGG_SUM`, `AVG`, `MIN`, `MAX` or `COUNT`, and an optional `bucket_origin_ms`). The answer streams as `TimeSeriesQueryResult` messages carrying `rows` (raw) or `buckets` (aggregated); the last message has `last` set, and `truncated` when the limit cut the answer short. The server-side ceiling `arcadedb.server.grpcTimeSeriesMaxResultRows` (default 1000000) cannot be raised by the client: a `limit` above it is refused with `RESOURCE_EXHAUSTED` before the first message.
- **`TimeSeriesLatest`**: `type` and optional `tags`. `TimeSeriesLatestResponse` returns `columns`, `found`, and the `latest` row (unset when `found` is `false`).

`TimeSeriesQuery` and `TimeSeriesLatest` accept an optional `transaction`; when set, the read sees that transaction's uncommitted points. An unresolvable transaction id is refused with `FAILED_PRECONDITION` (the HTTP time-series read routes instead fall back to reading outside the transaction).

ArcadeDbAdminService

Administration service for server and database management. Every admin RPC runs the same implementation as the HTTP control plane, so the two protocols behave the same way.

Authentication and authorization. Every request is authenticated from its `credentials` field, except `Health` and `Ready`, which are unauthenticated like `GET /api/v1/health` and `GET /api/v1/ready`. `Ping`, `GetServerInfo`, `ListDatabases`, `ExistsDatabase`, and `GetDatabaseInfo` need any valid account; `GetProgress` needs an account granted the named database; every other RPC requires the `root` user and otherwise fails with `PERMISSION_DENIED`.

Leader routing. On a replicated cluster, `CreateDatabase`, `DropDatabase`, `CreateUser`, `UpdateUser`, `DeleteUser`, `SaveGroup`, `DeleteGroup`, `CreateApiToken`, `DeleteApiToken`, `RestoreBackup`, `RestoreDatabase`, and `ImportDatabase` are refused on a follower with `FAILED_PRECONDITION` and the leader's address in the `arcadedb-leader-*` trailers. gRPC has no request proxy, so the client redirects itself. *(Since v26.10.2.)* The streaming inserts of `ArcadeDbService` (`InsertStream` and `InsertBidirectional`, and the rows of a `BulkInsert`) report a write a follower refused differently: the refused row or commit comes back as an `InsertError` with `code NOT_LEADER` and the leader's address in `leader_address`, rather than through the `arcadedb-leader-*` trailers (see Insert operations).

Server info and databases

RPC	Request	Response	Description
`Ping`	`PingRequest`	`PingResponse`	Health check
`GetServerInfo`	`GetServerInfoRequest`	`GetServerInfoResponse`	Server version, uptime, and configuration
`ListDatabases`	`ListDatabasesRequest`	`ListDatabasesResponse`	List all databases
`ExistsDatabase`	`ExistsDatabaseRequest`	`ExistsDatabaseResponse`	Check if a database exists
`CreateDatabase`	`CreateDatabaseRequest`	`CreateDatabaseResponse`	Create a new database (see below)
`DropDatabase`	`DropDatabaseRequest`	`DropDatabaseResponse`	Drop an existing database (see below)
`OpenDatabase`	`OpenDatabaseRequest`	`OpenDatabaseResponse`	Open the named database
`CloseDatabase`	`CloseDatabaseRequest`	`CloseDatabaseResponse`	Close the named database's open instance; the files stay, and a later request reopens it
`AlignDatabase`	`AlignDatabaseRequest`	`AlignDatabaseResponse`	Run `ALIGN DATABASE` on the named database
`GetDatabaseInfo`	`GetDatabaseInfoRequest`	`GetDatabaseInfoResponse`	Schema, types, and database statistics
`GetProgress`	`GetProgressRequest`	`GetProgressResponse`	Long-running operations (CHECK DATABASE, REBUILD INDEX, backup, import, ...) running on the database, as `GET /api/v1/progress/{database}`. Empty when nothing runs

`GetDatabaseInfoResponse.records` is the sum of the record counts of the database's types, read from the bucket counters rather than by a scan. *(Since v26.11.1)* a type whose records cannot be counted - a time series type whose storage engine did not open, for one - is named in `uncounted_types` and left out of the sum, so a non-empty list says the total is short by an unknown amount rather than that the database holds fewer records; and a schema that cannot be read fails the call instead of answering zero classes (issue #10877).

`CreateDatabase` and `DropDatabase` answer the no-op case the way the HTTP `create database` and `drop database` commands do: a name already taken is refused with `ALREADY_EXISTS`, a name that does not exist with `NOT_FOUND`. Set `if_not_exists` on `CreateDatabaseRequest`, or `if_exists` on `DropDatabaseRequest`, to make the call idempotent instead. Either way the response says what happened: `CreateDatabaseResponse.created` is `true` only when this call created the database, and `DropDatabaseResponse.dropped` only when it dropped one, so a provisioning tool can tell "created for me" from "already there" rather than reading an empty OK as ownership. Database names are exact and never case-folded. The Java client exposes the two forms as `createDatabase` / `createDatabaseIfMissing` and `dropDatabase` / `dropDatabaseIfExists`; the `IfMissing` / `IfExists` variants return the boolean.

(Since v26.11.1) `ListDatabases`, `ExistsDatabase`, `GetDatabaseInfo`, `CreateDatabase` with `if_not_exists` and `DropDatabase` with `if_exists` see the databases that are on disk but not open (closed for idleness, or not opened at startup because `arcadedb.server.databaseLoadAtStartup` is `false`) as existing, since the first request that names one opens it. Before, they answered as if such a database did not exist. A database closed with the `close database` server command, being restored or being dropped is still not listed. See Databases on disk that are not open.

(Since v26.12.1, issue #11310) A database closed with the `close database` server command, being restored or being dropped exists for `ExistsDatabase`, `CreateDatabase` and `DropDatabase` too: `ExistsDatabase` answers `true`, `CreateDatabase` with `if_not_exists` answers `created=false` (and without it `ALREADY_EXISTS`), `DropDatabase` drops a database closed with `close database` and refuses one a restore or a drop holds with `ABORTED` and the reason, and `GetDatabaseInfo` answers `UNAVAILABLE` with the reason rather than `NOT_FOUND`. Before, they answered as if it did not exist: `DropDatabase` with `if_exists` answered `dropped=false` and left the database on disk, while `CreateDatabase` with `if_not_exists` failed `ALREADY_EXISTS`.

Users, groups, and API tokens

RPC	Request	Response	Description
`CreateUser`	`CreateUserRequest`	`CreateUserResponse`	Create a new user. A name already taken is `ALREADY_EXISTS` *(since 26.10.2)*
`UpdateUser`	`UpdateUserRequest`	`UpdateUserResponse`	Update a user's `password` and/or `databases` grants. Each field is optional and an absent one is left untouched; a present but empty `databases` clears the grants
`DeleteUser`	`DeleteUserRequest`	`DeleteUserResponse`	Delete a user
`ListUsers`	`ListUsersRequest`	`ListUsersResponse`	List users with their per-database groups. Password hashes are never included
`ListGroups`	`ListGroupsRequest`	`ListGroupsResponse`	The whole group document, as JSON in `groups_json`
`SaveGroup`	`SaveGroupRequest`	`SaveGroupResponse`	Create or replace one group of a database (`*` for all) from `group_json`
`DeleteGroup`	`DeleteGroupRequest`	`DeleteGroupResponse`	Delete one group of a database
`ListApiTokens`	`ListApiTokensRequest`	`ListApiTokensResponse`	List API tokens as `ApiTokenInfo` (hash, last four characters, expiry, `expired` flag), never the token itself

RPC	Request	Response	Description
`CreateApiToken`	`CreateApiTokenRequest`	`CreateApiTokenResponse`	Mint an API token. The plaintext `token` is returned once only. Refused with `FAILED_PRECONDITION` unless the channel uses TLS or the client is on loopback; a duplicate name is `ALREADY_EXISTS`
`DeleteApiToken`	`DeleteApiTokenRequest`	`DeleteApiTokenResponse`	Revoke a token by its `token_hash` (the plaintext token is not accepted)

Settings

RPC	Request	Response	Description
`SetServerSetting`	`SetServerSettingRequest`	`SetServerSettingResponse`	Set a server setting. `key` and `value` are separate fields, so a value with spaces needs no quoting
`SetDatabaseSetting`	`SetDatabaseSettingRequest`	`SetDatabaseSettingResponse`	Set a setting of one database

Backup, restore, and import

RPC	Request	Response	Description
`GetBackupConfig`	`GetBackupConfigRequest`	`GetBackupConfigResponse`	Auto-backup configuration as JSON, and whether the auto-backup plugin is running
`SetBackupConfig`	`SetBackupConfigRequest`	`SetBackupConfigResponse`	Save the auto-backup configuration from `config_json`
`ListBackups`	`ListBackupsRequest`	`ListBackupsResponse`	Backup archives of a database, with retention totals
`TriggerBackup`	`TriggerBackupRequest`	`TriggerBackupResponse`	Back up a database now; returns the archive path
`DeleteBackup`	`DeleteBackupRequest`	`DeleteBackupResponse`	Delete one archive by plain file name (path separators and traversal are refused)
`RestoreBackup`	`RestoreBackupRequest`	stream `RestoreProgress`	Restore a database's backup archive into `target_database`; `overwrite` replaces an existing target only after the restore succeeded
`RestoreDatabase`	`RestoreDatabaseRequest`	stream `RestoreProgress`	Create a new database from an archive at a `url`

RPC	Request	Response	Description
`ImportDatabase`	`ImportDatabaseRequest`	stream `ImportProgress`	Create a new database and import the source at a `url` into it; progress carries importer lines and `parsed` / `vertices` / `edges` counters. A name already taken is `ALREADY_EXISTS` *(since 26.10.2)*

The three restore and import RPCs are server-streaming: they send progress messages while the operation runs, then one final message with `completed = true` (`ImportProgress.result_json` carries the importer's final report). A failure ends the stream with an error status instead. Cancelling the call stops the progress messages, not the operation. Unless `arcadedb.server.restoreImportAllowLocalUrls` is enabled, only `http`/`https` URLs to non-private hosts are accepted. See also Backup.

Query profiler

RPC	Request	Response	Description
`ProfilerStart`	`ProfilerStartRequest`	`ProfilerStateResponse`	Start recording. `timeout_seconds` of 0 or less applies the server default; the effective timeout is returned
`ProfilerStop`	`ProfilerStopRequest`	`ProfilerDocumentResponse`	Stop recording and return the run as JSON
`ProfilerReset`	`ProfilerResetRequest`	`ProfilerStateResponse`	Reset the profiler
`ProfilerResults`	`ProfilerResultsRequest`	`ProfilerDocumentResponse`	Current profiler results as JSON, without stopping
`ProfilerList`	`ProfilerListRequest`	`ProfilerListResponse`	Saved profiler runs on disk, newest first
`ProfilerLoad`	`ProfilerLoadRequest`	`ProfilerDocumentResponse`	Load a saved run by file name

Server lifecycle and cluster

RPC	Request	Response	Description
`GetServerEvents`	`GetServerEventsRequest`	`GetServerEventsResponse`	Server events as a JSON array, from the current event file or a named one, plus the list of event files

RPC	Request	Response	Description
`Shutdown`	`ShutdownRequest`	`ShutdownResponse`	Shut down this server (empty `server_name`) or the named HA peer. The local shutdown is scheduled asynchronously, so the response arrives first
`DisconnectCluster`	`DisconnectClusterRequest`	`DisconnectClusterResponse`	Disconnect this server from the cluster. `FAILED_PRECONDITION` when the server does not run HA
`AcceptDivergedDatabase`	`AcceptDivergedDatabaseRequest`	`AcceptDivergedDatabaseResponse`	*Since 26.10.2.* Root only. Lifts the quarantine, and its read floor, standing on `database` of a node that is the only voter of its cluster (or whose every voter holds the same database quarantined), accepting the local copy as it is without a resync. The response carries `local_server`, `applied_index`, `divergence_cause` and `read_floor`. `NOT_FOUND` when nothing stands on the database, `FAILED_PRECONDITION` when the node has peers it can resync from or runs without HA, `INVALID_ARGUMENT` on a malformed name. The RPC twin of `POST /api/v1/cluster/accept-diverged/{database}`, see Lifting a quarantine without a resync

RPC	Request	Response	Description
`AcceptStaleSnapshot`	`AcceptStaleSnapshotRequest`	`AcceptStaleSnapshotResponse`	*Since 26.10.2.* Root only. Lifts the node-wide stale-snapshot read floor of a node that is the only voter of its cluster, accepting its databases as they are without a resync. The response carries `local_server`, `read_floor`, `snapshot_index` and `applied_index`. `NOT_FOUND` when no floor stands, `FAILED_PRECONDITION` when the node has peers it can resync from, while a snapshot download runs, or without HA. The RPC twin of `POST /api/v1/cluster/accept-stale-snapshot`, see Lifting the stale-snapshot read floor
`ConnectCluster`	`ConnectClusterRequest`	`ConnectClusterResponse`	Join the server at `server_address` (an `arcadedb.ha.serverList` entry) to the cluster. `FAILED_PRECONDITION` when HA is not running or cannot change membership at runtime; empty address is `INVALID_ARGUMENT`
`ListSessions`	`ListSessionsRequest`	`ListSessionsResponse`	Open HTTP authentication sessions, including their bearer tokens. Enable TLS before calling it across an untrusted network

Health probes

RPC	Request	Response	Description
`Health`	`HealthRequest`	`HealthResponse`	Unauthenticated liveness probe; `ok` is `false` only when the HA layer escalated a crash loop
`Ready`	`ReadyRequest`	`ReadyResponse`	Unauthenticated readiness probe; a not-ready node answers `ready = false` with a `reason` rather than an error status

Error statuses

A refused admin call answers the status of the refusal whatever the server wrapped it in on its way out: a name already taken is `ALREADY_EXISTS`, a user, group or token that does not exist `NOT_FOUND`, a security policy refusing an argument `PERMISSION_DENIED`, failed authentication `UNAUTHENTICATED`, a backup, restore or import already running on the database `ABORTED`, a malformed argument `INVALID_ARGUMENT`, an operation this server's configuration does not support `FAILED_PRECONDITION`, and a transient failure such as a Raft quorum not reached `UNAVAILABLE`, which a client retries. The description is the refusal's own sentence, also in production mode, except for a generic argument, authentication or retry failure raised underneath another error: that one keeps its status but its text is concealed in production mode, because it may come from a library and carry internal details. A statement stopped on request is `CANCELLED`, as on the data-plane RPCs (see Cancellation). Only an unexpected server fault is `INTERNAL`, and in production mode its description is concealed.

On the data-plane RPCs, the `arcadedb-exception-class` trailer names the exception that decided the status rather than the one wrapped around it, so the Java client rebuilds the engine's own type: a `SchemaException` for a missing type (not the `RecordNotFoundException` that `NOT_FOUND` alone reads as), or a `DuplicatedKeyException` carrying its index name.

(Since v26.11.1) A command that creates, or renames onto, a schema object name already taken (a type, property, index, bucket, view, ...) answers `ALREADY_EXISTS`, with the `arcadedb-already-exists-kind` trailer naming the kind. The Java client rebuilds it as `AlreadyExistsException`, a `SchemaException`. Before, it was `INTERNAL`, or `NOT_FOUND` for some of them. The admin RPCs answer a taken user, token or database name as before, and the Java client raises `RemoteAlreadyExistsException` for it on every admin call, `ImportDatabase` included.

In v26.10.2 and earlier, only the leader-redirect refusal was recognised under a wrapper. Any other refusal the server raised under one (for example inside a commit or a backup) answered `INTERNAL`, and in production mode its sentence was replaced by the wrapper's. A call that failed on a momentary loss of quorum was therefore not retried by the client. A search refused by the per-type ACL answered `INTERNAL` the same way. On the data plane, a wrapped duplicated key lost its index-name trailer, and a wrapped maintenance-slot conflict answered `INTERNAL` instead of `ABORTED`.

5.3.3. Cancellation and running statements

Since 26.10.2. The `ArcadeDbService` RPCs (such as `ExecuteQuery`, `ExecuteCommand`, `StreamQuery`, `BulkInsert`, the search RPCs and `TimeSeriesQuery`, but not `BeginTransaction` and `RollbackTransaction`) are listed, while they run, by the HTTP `list queries` server command and by Cypher `SHOW TRANSACTIONS`, with `protocol grpc`, and can be stopped by `terminate query`, `TERMINATE TRANSACTIONS` or Studio. A `BulkInsert` stops between two rows.

The client-streaming loads (`InsertStream`, `InsertBidirectional`, `GraphBatchLoad` and `TimeSeriesWriteStream`) are listed from the first chunk, which names the database, to the end of the stream, including while the stream is idle between two chunks, and `terminate query` stops them there too. A terminated load ends with status `CANCELLED`, and what the load already wrote depends on its transaction mode:

- `PER_STREAM` and `PER_REQUEST`: the open transaction is rolled back, so nothing the stream sent is

committed.

- `PER_BATCH`: the open batch is rolled back; batches committed before the termination stay committed.
- `PER_ROW`: the rows committed before the termination stay, the rest are not applied.
- A load that writes into a client transaction (`BeginTransaction`) rolls that transaction back and ends it.
- `GraphBatchLoad` commits incrementally: what is already durable is reported in the `arcadedb-graph-batch-result-bin` trailer with `partial_commit` set. `TimeSeriesWriteStream` stops appending; batches already appended stay.

A client that cancels a call, or whose deadline expires, or whose connection drops, terminates the server-side work too: the server stops it at its next check instead of running it to the end for an answer nobody reads. A `StreamQuery` or `TimeSeriesQuery` that the client cancels, for example because it stopped reading once it had enough rows, stops at once and gives back its worker thread, its result set and its slot of the query admission gate (`arcadedb.queryMaxConcurrent`). Before 26.12.1 it kept them until `server.grpcStreamWriteTimeoutMs` expired.

A terminated call ends with status `CANCELLED` and the trailer `arcadedb-exception-class` set to `com.arcadedb.exception.QueryTerminatedException`; the gRPC Java client raises that exception, which it does not retry. A statement terminated inside a client-managed transaction (`BeginTransaction`) ends that transaction: it is rolled back, and a later `CommitTransaction` answers `committed = false`.

5.3.4. Metrics

The gRPC server records three meters per RPC, exported with the other server meters (see Metrics depth):

Meter	Tags	Meaning
`grpc.requests.total`	`method`	Calls received
`grpc.request.duration`	`method`, `status`	Time from the call to its close
`grpc.errors.total`	`method`, `status`	Calls that failed: closed with a status other than `OK`, or closed with `OK` but answering a failure in the response

(Since v26.11.1) Authentication faults are visible. When the server cannot read the credentials of an admin request, or the security backend fails unexpectedly, the call still fails closed as `UNAUTHENTICATED`, but it is logged at `SEVERE` with the cause and the method name (before, at `FINE`, which the default level hides) and counted on the Micrometer counter `grpc.auth.faults.total`, tagged `stage` = `admin-credential-read`, `authenticate` or `intercept`. The counters are registered at zero. An unexpected failure inside the security backend answers "Authentication error: the server could not evaluate the credentials (server fault, see the server log)" on the data plane and the admin service, instead of "Invalid credentials"; a wrong password, a missing grant and a lockout answer as before. A linkage error such as `NoClassDefFoundError` during authentication is reported the same way.

(Since v26.11.1) `grpc.errors.total` counts every call closed with status `OK` whose response message

declares a top-level boolean `success` field and sends it `false`, such as an `UpdateRecord` or a `DeleteRecord` that a listener vetoed. Such a call is counted with `status OK`. Before, only an `ExecuteCommandResponse` with `success = false` was counted, and the other in-band failures read as successes (issue #11054).

5.3.5. Message Types

Connection & Authentication

`DatabaseCredentials`	Database name, username, and password for per-request authentication

Value Types

The `GrpcValue` message uses a `oneof` to represent any ArcadeDB value:

Field	ArcadeDB Type
`bool_value`	Boolean
`int32_value`	Integer
`int64_value`	Long
`float_value`	Float
`double_value`	Double
`string_value`	String
`bytes_value`	Binary / byte array
`timestamp_value`	Datetime (google.protobuf.Timestamp)
`list_value`	List / array
`map_value`	Map / embedded document
`link_value`	RID reference (bucket:position)
`decimal_value`	Decimal (string-encoded)
`null_value`	Null

A `GrpcValue` can also carry a `logical_type` tag that says which ArcadeDB type it stands for. A client that ignores the tag still reads the plain string or timestamp.

`logical_type`	Carried as
`date`, `datetime`	`timestamp_value`, anchored at UTC
`zoned_datetime`	`string_value` with the zone kept, for example `2026-01-01T00:30:00+01:00[Europe/Rome]`
`offset_time`, `local_time`	`string_value`, ISO 8601 time of day
`duration`	`string_value`, ISO 8601 duration, for example `P1Y2M3DT4H5M6.789S`

Text tagged with one of these types that cannot be parsed is returned as a plain string.

(Since v26.12.1) A date, time or datetime returned by an openCypher query - a `date()`, `localdatetime()`, `datetime()`, `localtime()` or `time()` result, or a stored `DATE` or `DATETIME` read as `n.prop` - is carried as the same kind and `logical_type` as the same value from SQL, also inside a list or a map: a Cypher date or local datetime is a `timestamp_value` tagged `date` or `datetime`, a zoned datetime, a time and a local time are a tagged `string_value`. Before, they were sent as an untagged `string_value` (issue #11955).

`GrpcRecord` wraps a record with its RID, type name, and a map of property name to `GrpcValue`. A property stored with a `null` value is in the map, with a `GrpcValue` that has no value set, so a client reads it back as `null` *(since v26.12.1)*. Earlier versions left it out of the map, so a record read with `LookupByRid` or a streamed query did not list the property at all (issue #12197).

Enums

Enum	Values
`RetrievalMode`	`CURSOR` (streaming), `MATERIALIZE_ALL` (all at once), `PAGED` (same as `CURSOR`)
`ProjectionEncoding`	`PROJECTION_AS_LINK`, `PROJECTION_AS_MAP`, `PROJECTION_AS_JSON`
`TransactionIsolation`	`READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`
`ConflictMode`	`CONFLICT_ERROR`, `CONFLICT_UPDATE`, `CONFLICT_IGNORE`, `CONFLICT_ABORT`
`TransactionMode`	`PER_REQUEST`, `PER_BATCH`, `PER_STREAM`, `PER_ROW`, `NONE`

`BeginTransaction` accepts `READ_COMMITTED` and `REPEATABLE_READ`, the two levels the engine provides, and `READ_UNCOMMITTED`, which is the proto default and runs at the stronger `READ_COMMITTED`. The level the transaction really runs at is returned in `BeginTransactionResponse.effective_isolation`. `SERIALIZABLE` is not supported: it is refused with `INVALID_ARGUMENT`, as is a value the server does not know, instead of starting the transaction at a weaker level than the one asked for *(since v26.11.1)*. HTTP, SQL, openCypher and PostgreSQL refuse it the same way.

`PAGED` retrieval is the same as `CURSOR` *(since v26.12.1)*: the query runs once, in any language, and its rows are streamed in batches of `batch_size`, in the order the query returns them. Earlier versions re-ran a SQL query with `SKIP`/`LIMIT` for every page, so the time of a stream grew with the square of its rows. As a consequence a `PAGED` stream now carries the facets of a `FACETS` query on its last batch, `_skip` and `_limit` are ordinary parameter names, and a query with no `ORDER BY` is no longer sorted by `@rid`: add an `ORDER BY` when you need an order.

Projected records

`StreamQueryRequest`, `ExecuteQueryRequest` and `ExecuteCommandRequest` accept an optional `ProjectionSettings` that says how a record the query projects as a value is sent:

Field	Meaning
include_projections	When false, a projected record is sent as its RID (link_value) only
projection_encoding	PROJECTION_AS_LINK sends the RID, PROJECTION_AS_MAP a map_value with @rid, @type and the properties, PROJECTION_AS_JSON the record's JSON in bytes_value with logical_type json
soft_limit_bytes	A soft bound, in bytes, on the projected records of the request. Past it, the remaining properties of a record are left out (MAP), and a record whose JSON is larger is sent as its RID (JSON). Unset or 0 means no limit

A request without ProjectionSettings gets the records in JSON, with no limit.

(Since v26.11.1) A projected value that holds records - a graph traversal (SELECT out() AS n, and likewise in(), outE(), inE(), both(), bothE()), a list, a map, an array or a nested result row - is encoded like a top-level projected record, with the request's ProjectionSettings, and soft_limit_bytes is one budget for the whole response, shared by every record, nested ones included, in both MAP and JSON encoding. A traversal is a list_value, empty for a vertex with no edges. A nested result row that wraps a record is sent as that record, @rid and @type included; if the row has columns of its own (a subquery that projects only some properties, for instance), those columns are sent next to @rid and @type instead of the record's properties. This is what HTTP returns for the same query.

Before v26.11.1 out(), in(), outE() and inE() reached a gRPC client as a Java class name string, such as com.arcadedb.graph.GraphEngine$10@63e44da5, and the records inside a list (both()) as bare RIDs whatever the ProjectionSettings asked for (issues #11044, #11048).

Insert Operations

BulkInsertRequest	Batch of records with type name, properties, and insert options
InsertChunk	A chunk of records for client-streaming inserts
InsertRequest / InsertResponse	Per-record request/response for bidirectional streaming
InsertSummary	Result summary: received, inserted, updated, ignored, failed, the errors list of InsertError, execution_time_ms, started_at and finished_at
InsertError	One failed row or commit, see below
InsertOptions	Transaction mode, conflict handling, and batch size configuration

A row that cannot be applied does not fail the call: it is counted in failed and reported as an InsertError, and the rest of the batch still goes in.

Under PER_BATCH a batch is server_batch_size rows of the stream, whatever the size of the chunks the client sends: a batch can span several chunks, and several batches can be committed inside one chunk. When a commit fails (a duplicated key under CONFLICT_ERROR, for one), only the open

transaction is lost: the open batch under PER_BATCH, the failing row under PER_ROW, the whole stream under PER_STREAM and PER_REQUEST. The summary matches what is stored: inserted and updated count every row committed before the failure, including the rows committed inside the failing chunk, and the rows lost with the failed transaction, or never reached in the failing chunk, are counted in failed, so received = inserted + updated + ignored + failed. The acknowledgement of a failing InsertBidirectional chunk reports its rows committed before the failure as inserted, and the session goes on with the next chunk in a new transaction. *(Since v26.12.1.)* Before, PER_BATCH never committed a batch spanning chunks, so a stream sent in chunks smaller than server_batch_size ran as one transaction and a failure rolled back every row of it, and rows committed inside a failing chunk were stored but not counted as inserted, so a client retrying the rows reported as not inserted could duplicate them (issue #12130). A client that replays a failed InsertBidirectional chunk whole can still duplicate the rows the acknowledgement reported as inserted: replay only the rest (issue #12576).

Field	Number	Meaning
message	1	The engine's error text. In production server mode it is replaced by a generic message, see Error details in production mode
row_index	2	Index of the row within the chunk, or -1 when the error belongs to a transaction or a commit rather than to one row
code	3	Machine-readable code, see below
field	4	The offending field, when known
leader_address	5	*Since 26.10.2.* Set only when code is NOT_LEADER: the address of the cluster leader to send the load to

Code	Meaning
CONFLICT	The row's key is already taken (ConflictMode CONFLICT_ERROR or CONFLICT_ABORT), or a commit found a duplicated key
DB_ERROR	Any other failure of one row or chunk
NOT_LEADER	*Since 26.10.2.* The node is a follower and refused the write; leader_address names the leader. Before, a row refusal was DB_ERROR and a commit refusal COMMIT_FAILED
COMMIT_FAILED	A transaction commit failed for a reason other than a duplicated key or a leader refusal
CONTRACT_VIOLATION	The stream broke the protocol, for example its options changed mid-stream
UNKNOWN_TRANSACTION	The chunk named a transaction id the server does not know or has expired
INVALID_KEY_COLUMN	key_columns contains an empty name
MISSING_ENDPOINTS	An edge row has no out or no in

Command Results & Write Statistics

`ExecuteCommand` returns an `ExecuteCommandResponse`:

Field	Meaning
`success` / `message`	Whether the command succeeded, and an optional message
`affected_records`	Server-computed count of affected elements and numeric scalars
`execution_time_ms`	Server-side execution time
`records`	Optional row payload, returned when `return_rows` is set on the request
`stats`	A `QueryUpdateStats` message, **present only for write commands that mutated data**

`QueryUpdateStats` carries the same write counters the HTTP `stats` object and the Bolt result summary expose, including counts aggregated across OpenCypher `UNION` branches and `CALL { … }` subqueries:

Field	Meaning
`nodes_created` / `nodes_deleted`	Vertices created / deleted
`relationships_created` / `relationships_deleted`	Edges created / deleted
`properties_set`	Properties assigned a value
`labels_added` / `labels_removed`	Type/label memberships added / removed
`indexes_added` / `indexes_removed`	Indexes created / dropped
`constraints_added` / `constraints_removed`	Constraints added / removed
`contains_updates`	`true` when any counter above is non-zero

The equivalent HTTP response object is documented at Write statistics.

Connecting from Python

```
import grpc
import arcadedb_server_pb2 as pb
import arcadedb_server_pb2_grpc as rpc

channel = grpc.insecure_channel('localhost:50051')
stub = rpc.ArcadeDbServiceStub(channel)

# Execute a query
request = pb.ExecuteQueryRequest(
    database='mydb',
```

```
    query='SELECT FROM V LIMIT 10',
    language='sql',
    credentials=pb.DatabaseCredentials(username='root', password='arcadedb'),
)
response = stub.ExecuteQuery(request)
for result in response.results:
    for record in result.records:
        print(record.rid, record.type)
```

The `arcadedb-driver-grpc` package ships the generated stubs and attaches the credentials to the channel, so it needs no code generation. See Python — gRPC Driver.

Connecting from Node.js

```
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const packageDef = protoLoader.loadSync('arcadedb-server.proto');
const proto = grpc.loadPackageDefinition(packageDef).arcadedb;

const client = new proto.ArcadeDbService(
  'localhost:50051', grpc.credentials.createInsecure()
);

client.ExecuteQuery({
  credentials: { database: 'mydb', username: 'root', password: 'arcadedb' },
  language: 'sql',
  command: 'SELECT FROM V LIMIT 10',
}, (err, response) => {
  console.log(response.records);
});
```

Further Reading

- Proto definition on GitHub
- HTTP/JSON API — Alternative REST-based access
- PostgreSQL Protocol — Alternative wire protocol

5.4. Bolt Driver Compatibility Matrix

ArcadeDB's Neo4j BOLT protocol implementation is certified against every official Neo4j driver — Java, JavaScript, Python, .NET, and Go — driven by a single, language-neutral conformance spec. Every driver is proven against the full feature matrix, and every unsupported cell is an explicitly documented limitation, never a silent omission.

The full driver × feature × version matrix is **generated from CI** and is the authoritative source. It is published and kept current in the ArcadeDB repository:

- Compatibility matrix (COMPATIBILITY.md) — one cell per scenario × driver × pinned version, with a "last verified" timestamp and a link to the CI run that produced it.
- Conformance spec (spec.yaml) — the shared, language-neutral scenario definitions.
- Driver-version matrix (driver-versions.md) — the pinned driver versions, re-tested nightly against new driver releases.

The matrix on this page summarizes the certified feature areas. Consult the generated matrix above for exact per-version results.

5.4.1. Certified Feature Areas

Area	Scenarios	Status
Connection	`bolt://`, `bolt+s://` (TLS required), `neo4j://` routing (single-node and HA), TLS `OPTIONAL` fallback	Certified
Authentication	Basic auth (valid / invalid credentials); scheme `none` rejected (intentional)	Certified
Transactions	Autocommit; explicit `BEGIN`/`COMMIT`/`ROLLBACK`; managed `executeRead`/`executeWrite` with retry-on-transient; READ access mode enforced (`Neo.ClientError.Statement.AccessMode`) and `tx_timeout` honoured (`Neo.ClientError.Transaction.TransactionTimedOut`) since 26.12.1, see Access mode and transaction timeout	Certified
Causal consistency	Bookmarks (read-after-write across sessions)	Certified
Multi-database	Per-session named-database selection; isolation across databases	Certified
Result handling	Streaming `PULL n`, `DISCARD n` (keeps the stream open while records remain, since 26.12.1; before, it discarded everything), accurate `ResultSummary` counters; `ResultSummary` query type `r` (read only), `w` (write only), `rw` (a write that returns columns) or `s` (index and constraint statements), and for `EXPLAIN` / `PROFILE` the type of the statement they wrap, as Neo4j reports it (since 26.12.1; before, only `r` or `w`)	Certified
Type round-trip	Node, Relationship, Path; temporal (`Date`, `Time`, `LocalTime`, `DateTime`, `LocalDateTime`); `Duration`; spatial `Point`; byte array; lists, sets and typed array properties (`ARRAY_OF_FLOATS` and the other `ARRAY_OF_*` types, as used for vector embeddings) all returned as lists; nested lists/maps; null	Certified

Area	Scenarios	Status
Errors	`Neo.ClientError.` (`syntax`, `semantic`, `security`, `missing entity`, `invalid argument`, `constraint`, `type`) and `Neo.TransientError.` (drives driver retry). A missing record, an unknown type or property, and an invalid value report their own client error codes instead of the generic `Neo.DatabaseError.General.UnknownError`. See Error codes	Certified
Protocol	BOLT 3.0 / 4.0 / 4.4 and 5.0–5.4 negotiation; `RESET` mid-stream	Certified

5.4.2. Error Codes

Every failure carries the Neo4j status code that describes it, so a driver can tell a mistake in your request from a problem with the server, and retry only what is worth retrying.

Selecting a database. These are the first errors a driver can meet, because the database is resolved before anything else runs:

Code	Meaning
`Neo.ClientError.Database.DatabaseNotFound`	The name in the `database` connection parameter does not exist on this server. Permanent: retrying cannot help, fix the name.
`Neo.TransientError.Database.DatabaseUnavailable`	The database exists but cannot serve the request right now - it is closed, or the server has not finished opening its databases. The same request succeeds once it is available, so a retry is worthwhile.
`Neo.ClientError.Security.Forbidden`	The user is not allowed to reach that database. A permissions problem, not a database one.
`Neo.DatabaseError.General.UnknownError`	The database is genuinely broken - corruption, an I/O failure. This is the only case that is the server's fault.

Transactions. `BEGIN`, `COMMIT` and `ROLLBACK` all report the same code for the same underlying failure:

Code	Meaning
`Neo.TransientError.Transaction.DeadlockDetected`	A concurrent modification or lock contention. Drivers retry a managed transaction on this code automatically.
`Neo.ClientError.Transaction.TransactionTimedOut`	A query or transaction deadline ran out. Retrying with a longer timeout is a decision for the caller, so drivers do not retry it on their own.
`Neo.ClientError.Security.Forbidden`	The operation was refused for permission reasons.
`Neo.ClientError.Transaction.TransactionNotFound`	The fallback when the failure matches none of the above.

5.4.3. Documented Limitations

These cells are deliberate, documented differences rather than defects:

- **Auth `none`** — always rejected; ArcadeDB requires credentials.
- **Unauthenticated `RUN`** — the "reject a query before authentication" scenario cannot be exercised through any official driver, because every driver completes the handshake before exposing query methods. It is therefore marked not-applicable rather than tested.
- **Heterogeneous HA BOLT ports** — clusters whose nodes expose BOLT on different ports must declare each node's client-reachable address in `arcadedb.ha.serverList` (see Routing and High Availability).
- **`RID` / `UUID`** — serialized as strings (no native BOLT type).
- **`BigDecimal` / oversized `BigInteger`** — down-converted to `double` (possible precision loss).
- **Auto-commit commit point** - an auto-commit write commits when its `RUN` executes, not at the end of its result stream, so a `RESET` or a dropped connection between `RUN` and `PULL` does not roll it back. This keeps the server-side retry on MVCC conflicts; the official drivers send `PULL` right after `RUN`, and explicit transactions are unaffected. See When an auto-commit write commits.

5.4.4. Certified Drivers and Versions

Each driver is pinned for PR-gating CI and re-tested nightly across a wider version band, so a driver-side release that breaks compatibility is caught quickly. The pinned versions live in the driver-version matrix.

Language	Driver	Certified version bands
Java	`neo4j-java-driver`	4.4.x (legacy), 5.x, 6.x
JavaScript	`neo4j-driver`	5.x, 6.x
Python	`neo4j`	5.x (LTS), 6.x
.NET	`Neo4j.Driver`	5.x, 6.x
Go	`neo4j-go-driver/v5`	5.x

For the advertised server identity these results are measured against, see Server Identity and Feature Envelope.

5.4.5. See Also

- Neo4j BOLT Protocol
- OpenCypher Query Language
- Generated compatibility matrix

5.5. Postgres Protocol Plugin

ArcadeDB Server supports a subset of the Postgres wire protocol, such as connection and queries.

If you're using ArcadeDB as embedded, please add the dependency to the `arcadedb-postgresw` library. If you're using Maven include this dependency in your `pom.xml` file.

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-postgresw</artifactId>
    <version>26.11.1</version>
</dependency>
```

To start the Postgres plugin, enlist it in the `server.plugins` settings. To specify multiple plugins, use the comma `,` as separator. Example:

```
~/arcadedb $ bin/server.sh -Darcadedb.server.plugins
="Postgres:com.arcadedb.postgres.PostgresProtocolPlugin"
```

If you're using MS Windows OS, replace `server.sh` with `server.bat`.

In case of an incompatibility, restart the server with the additional option `-Darcadedb.postgres.debug=true`, repeat the connection attempt, and add the debug output to the issue report.

In case you're running ArcadeDB with Docker, use `-e` to pass settings and open the Postgres default port `5432`:

```
docker run --rm -p 2480:2480 -p 5432:5432 \
       --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata \

-Darcadedb.server.plugins=Postgres:com.arcadedb.postgres.PostgresProtocolPlugin " \
          arcadedata/arcadedb:latest
```

The Server output will contain this line:

```
2026-10-29 05:19:27.030 INFO  [PostgresNetworkListener] <ArcadeDB_0> Listening for
incoming connections on /0.0.0.0:5432 (protocol v.-1, TLS: DISABLED)
2026-10-29 05:19:27.031 INFO  [PluginManager] <ArcadeDB_0> - PostgresProtocolPlugin
plugin started
```

Once you have enabled the Postgres Protocol, you can interact with ArcadeDB server by using any Postgres drivers. The driver sends the queries to the ArcadeDB server without parsing or checking the syntax. For this reason, even if ArcadeDB SQL is different from Postgres SQL, you're still able to execute any ArcadeDB SQL command through the Postgres driver. Check out the following list with the official drivers for the most popular programming languages:

- C
- C#

- C++
- Go
- Javascript - Node.js
- PHP
- Python
- R
- Ruby
- Rust

For the complete list, please check Postgres website.

5.5.1. Other query languages

By default the Postgres driver interprets all the commands as SQL. To use another supported language, like Cypher, Gremlin, GraphQL or MongoDB, prefix the command with the language to use between curly brackets.

Example to execute a query by using GraphQL:

```
{graphql}{ bookById(id: "book-1"){ id name authors { firstName, lastName } }
```

Example to use Cypher:

```
{cypher}MATCH (m:Movie)<-[a:ACTED_IN]-(p:Person) WHERE id(m) = '#1:0' RETURN *
```

Example of using Gremlin:

```
{gremlin}g.V()
```

The marker must be the first token of the statement: only whitespace and comments (`-- …` or `/* … */`) may precede it. A `{language}` written anywhere else, for example inside a string literal, is part of the SQL statement and does not change its language.

The languages a marker may select are listed in the server setting `arcadedb.postgres.languages` (default `sqlscript,cypher,opencypher,gremlin,graphql,mongo`). SQL is always accepted. A marker naming another language, or one no query engine of the server provides, is refused with SQLSTATE `0A000` (`feature_not_supported`). Set the setting to an empty value to accept SQL only:

```
bin/server.sh -Darcadedb.postgres.languages=
```

5.5.2. TLS/SSL

The plugin supports TLS through the Postgres protocol's own handshake, so `psql`, JDBC and BI tools can connect with `sslmode=require`. Set `arcadedb.postgres.ssl` to one of:

- `DISABLED` (default): no TLS. A client asking for TLS is told to continue in plain text.
- `OPTIONAL`: the client chooses TLS or plain text. A client that does not ask for TLS sends its password in the clear.
- `REQUIRED`: plain text connections are refused before any password is requested. Recommended once certificates are in place.

TLS uses the same key store and trust store settings as the HTTP server (`arcadedb.ssl.keyStore`, `arcadedb.ssl.keyStorePassword`, `arcadedb.ssl.trustStore`, `arcadedb.ssl.trustStorePassword`), and all four must be set. The server does not ask clients for a certificate.

```
bin/server.sh -Darcadedb.server.plugins
="Postgres:com.arcadedb.postgres.PostgresProtocolPlugin" \
  -Darcadedb.postgres.ssl=REQUIRED \
  -Darcadedb.ssl.keyStore=/path/keystore.pkcs12 -Darcadedb.ssl.keyStorePassword=secret
\
  -Darcadedb.ssl.trustStore=/path/truststore.jks -Darcadedb.ssl.trustStorePassword
=secret
psql "host=localhost port=5432 dbname=mydb user=root sslmode=require"
```

PostgreSQL 17 "direct" TLS (`sslnegotiation=direct`) is not supported.

(Since v26.10.2.) The user a connection authenticated as is checked again before every message: a user deleted, given a new password or stripped of the grant on the connected database is disconnected with a `FATAL` error, SQLSTATE `28P01`. See User changes on open connections.

5.5.3. Current limitations

The documentation about Postgres wire protocol is not exhaustive to build a bullet proof protocol. In particular the state machine. For this reason this plugin was created by reading the available documentation online (official and not official) and looking into Postgres drivers or implementations.

Both the "simple" query protocol and the extended query protocol (`Parse`/`Bind` /`Describe`/`Execute`/ `Sync`, what `PreparedStatement` uses) are supported.

`BEGIN`, `COMMIT` and `ROLLBACK` sent through the extended query protocol are acknowledged the way PostgreSQL acknowledges them. Previously a libpq-based client - psycopg3, asyncpg, the Apache Arrow ADBC PostgreSQL driver - could fail to open a transaction with `Failed to begin transaction`. In the same release, a statement the server rejects no longer leaves the previously prepared statement usable under the same name, so a client can no longer receive the earlier statement's rows in answer to one that was refused.

A portal (the bound statement `Execute` runs) lives only as long as the transaction it was created in, as in PostgreSQL: in autocommit it ends at `Sync`, inside `BEGIN` it ends at `COMMIT`/`ROLLBACK`. Fetching a large result in chunks (JDBC `setFetchSize()`) therefore needs a transaction, which is what the PostgreSQL JDBC driver already requires. Binding from a prepared statement that does not exist is refused with SQLSTATE `26000`, and executing or describing a portal that does not exist with `34000`, instead of answering with a success message.

Since v26.11.1, a `Bind` must supply exactly as many parameter values as its prepared statement requires: one for every `$N` up to the highest the statement text references, or for every type the `Parse` declared if that is more. A `Bind` with fewer or more values is refused with SQLSTATE `08P01` (`bind message supplies 0 parameters, but prepared statement "s1" requires 1`), and so is a `Bind` carrying several parameter format codes that do not match its value count, as in PostgreSQL. Before, a missing value left its `$N` to be read as a property of that name, so an `UPDATE ⋯ SET x = $1` bound with no value wrote `NULL` and reported success. A `$N` inside a string literal, a quoted name or a comment is not counted.

In the same release a named prepared statement and a named portal can no longer be redefined while they exist: a `Parse` naming a live statement is refused with `42P05` and a `Bind` naming a live portal with `42P03`, and the existing one is left as it was. Close it first (`Close`, `DEALLOCATE`, `CLOSE`). The unnamed statement and the unnamed portal are still replaced by every `Parse`/`Bind` that names them, which is what drivers rely on. Closing a prepared statement leaves the portals already bound from it executable, as PostgreSQL does.

Since v26.11.1, a `SELECT` whose columns are known before its first row is read is fetched from a live cursor: each `Execute` (each chunk of a JDBC `setFetchSize()` fetch) reads only the rows it asks for, instead of the server reading the whole result before it sends the first row, and the query is not subject to `arcadedb.postgres.queryMaxRows`. The columns are known when the query projects only properties its type declares (`SELECT name, age FROM Person`), of a scalar type, `BINARY`, `MAP` or a `LIST` of a scalar type, or when its prepared statement was already described to the client (`Describe` on the statement, or a statement the PostgreSQL JDBC driver re-executes after `prepareThreshold`). Any other query - `SELECT FROM Person`, `SELECT *`, an expression, a property the type does not declare, another query language - can only be described from its rows, so it is still read whole first, within that limit. To fetch a large type in chunks, list the declared properties you need.

A live cursor is a running statement until it is drained or its portal ends: it is listed by `list queries` and `SHOW TRANSACTIONS`, and a cancel request or `terminate query` stops it, also while the client is between two fetches. Its `arcadedb.command.timeout` (or the session's `statement_timeout`) runs from the moment the query starts to its last fetch, including the time the client takes between two fetches, where PostgreSQL restarts `statement_timeout` for each `Execute`. An idle cursor does not hold a slot of `arcadedb.queryMaxConcurrent`: each fetch is admitted on its own. A write the same transaction makes between two fetches can be seen by the rows not fetched yet, unless the query sorts (`ORDER BY` reads every row on the first fetch).

Running a prepared statement without asking for its row description again (what the PostgreSQL JDBC driver does once a statement has run `prepareThreshold` times, 5 by default) no longer sends an unrequested row description. Before, the JDBC connection failed with `NoSuchElementException` from the 6th execution on.

When a prepared statement is described (`Describe` on the statement, what asyncpg, npgsql and pgx

do) and the server answers that it returns no rows, running it keeps that promise. An `INSERT`, `UPDATE`, `DELETE`, `CREATE VERTEX` or `CREATE EDGE` without `RETURN` answers only its command tag with the number of affected records (e.g. `INSERT 0 1`), like PostgreSQL. A statement whose columns cannot be known before it runs (e.g. a `{cypher}` query) and that then returns rows is refused with SQLSTATE `0A000`: describe the portal instead (what the PostgreSQL JDBC driver, libpq and psycopg do), which runs the statement first and reports its real columns.

5.5.4. System catalog and schema introspection

Queries against `pg_catalog` and `information_schema` - the ones a client's driver sends to discover schemas, tables, columns and types, typically via `DatabaseMetaData.getSchemas()`/`getTables()` /`getColumns()`/`getTypeInfo()` - are answered by recognizing the **shape** of the query (which relations it names, which columns it projects, including a client's own `CASE` expressions) rather than by matching one tool's exact spelling. This means the same question is answered the same way whichever Postgres client sends it, rather than only for a short list of tools this plugin happened to be tested against. A catalog query in a shape the server does not recognize, or over a catalog relation it does not model (`pg_proc`, `pg_trigger`, `pg_extension`, `information_schema.routines` and so on), fails with SQLSTATE `0A000` (`feature_not_supported`) whose message names the relation, *since v26.11.1*. Before, it returned an empty result set, which a client reads as an empty catalog: an ungrouped `SELECT count(*) FROM pg_proc WHERE …` came back with no row at all where PostgreSQL always sends one, and `psql␣s '\di`, psqlodbc's `SQLPrimaryKeys` and `SQLStatistics` reported nothing. A refusal a client can see is recoverable; a wrong empty answer is not. The connection stays usable after the error. One kind of statement keeps the empty result, on purpose: a statement that names `pg_catalog` but reads no catalog relation, such as `SELECT pg_catalog.set_config(…)`, the session probes `pg_dump` sends.

Since v26.11.1, the *type lookups* a driver sends to name a type it has not met are answered like any other catalog query, from the types this protocol can produce, with the values the driver bound: the OID of a type name, the array type of an element type, the element of an array type, the name of an OID and whether its schema is on the search path. A name or an OID this server does not have is an empty answer, which is what those drivers are built to handle; one it has is no longer reported as missing. The evaluator behind them reads the constructs those lookups are written with, so a differently spelled lookup, or another client's, is answered too: a table function in `FROM` (`generate_series(start, stop [, step])` and `unnest(array)`), `USING (column)`, `= ANY(array)` and `<> ALL(array)`, a derived table joined to other relations, and one relation read through two aliases (`pg_type t JOIN pg_type arr ON arr.oid = t.typarray`). A series is charged to the work budget of the statement before it is built, so a lookup cannot ask for a billion rows. `pg_range` and `pg_enum` are modelled, and have no row.

Every type is a table in one schema, named after the connected database - matching `SELECT current_schema()` - because a PostgreSQL connection is bound to one database and sees the schemas inside it, and in ArcadeDB a connection is likewise bound to one database whose types are its tables. *Since v26.11.1*, `pg_namespace` (and so `DatabaseMetaData.getSchemas()`) also lists `pg_catalog`, the schema every built-in type belongs to, as PostgreSQL does: a schema browser may show it, empty.

Since v26.11.1, a materialized view and a continuous aggregate have columns in the catalog: the ones their query returns, in that order and under their aliases, so `\d <view>` prints them.

Since v26.11.1, each materialized view and continuous aggregate also has its `_RETURN` rule in

`pg_rewrite`, as in PostgreSQL, and `pg_class.relhasrules` is true for it. `pg_depend` links that rule to its view and to every type the view reads (including types read only in a subquery), so a tool asking what is built on a table, for example before dropping it, finds the views that depend on it. Dependencies are listed per type, not per column. `pg_get_ruledef(oid)` prints the rule.

`pg_type` can also be queried on its own, and not only as part of a column list. Some clients build a map of every type the server has before they run a single query, and refuse to connect if that map comes back empty - the Apache Arrow ADBC PostgreSQL driver is one of them. Such a query is answered with the types this protocol can actually produce (see Column type mapping below), each described the way PostgreSQL describes it. A filter the server cannot interpret returns no rows rather than every type, so a client asking whether an extension type such as `hstore` exists is correctly told it does not.

The `regclass` and `regtype` casts resolve, in both directions: `'mytable'::regclass` is that type's OID and `attrelid::regclass` is its name, and likewise `'int4'::regtype` and `atttypid::regtype` for types. The function spellings `to_regclass()` and `to_regtype()` do the same. Some drivers ask for a table's columns by writing the name as a `regclass` cast and comparing it against `pg_class.oid` - the Apache Arrow ADBC PostgreSQL driver does this in `GetTableSchema` - and before this the cast was ignored, so the comparison silently matched nothing and the client saw a table with no columns. A name no type carries resolves to `NULL`, as `to_regclass()` does in PostgreSQL, so asking about a table that does not exist returns nothing rather than describing every table.

Type names are matched case-sensitively first, as ArcadeDB stores them, and only then ignoring case - so `'MyType'::regclass` finds the type spelled that way even though PostgreSQL would have folded the unquoted name to lower case. If two types differ only in case, an unquoted name that matches neither of them exactly resolves to nothing rather than picking one; quote the name to say which you mean.

A query that joins a catalog relation to `pg_roles` or `pg_user` to name an owner - what a GUI browser sends to list tables with their owner - answers with the connected user rather than `NULL`:

```
SELECT c.relname, r.rolname AS owner FROM pg_class c JOIN pg_roles r ON r.oid = c
.relowner
```

Only the connected user is described as a role; an emulated catalog does not hand out the server's account list. A query about roles, databases, privileges, character sets or collations that also mentions `pg_namespace` is now answered as the question it asks, instead of being answered as a schema query.

Since v26.11.1, `psql␣s` `\dt`, `\d <table>` and `\d+ <table>` list and describe the database's types, and the PostgreSQL ODBC driver (psqlodbc) answers `SQLColumns` and `SQLTables(SQL_ALL_TABLE_TYPES)`. Before, `\dt` printed `Did not find any relations`, `SQLColumns` returned no columns and `SQLTables(SQL_ALL_TABLE_TYPES)` failed with a syntax error. The catalog queries these clients send are now read as PostgreSQL reads them:

- a parenthesised join in `FROM` - `FROM (pg_class c JOIN pg_namespace n ON ⋯) JOIN pg_attribute a ON ⋯` - and a join to `pg_am` (every table reports the `heap` access method);
- the condition of an `INNER JOIN` filters rows exactly like the `WHERE` clause, and a `LEFT JOIN` whose

condition does not hold reads `NULL` from the joined relation instead of removing the row;

- `OPERATOR(pg_catalog.=)` and the other `OPERATOR(…)` spellings, and `COLLATE` (ignored: catalog names sort by code point, as with `COLLATE "C"`, whatever collation is named);
- `UNION`, `INTERSECT` and `EXCEPT` (with or without `ALL`) whose arms are catalog queries or `SELECT` statements over literals, such as `SELECT 1 UNION SELECT 2`. ArcadeDB SQL has no set operations, so a set operation that reads your own types is still passed to the SQL engine, which rejects it;
- `count(*)` and the other aggregates (`count`, `sum`, `avg`, `min`, `max`, `bool_and`, `bool_or`, `string_agg`, `array_agg`), with or without `GROUP BY`;
- correlated sub-selects: `(SELECT …)` as a value, `EXISTS (SELECT …)`, `ARRAY(SELECT …)` and `IN (SELECT …)`;
- a column default appears in `pg_attrdef` and is printed by `pg_get_expr()`, so `\d` and a JDBC client's `COLUMN_DEF` show it, as `information_schema.columns` already did.

A projection that names two columns alike, such as `SELECT NULL, NULL, relkind`, returns both columns.

A catalog query that would take more work than the server spends on one statement - a regular expression that backtracks heavily, or deeply nested correlated sub-selects - fails with SQLSTATE `54000` (`program_limit_exceeded`) instead of returning an empty result.

`format_type()` and the `regtype` cast now return the name PostgreSQL prints for a type (`integer`, `character varying`, `double precision`) instead of its internal name (`int4`, `varchar`, `float8`), matching PostgreSQL. `pg_type.typname` is unchanged.

Since v26.11.1, `information_schema.columns.data_type` answers the SQL standard name, as `format_type()` does (`integer`, `character varying`, `double precision`, `timestamp without time zone`), and the literal `ARRAY` for any array column. `udt_name` keeps the `pg_type` name (`int4`, `varchar`, `float8`, `_text`), so the two columns differ exactly where PostgreSQL's do. The precision columns the relation declares are written for the built-in types that have them: `numeric_precision` (16, 32 and 64 for the integers, 24 for `real`, 53 for `double precision`), `numeric_scale` (0 for the integers), `datetime_precision` (6 for a timestamp, 0 for a date) and `character_octet_length` (2^30 for a text column with no declared length). `numeric_precision_radix` is `NULL` for a type that is not a number, where it used to answer 2.

Since v26.11.1, a catalog query that matches no row describes its columns exactly as one that matches: every column of a catalog relation has a fixed type (`pg_class.oid` is always `int4`, `relhasindex` always `bool`). A `pg_type` lookup of a type that does not exist returns the columns it asked for with no rows. *(Before, such a lookup returned no columns at all, and an empty answer from another relation announced every column as* `varchar`*, issue #10869.)*

Since v26.11.1, the indexes of the database are in the catalog, so `psql␣s '\d <table>` prints the `Indexes:` section, `\d <index>` describes an index, `\di` lists them, and the index queries of psqlodbc and of the GUI tools are answered:

- `pg_index` has a row per index of a type, `pg_class` a row of kind `i` for it, `pg_attribute` a row per key column, `pg_indexes` the statement that creates it (also printed by `pg_get_indexdef()`), and `pg_class.relhasindex` and `pg_tables.hasindexes` are `true` for the types that have one. An index is

listed under the name ArcadeDB gives it (`Person[name]`) on the type that declares it, not on its sub-types, and its access method is the family it belongs to in PostgreSQL's terms: `btree` for an LSM tree, `hash`, `gin` for a full-text index, `gist` for a geospatial one and `hnsw` for the vector indexes.

- ArcadeDB declares uniqueness with a unique index and has nothing else that could be the constraint, so each unique index is also the unique constraint of the same name: it is in `pg_constraint` (kind `u`), in `information_schema.table_constraints` and `information_schema.key_column_usage`, and `psql` prints it as `UNIQUE CONSTRAINT`. There is no primary key, foreign key or check constraint in ArcadeDB, so none is reported and `SQLPrimaryKeys` returns no row.
- `pg_inherits` has a row per super-type of a type, which is what `\d` prints as `Inherits:` and `Child tables:`. `pg_policy`, `pg_statistic_ext`, `pg_publication`, `pg_publication_rel` and `pg_publication_namespace` have no row: ArcadeDB has no row-level security policy, extended statistics object or publication. `psql␣s '\d` asks about all of them for every table, and it used to fail on the first one it was refused.
- `indkey`, `indcollation`, `indclass` and `indoption` are read as text, the values separated by a space, and the first can be subscripted from 0 (`indkey[0]`), as PostgreSQL's `int2vector` is. The collations and operator classes have no equivalent here and are 0.
- A statement may read one relation more than once - `pg_class` as the table and as each of its indexes, `pg_attribute` as a column of the table and of the index - and an outer join on these relations keeps the rows nothing matches, reading `NULL`. The conditions are evaluated as soon as the relations they read are bound, and a condition equating a column with something already bound finds the rows by it, so a table's indexes are found without reading every index of the database. A catalog statement that reads one of the other relations twice is still answered the way it was.
- A comma join of catalog relations with no condition between them (`FROM pg_class c, pg_namespace n`) answers one row per entity, not PostgreSQL's cross product, when the statement reads a column of one of them. Clients join the catalogs with a condition, so this only shows in a hand-written query. A statement that reads no column (`SELECT count(*) FROM pg_class, pg_namespace`) does multiply, and a product too large for the work budget is refused with error `54000`.

The PostgreSQL JDBC driver's `DatabaseMetaData.getIndexInfo()` and `getPrimaryKeys()` - what Hibernate, Flyway, Liquibase and BI tools call - are answered too. They list an index's key columns with the set-returning function `information_schema._pg_expandarray()` (a row for each element of the array, the calls of one row in step) and read its members as `(expression).n`; the catalog reads those, SQL's `trim(both '"' from …)` and the bitwise `&` the driver uses to find a descending key. `getPrimaryKeys()` returns no row, as ArcadeDB has no primary key. A set-returning call under `GROUP BY` or an aggregate is not read.

A text compared with a flag is read as the boolean it spells (`indisprimary = 't'`).

An index key over a `HIDDEN` property is not a column of its table in the catalog, so it is described the way PostgreSQL describes an index on an expression: its column is named `expr`, and a unique index with such a key is not listed as a unique constraint.

5.5.5. COPY … TO STDOUT

`COPY` in its export direction is supported, in every form PostgreSQL accepts for it:

```
COPY (SELECT name, age FROM Person WHERE age > 18) TO STDOUT
COPY Person TO STDOUT (FORMAT csv, HEADER)
COPY Person (name, age) TO STDOUT WITH (FORMAT binary)
COPY (SELECT FROM Person) TO STDOUT CSV HEADER DELIMITER ';' NULL 'nil'
```

The query in parentheses runs as it stands, in the session's query language (so a `{cypher}` prefix applies to it), and a type name with an optional column list stands for the equivalent `SELECT`. Quote the name if it is not a plain identifier - `COPY "My Type" TO STDOUT` - and any character is allowed in it, backslashes and backticks included. The rows travel in PostgreSQL's own `text`, `csv` or `binary` COPY encoding, so anything that reads a PostgreSQL COPY stream reads them: `psql␣s `\copy (\copy (SELECT FROM Person) TO 'people.csv' CSV HEADER)`, pgjdbc's `CopyManager`, pandas, and the Apache Arrow ADBC PostgreSQL driver, which reads every result set through `COPY … TO STDOUT (FORMAT binary)` by default and no longer needs `adbc.postgresql.use_copy=false`.

The option list (`FORMAT`, `DELIMITER`, `NULL`, `HEADER`, `QUOTE`, `ESCAPE`, `FORCE_QUOTE`, `ENCODING`) and the older keyword spelling (`BINARY`, `CSV HEADER`, `DELIMITER AS`, `NULL AS`, `QUOTE AS`, `ESCAPE AS`, `FORCE QUOTE`) are both accepted, with the same restrictions PostgreSQL applies to their combinations. The output is always UTF-8.

The exported columns are fixed before the first row goes out, the way `RowDescription` fixes them for a query. For a whole type they are every property the type declares, inherited ones included, plus whatever a sample row adds on top - a declared property is exported even when the first row does not have one. Because ArcadeDB types can hold rows of different shapes, that set is a good answer but not a provable one: if a row turns out to carry a property the export never announced, the `COPY` fails with SQLSTATE `0A000` naming that column, rather than writing the row without it. List the columns yourself to make the set exact and the export complete:

```
COPY Person (name, age, email) TO STDOUT
COPY (SELECT name, age, email FROM Person) TO STDOUT
```

A `COPY` whose columns the schema can name is streamed row by row and is not subject to `arcadedb.postgres.queryMaxRows`, which is what makes it the way to export a large type over this protocol; a query whose columns only its rows can name (a schemaless type with no sample row, another query language) is materialized first, within that limit, exactly as a plain query is. The streamed form holds the query's cursor open for as long as the client reads, which is what a bulk export needs; where that is not wanted for every user of the wire, bound the query itself with a `WHERE` or a `LIMIT`, as `psql \copy` users do.

Not supported, and refused with SQLSTATE `0A000` (`feature_not_supported`): the ingest direction `COPY … FROM STDIN` (use `INSERT`, or the HTTP import API), and the server-side targets `COPY … TO 'file'` and `COPY … TO PROGRAM`, which would let a wire client write files or run commands on the server host. In binary format a column without a binary encoding on this server (an array) is refused the same way; export it in text or CSV format, or project it as a string.

5.5.6. Column type mapping

A column's advertised PostgreSQL type must not depend on whether the query happens to return any rows - a client that prepares a statement against an empty result (or a schema-discovery `DESCRIBE`) and later re-executes it against a populated one would otherwise see the column's type change under it. This holds for every type ArcadeDB maps onto a native PostgreSQL OID:

- `BINARY` is announced as `bytea` (with the standard `\x<hex>` text encoding), so `ResultSet.getBytes()` /`PreparedStatement.setBytes()` round-trip it as raw bytes, rather than lossy `varchar`/`"char"[]` answers. *Since v26.11.1*, a `LIST OF BINARY` is announced as `bytea[]` (`_bytea`) whatever rows a query returns, and each element is sent as its bytes, so `ResultSet.getArray()` returns a `byte[][]`. Before, it was announced as `text[]`, `json` or `varchar` depending on the rows, and its elements were sent as JSON arrays of numbers.
- `DECIMAL` is announced as `numeric` (OID 1700), with a real binary encoder/decoder, rather than the lossy `double precision` or the text-only `varchar`: `ResultSet.getBigDecimal()` round-trips the full precision. A `NUMERIC` value is capped at 16,000 total decimal digits (well under PostgreSQL's own limit of 131,072 integer / 16,383 fractional digits), because ArcadeDB's `DECIMAL` type has no configured precision or scale limit of its own; a value beyond this cap is declined with an error rather than accepted and silently truncated.
- `SHORT` and `BYTE` are announced as `int2` (`smallint`), matching the declared schema type, rather than widening to `int4` (`integer`).
- `DATE` is announced as `date`.
- `DATETIME` is announced as `timestamp`. The one case where a `DATETIME` column's announced type can still depend on whether a row was sampled is a database explicitly configured with `arcadedb.dateTimeImplementation=java.util.Date` (the default is `java.time.LocalDateTime`): `java.util.Date` is also `DATE`'s default representation, so a bare sampled value cannot always tell the two apart on its own. The server resolves this from the schema whenever the query names a single source type, which covers ordinary column selection; a column reached through a table alias, a computed expression, or a multi-table `JOIN` can still fall back to the value-only answer in that one configuration.
- *Since v26.12.1*, a `{cypher}` query announces its temporals as SQL does: a stored `DATE` read as `n.prop` or a `date()` result is `date`, a stored `DATETIME` or a `localdatetime()` result is `timestamp`, and a list of them is written as SQL writes the same list. A zoned `datetime()`, a `time()` and a `localtime()` stay `varchar`, as their SQL counterparts do. Before, every Cypher temporal was a `varchar` holding its ISO text (issue #11955).

A column is typed from all the rows of a result, not from the first one, and a list from all its elements: `[1, 2.5]` is announced as `float8[]`, `[1, 3000000000]` as `int8[]`, and a list that also holds a `DECIMAL` as `numeric[]`. A list or column that mixes values no common number type can hold is announced as text.

A prepared statement keeps ONE layout for all its rows, because a client may describe it once and reuse that description. A property the type does not declare (ArcadeDB is schemaless) can hold an integer in one record and a long, a double or a string in the next, so such a property is announced as `varchar` (a list as `text[]`) when the statement is described (`Describe` on the statement, or on a portal of a named statement, which is what the PostgreSQL JDBC driver does from its `prepareThreshold`-th execution). Declared properties keep their exact native type. To get a native

type back for a property in a prepared statement, declare it, for example `CREATE PROPERTY Event.score INTEGER`.

Since v26.11.1, `pg_type` also lists the array types of `int2`, `bytea`, `bpchar`, `varchar`, `date` and `timestamp`, and a parameter bound as one of them is accepted - for example the `varchar[]` the PostgreSQL JDBC driver sends for `Connection.createArrayOf("varchar", …)`, which used to fail.

Since v26.11.1, an array parameter sent in text format (a `{…}` literal, what a client that does not negotiate the binary format sends) is read as PostgreSQL reads it. An unquoted `NULL` element is a SQL NULL, while `"NULL"` is the string `NULL`. An empty element `""` is kept. A malformed literal, such as `{a,,b}`, is refused with `malformed array literal` instead of being stored as something else. A parameter sent without a type is read as an array only when it is a valid array literal, so a JSON document such as `{"k": 1}` stays a string. *(Before, an empty element was dropped and* `NULL` *was stored as the string* `NULL`*, issue #10870.)*

Since v26.11.1, every type of the database (a table or a materialized view in the catalog) also has its row type in `pg_type`, as every PostgreSQL table has: a composite type (`typtype = 'c'`, `typcategory = 'C'`) named after the type, in its schema, whose `typrelid` is the relation's `pg_class.oid`. `pg_class.reltype` holds its OID, so the join `pg_class c JOIN pg_type t ON t.oid = c.reltype` finds it, and `format_type(c.reltype, NULL)` and `c.reltype::regtype` print the relation's name. Only an index has `reltype = 0`. A column's `pg_attribute.attcollation`, and each key of `pg_index.indcollation`, is the collation of its type: `100` (`default`) for `text`, `varchar`, `bpchar` and their arrays, `0` for every other type, as `pg_type.typcollation` says. *(Before,* `reltype` *and* `attcollation` *were 0 for every relation and column, so the join from a table to its row type answered no rows, issue #10873.)* A client that enumerates `pg_type` now reads one more row per type of the database.

5.5.7. Transactions

In autocommit mode (the JDBC default) every statement is one transaction: a write statement that touches many records - `UPDATE`, `DELETE`, `INSERT`, `CREATE VERTEX`, `CREATE EDGE` - either lands whole or is rolled back whole, for vertex, edge and document types alike. The one exception is a statement that asks for chunked commits with `BATCH <n>`, which commits every `<n>` records: use it on a statement touching millions of records, since one transaction otherwise holds every modified page until the statement ends. To span several statements with one transaction, send an explicit `BEGIN` … `COMMIT` (or `ROLLBACK`).

A client that PIPELINES several statements before waiting for their answers - a JDBC `executeBatch()`, psycopg3 pipeline mode, asyncpg - groups them differently: everything between two `Sync` messages is one transaction, so a batch either lands whole or leaves nothing behind, and the statements after the one that failed are not run at all. This is what PostgreSQL does and what `BatchUpdateException` handling assumes.

Since v26.12.1, a simple query holding several statements separated by `;` runs them all, as PostgreSQL does. This is what psycopg2's `cursor.execute("…; …")`, `psql -c` and the connect-time setup of psqlodbc 16 (the `odbc-postgresql` package of Debian 12/13 and Ubuntu 24.04: `SET DateStyle = 'ISO';SET extra_float_digits = 2;show transaction_isolation`) send:

- the statements run in order and each one is answered on its own (its rows and its command tag), followed by one `ReadyForQuery` for the whole message;

- outside an explicit `BEGIN` they form one transaction: they are committed together after the last one, and the first error stops the message and rolls back every statement before it, `SET` included. The statements after the error do not run;
- a `BEGIN` among them turns that transaction into an explicit one, the statements before it included, and a `COMMIT` or `ROLLBACK` among them ends it (the next write opens a new one). Inside an explicit block an error aborts the block, as a single statement would;
- `;` splits only outside string literals, quoted names, dollar-quoted literals, comments and ArcadeDB JSON objects or lists (`CONTENT {…}`, `[…]`), and an empty statement (a trailing or doubled `;`, or only a comment) is ignored; a message with no statement at all is the empty query;
- a message that starts with a language marker other than `{sql}` is handed whole to that language, so a `{sqlscript}` script keeps its own `;` separators;
- the extended query protocol takes one statement per `Parse`, as in PostgreSQL: a `Parse` of several statements is refused with `42601` (`cannot insert multiple commands into a prepared statement`).

In v26.11.1 and earlier such a message was run as a single statement and refused as a syntax error (`42601`), which kept psqlodbc 16 from connecting.

A transaction-control statement is recognized in every spelling ArcadeDB SQL accepts, not only the bare keyword: with a leading or trailing comment (as added by sqlcommenter, OpenTelemetry or connection-pool routing hints, e.g. `COMMIT /*traceparent='…'*/`), with extra `;`, `BEGIN ISOLATION <level>` (which opens the transaction with that isolation level, e.g. `BEGIN ISOLATION REPEATABLE_READ`) and `COMMIT RETRY <n>` (a plain `COMMIT` outside a SQL script, which has no block to retry). A `COMMIT` that fails, for example on a duplicated unique key, reports the error and ends the transaction, as in PostgreSQL: the session is back in idle state and no `ROLLBACK` is needed.

Since v26.10.2 PostgreSQL's own transaction-control spellings are accepted too, as ORMs and drivers send them:

- `BEGIN [WORK | TRANSACTION] [<mode> [, …]]` and `START TRANSACTION [<mode> [, …]]`, where `<mode>` is `ISOLATION LEVEL <level>`, `READ WRITE` or `[NOT] DEFERRABLE`;
- `{COMMIT | END} [WORK | TRANSACTION] [AND NO CHAIN]`;
- `{ROLLBACK | ABORT} [WORK | TRANSACTION] [AND NO CHAIN]`.

`ISOLATION LEVEL READ COMMITTED` and `READ UNCOMMITTED` open a `READ_COMMITTED` transaction (in PostgreSQL too `READ UNCOMMITTED` behaves as `READ COMMITTED`), and `REPEATABLE READ` a `REPEATABLE_READ` one. What this server cannot honor is refused with SQLSTATE `0A000` (feature not supported) instead of a syntax error: `ISOLATION LEVEL SERIALIZABLE`, `READ ONLY`, and `COMMIT AND CHAIN` / `ROLLBACK AND CHAIN`.

Savepoints are not supported. `SAVEPOINT <name>` and `RELEASE [SAVEPOINT] <name>` are accepted and do nothing, but `ROLLBACK TO [SAVEPOINT] <name>` is refused with SQLSTATE `0A000` (feature not supported) and the transaction is marked aborted, so the client must `ROLLBACK` it. An ORM or driver that wraps statements in savepoints (for example a nested `transaction.atomic()` in Django, or psql's `ON_ERROR_ROLLBACK`) only works as long as it never rolls back to one.

Setting auto commit to false is not 100% supported: it does not start a transaction on its own, so a

write sent after `setAutoCommit(false)` without a `BEGIN` is refused with "Transaction not begun". With JDBC, leave the default settings or set:

```
conn.setAutoCommit(true);
```

5.5.8. Session settings (SET, SHOW, RESET)

`SET` changes a setting for the current connection only: it never changes the database or any other connection, and `SHOW` reads it back. It follows PostgreSQL's rules:

- a `SET` made inside a transaction that is rolled back, or that fails, is undone;
- `SET LOCAL` lasts only until the end of the current transaction (outside a transaction it has no lasting effect);
- `RESET <name>`, `RESET ALL` and `SET <name> TO DEFAULT` restore the value the client sent when it connected, else the server default;
- `SET TIME ZONE '<zone>'` is accepted as a synonym of `SET TimeZone TO '<zone>'`, and `SHOW TIME ZONE` / `RESET TIME ZONE` as synonyms of `SHOW TimeZone` / `RESET TimeZone`;
- `SHOW TRANSACTION ISOLATION LEVEL` is a synonym of `SHOW transaction_isolation`: it answers the level of the transaction that is open (for example `repeatable read` after `BEGIN ISOLATION REPEATABLE_READ`), under the column `transaction_isolation`, which is also what the JDBC driver's `Connection.getTransactionIsolation()` reads;
- `SHOW` answers under PostgreSQL's spelling of the parameter (`DateStyle`, `TimeZone`), and `SHOW ALL` lists the parameters the server answers for plus every one the connection has set;
- a `SET` answers only `SET`, like in PostgreSQL, instead of a one-row result;
- the parameters PostgreSQL reports to drivers (`DateStyle`, `TimeZone`, `application_name`, `client_encoding`, `IntervalStyle`, `is_superuser`, `server_version`, ...) are sent at connection time and again whenever they change, so a driver that caches them sees the new value.

Some settings describe what the server does and cannot be changed: dates always travel in ISO format (`DateStyle` keeps only the field order you choose) and text is always `UTF8`. A `SET` of a read-only setting such as `server_version` is refused.

String literals are read as PostgreSQL reads them. With `standard_conforming_strings on`, the default and what the server reports to drivers, a backslash in a plain `'…'` literal is an ordinary character (`'C:\'` is the three characters `C:\`) and a quote inside a literal is doubled (`'it''s'`). Backslash escapes are processed in an `E'…'` escape string (`E'a\tb'` holds a tab), with PostgreSQL's escape set: `\b \f \n \r \t`, octal `\ooo`, hex `\xhh`, `\uXXXX` and `\UXXXXXXXX`. `SET standard_conforming_strings = off` (or the same option in the startup packet, e.g. pgjdbc's `options=-c standard_conforming_strings=off`) makes every plain literal an escape string too, as in PostgreSQL. Inside an ArcadeDB JSON object (`CONTENT {…}`) a single-quoted string keeps the ArcadeDB meaning.

Since v26.11.1. Before, `standard_conforming_strings` was reported `on` while every literal was read with the ArcadeDB SQL escapes: `'a\b'` was stored as `ab`, a literal ending in a backslash was a syntax error, `E'…'` did not parse, and a `SET` of the parameter to `off` was accepted and ignored. A client that wrote the ArcadeDB

> escapes in plain literals over the PostgreSQL protocol (`'line\nbreak'` for a line break, `'it's'`) now gets them literally: write them in an `E'…'` literal, or set `standard_conforming_strings` to `off` for the connection. Values bound as parameters were never affected.

A `SET` is answered `SET` only when it did what it asked. Every other `SET` is answered with an error, on both the simple and the extended query protocol, rather than accepted and ignored:

- a `SET` the server cannot parse (for example `SET x`, with neither `=` nor `TO`) is a syntax error (`42601`), reported at Parse on the extended protocol, like in PostgreSQL;
- `SET ROLE <role>` and `SET SESSION AUTHORIZATION <user>` are refused (`0A000`): a connection always keeps the privileges of the user it authenticated as, and a client told otherwise would believe it had dropped privileges it still holds. `SET ROLE NONE`, `SET SESSION AUTHORIZATION DEFAULT`, `RESET ROLE` and `RESET SESSION AUTHORIZATION` are accepted, since that is already the case;
- `SET TRANSACTION ISOLATION LEVEL …` and `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL …` (what the JDBC driver's `Connection.setTransactionIsolation()` sends) are accepted for the level transactions really run at (the database's `READ COMMITTED` by default, which is also what `READ UNCOMMITTED` means in PostgreSQL) and refused (`0A000`) for any other, such as `SERIALIZABLE`. To run a transaction at `REPEATABLE READ`, open it with `BEGIN ISOLATION LEVEL REPEATABLE READ`;
- `READ ONLY` transactions are not supported, so `SET TRANSACTION READ ONLY` and `SET default_transaction_read_only = on` are refused (`0A000`); `READ WRITE` and `[NOT] DEFERRABLE` are accepted;
- `SET CONSTRAINTS ALL DEFERRED | IMMEDIATE` is accepted without effect, as in PostgreSQL for constraints that are not deferrable, which ArcadeDB's never are;
- `SET NAMES '<encoding>'` and `SET SCHEMA '<schema>'` are accepted as synonyms of `SET client_encoding` and `SET search_path`. Since v26.10.2 `search_path` is accepted but not stored: a connection always resolves names in the database it is connected to, so `SHOW search_path` answers that database, as `current_schema()` does;
- `statement_timeout` is enforced since v26.10.2: a statement that runs longer is cancelled with `57014`. It takes a number of milliseconds or a value with a unit (`30s`, `2min`), and `0` disables it. The shorter of it and `arcadedb.command.timeout` applies;
- `DISCARD ALL` (what connection poolers such as pgbouncer send between two client sessions) resets every setting and drops the prepared statements and portals. It is refused inside a transaction block (`25001`), which a simple query of several statements is too. `DISCARD PLANS | SEQUENCES | TEMP` are accepted without effect, and `DEALLOCATE [ALL | <name>]` and `CLOSE [ALL | <name>]` are supported;
- a parameter PostgreSQL does not know is refused by `SET`, `SHOW` and `RESET` alike with `42704 unrecognized configuration parameter`, as in PostgreSQL (for example `SET bogus_param = 1` or `SHOW databases`). Every parameter PostgreSQL 17 knows is accepted, as is any name the client sent when it connected. A name with a dot (`SET myapp.tenant = 'acme'`) is a custom setting: it is always accepted, and `SHOW` reads it back once it is set;
- a parameter PostgreSQL lets only the server configuration change, such as `shared_buffers` or `wal_level`, is refused with `55P02`.

5.5.9. Cancelling a statement

Since v26.10.2 a cancel request works as in PostgreSQL. The `CancelRequest` that JDBC's `Statement.cancel()`, pgjdbc's `setQueryTimeout()` and libpq's `PQcancel` send stops the statement the connection is running: it fails with SQLSTATE `57014` (`query_canceled`), what it wrote is rolled back as after any other failed statement, and the connection stays open for the next one. A connection that is not running a statement is not affected.

Each connection gets its own backend process id and a random 32-bit secret key, which the client receives at connect time (`BackendKeyData`) and must quote in its cancel request. The statements a connection runs are listed by the HTTP `list queries` command and by Cypher `SHOW TRANSACTIONS` with `protocol postgres` and the process id as `connectionId`, and stopped by `terminate query` or `TERMINATE TRANSACTIONS` with the same `57014`.

As on PostgreSQL, a cancel request is not authenticated: the secret key is its only credential, and process ids are sequential. A cancel request that quotes a live connection with the wrong secret stops nothing and gets no answer; the server logs it, at `WARNING` at most once a minute with the number refused since, so guessing is visible in the log. A cancel is best effort, as on PostgreSQL: one that arrives between two statements stops nothing, and one that crosses the start of the next statement may stop that one.

Three SQL functions work on connections by process id, as their PostgreSQL namesakes:

Function	Returns
`pg_backend_pid()`	The process id of the calling connection, `null` when the statement does not run on a PostgreSQL connection.
`pg_cancel_backend(pid)`	Stops the statement that connection is running and leaves the connection open. `true` when the connection was signalled.
`pg_terminate_backend(pid)`	Stops the statement that connection is running and closes the connection, rolling back its open transaction. `true` when the connection was signalled.

root may signal every connection, any other user only their own. Both signal functions answer `false` for a process id that names no connection the caller may signal. They answer `NULL` for a `NULL` process id and fail on a fractional one (such as `1.5`), as on PostgreSQL. They signal only connections of the server that runs them, and are registered with the SQL functions of the JVM once the PostgreSQL plugin starts, so SQL sent over any protocol can call them.

```
SELECT pg_backend_pid();          -- this connection's process id, e.g. 1042
SELECT pg_cancel_backend(1042);   -- stop what connection 1042 is running
SELECT pg_terminate_backend(1042);-- stop it and close the connection
```

Before v26.10.2 a cancel request closed the whole connection, and since every connection had the backend process id `0`, a cancel could reach a different connection than the one it was meant for. A client that relied on a cancel request closing the connection must now call `pg_terminate_backend()` instead.

5.5.10. Creating an object whose name is taken

(Since v26.11.1.) A statement that creates, or renames onto, a name that is already taken fails with the SQLSTATE PostgreSQL uses for the same case: `42P07` (`duplicate_table`) for a type, an index or a view, `42701` (`duplicate_column`) for a property, and `42710` (`duplicate_object`) for anything else, such as a constraint, a bucket, a trigger or a function. Before v26.11.1 it was `XX000` (internal error), or `42P01` (undefined table) for some of them.

(Since v26.12.1.) An index equivalent to an existing one under another name (for example `CREATE INDEX b ON T (a) NOTUNIQUE` when an index on the same property with the same definition already exists) fails with `42P07` too, like a same-name duplicate. Before v26.12.1 it answered `22023` (`invalid_parameter_value`).

5.5.11. Naming an object that does not exist

(Since v26.12.1.) A statement that names a type, an index, a bucket, a property, a trigger or a view that does not exist fails with `42P01` (`undefined_table`), an unknown method or Cypher procedure with `42883` (`undefined_function`), and an unknown `ALTER DATABASE` setting with `22023` (`invalid_parameter_value`). Before v26.12.1 many of these answered `XX000` (internal error), so a typo looked like a server fault.

5.5.12. A refused value and an unknown function

(Since v26.12.1.) A value refused by a constraint declared on its property, or by the conversion to the property's type, fails with the SQLSTATE PostgreSQL answers the same refusal with, so a driver maps it to the right error class (Rails `NotNullViolation`, Django `IntegrityError` for class `23` against `DataError` for class `22`, Hibernate `ConstraintViolationException`):

Refusal	SQLSTATE	PostgreSQL name
`MANDATORY` or `NOTNULL`	`23502`	`not_null_violation`
`MIN`, `MAX`, `REGEXP`, `READONLY`	`23514`	`check_violation`
a `STRING` or `BINARY` longer than its `MAX`	`22001`	`string_data_right_truncation`
a value of another type (a link to another type, an embedded document of another type)	`42804`	`datatype_mismatch`
a number the property's type cannot hold	`22003`	`numeric_value_out_of_range`
text that is not a number	`22P02`	`invalid_text_representation`
text that is not a date	`22007`	`invalid_datetime_format`
a call of a function that does not exist	`42883`	`undefined_function`

Before v26.12.1 every refused value answered `22023` (`invalid_parameter_value`), and a call of an unknown SQL function answered `XX000` (internal error).

5.5.13. Schema changes and command tags

(Since v26.12.1.) A schema change (`CREATE TYPE`, `CREATE PROPERTY`, `CREATE INDEX`, `ALTER TYPE`, `DROP TYPE`, `REBUILD INDEX`, ...) is answered as PostgreSQL answers `CREATE TABLE`, on the simple and the extended protocol alike: no result set (`NoData` to a `Describe`), and a command tag naming the statement with no row count, for example `CREATE DOCUMENT TYPE`. So JDBC's `executeUpdate()` accepts it and returns `0`, and `execute()` returns `false`. Before v26.12.1 it answered a result set with one row describing the change and an empty tag: `executeUpdate()` threw "A result was returned when none was expected", and a client that described the statement before running it (asyncpg, pgx, npgsql) received an error with SQLSTATE `0A000` after the change had been applied.

The other statements that only acknowledge what they did are answered the same way: `DEFINE FUNCTION`, `DELETE FUNCTION`, `SLEEP`, `LOCK`, `SET GLOBAL`, `LET`, `CONSOLE`, `RESTORE …`, `BEGIN`/`COMMIT`/`ROLLBACK`, and an `INSERT`, `UPDATE`, `DELETE` or `CREATE VERTEX` without a `RETURN`. Before v26.12.1 `DEFINE FUNCTION`, `DELETE FUNCTION`, `SLEEP`, `LOCK`, `SET GLOBAL`, `LET` and `RESTORE` still answered a result set, with the same consequences.

The command tag of a statement (`INSERT 0 n`, `UPDATE n`, `DELETE n`, `SELECT n`) is the one of the statement the server parsed, whatever its text looks like: a leading comment (`/* hint */ INSERT …`, as Hibernate's `hibernate.use_sql_comments` writes), a line break or a doubled space between its keywords does not change it. Before v26.12.1 such a statement was tagged with an empty string, so JDBC reported 0 rows for rows it had written.

5.5.14. Generated keys (RETURNING)

(Since v26.12.1.) `INSERT`, `UPDATE` and `DELETE` accept PostgreSQL's trailing `RETURNING <projection>` clause (see `INSERT`, `UPDATE` and `DELETE`). The PostgreSQL JDBC driver appends `RETURNING *` to a statement prepared with `Statement.RETURN_GENERATED_KEYS`, and `RETURNING "col"` for the columns named in `prepareStatement(sql, String[])`, so `getGeneratedKeys()` answers the written records, and Hibernate `IDENTITY` ids, Spring's `KeyHolder` and jOOQ's generated keys work. Before v26.12.1 such a statement was refused with a syntax error.

5.5.15. Postgres Tools Known to Work

Some tools compatible with Postgres may execute queries on internal Postgres tables to retrieve the schema. Most of these queries - the ones asking about schemas, tables, columns and a few other well-known catalogs - are now answered generically (see above), but a tool asking about a `pg_catalog` object with no ArcadeDB equivalent (functions, triggers, extensions, and similar) gets an error with SQLSTATE `0A000` for that specific question, since v26.11.1, instead of an empty result that reads as an empty catalog. See tested compatible tools below. If the tool that you use to work with Postgres is not compatible with ArcadeDB, please open an issue.

PostgreSQL Client psql

Postgres's `psql` tool works out of the box, just like with an actual Postgres server. To install this Postgres client, see here.

Connect from a terminal or console like this:

```
psql -h localhost -p 5432 -d mydatabase -U root
```

After authenticating, you can run SQL queries as normal, and `\copy` exports a query or a type to a local file (see COPY … TO STDOUT above). One can also submit the password via the environment:

```
PGPASSWORD=password psql -h localhost -p 5432 -d mydatabase -U root
```

or use the `postgres` protocol address:

```
psql postgres://username:password@host:port/database
```

In case the password contains special characters (like `/`, `\`, `@`, `?`, `!`, `&`), it needs to be URL encoded (also known as "percent encoding").

Note, that in the `psql` console queries or commands need to be terminated with a semi-colon `;` to be submitted.

Apache Arrow (ADBC)

ADBC is Apache Arrow's database API: a client reads a result set as Arrow record batches instead of rows. Two drivers reach ArcadeDB over the Postgres protocol:

- The Apache Arrow ADBC PostgreSQL driver connects directly and works with its default settings since v26.10.1.
- adbcBridge is a driver for ADBC that sits on top of the PostgreSQL ODBC driver (psqlodbc).

With the Apache Arrow ADBC PostgreSQL driver in Python:

```
pip install adbc-driver-postgresql pyarrow
```

```
import adbc_driver_postgresql.dbapi as dbapi

conn = dbapi.connect("postgresql://root:password@localhost:5432/mydatabase")
with conn.cursor() as cur:
    cur.execute("SELECT name, price FROM Product ORDER BY name")
    print(cur.fetch_arrow_table())
conn.close()
```

5.6. MCP Server

ArcadeDB includes a built-in Model Context Protocol (MCP) server that allows AI assistants and LLM-based tools to interact with the database. The MCP server exposes database operations as tools

that AI clients can discover and invoke using the standard MCP protocol.

The MCP server implements the MCP specification (version 2025-03-26) over HTTP using the JSON-RPC 2.0 protocol.

5.6.1. Endpoint

All MCP requests are sent as `POST` requests to a single endpoint:

```
POST /api/v1/mcp
```

Requests must include a valid `Authorization` header (Basic Auth or Bearer token) and a JSON-RPC 2.0 body.

An alternative stdio transport carries the same protocol over standard input and output, for clients that launch their MCP servers as a child process rather than connecting to one over the network.

The MCP server ships as the optional `mcp` module and is installed automatically when its jar is on the classpath, so no `SERVER_PLUGINS` entry is needed. The standard distribution includes it. A distribution assembled without it, through the `arcadedb-builder.sh` module selection described in Custom Package Builder, registers neither endpoint and answers `404` on both.

5.6.2. Configuration

The MCP server is configured via the file `config/mcp-config.json` in the ArcadeDB server directory. You can also manage the configuration at runtime through the configuration endpoint.

Example `mcp-config.json`

```
{
  "enabled": true,
  "allowReads": true,
  "allowInsert": true,
  "allowUpdate": true,
  "allowDelete": false,
  "allowSchemaChange": false,
  "allowAdmin": false,
  "profile": "rag",
  "allowedUsers": ["root", "analyst"],
  "principalProfiles": {
    "analyst": "rag"
  },
  "databases": {
    "analytics": {
      "allowInsert": false,
      "allowedUsers": ["analyst"]
    }
  }
}
```

Property	Default	Description
`enabled`	`false`	Enable or disable the MCP server. When disabled, all MCP requests return an error.
`allowReads`	`true`	Allow read-only queries (SELECT, MATCH, etc.).
`allowInsert`	`false`	Allow INSERT operations.
`allowUpdate`	`false`	Allow UPDATE operations.
`allowDelete`	`false`	Allow DELETE operations.
`allowSchemaChange`	`false`	Allow schema modifications (CREATE TYPE, CREATE PROPERTY, CREATE INDEX, etc.).
`allowAdmin`	`false`	Allow administrative operations.
`profile`	`"all"`	Server-global tool profile: `"all"`, `"rag"`, or `"admin"`. See Tool Profiles.
`principalProfiles`	`{}`	Optional per-principal tool profiles, keyed by user name or API token name. Intersected with `profile`, so an entry can only narrow the tool surface. See Per-Principal Profiles.
`allowedUsers`	`["root"]`	List of usernames allowed to use the MCP server. Use `"*"` to allow all authenticated users.
`allowedOrigins`	`[]`	Extra browser `Origin` values accepted by the HTTP transport, on top of the loopback and same-host origins that are always allowed. See Transport.
`databases`	`{}`	Optional per-database permission and user restrictions. See Per-Database Scoping.

5.6.3. Tool Profiles

Tool profiles keep the surface presented to an agent focused. The selected profile filters `tools/list` and is enforced again by `tools/call`, so a client cannot invoke a hidden tool directly. A profile never grants an operation: read, write, schema, administrative, user, origin, and native database authorization checks remain independent and mandatory.

Profile	Tool membership
`all`	Every tool registered by the running ArcadeDB version. This is the default.
`rag`	`list_databases`, `get_schema`, `query`, `sample_records`, `vector_search`, `hybrid_search`, `full_text_search`, `upsert_entity`, and `upsert_relationship`.
`admin`	`list_databases`, `get_schema`, `query`, `execute_command`, `server_status`, `profiler_start`, `profiler_stop`, `profiler_status`, `get_server_settings`, and `set_server_setting`.

Only registered tools are advertised. For example, a version that does not include `hybrid_search` does not expose it merely because the name belongs to the `rag` profile. The agent-memory upsert tools in `rag` remain unavailable unless their independent write permissions are enabled; all write permissions default to false.

The `profile` setting is server-global and applies to every principal that has no entry in `principalProfiles`. Per-database overrides restrict permissions and users only; they cannot select a different profile. See Per-Database Scoping.

Profiles filter prompts as well, because a prompt's text names tools: a prompt is offered only when every tool it names survives the profile. See Prompts. Resources are not filtered by profile.

Per-Principal Profiles

Entries under `principalProfiles` assign a tool profile to an individual authenticated principal. The effective profile is the **intersection** of the server-global `profile` and the principal entry: a tool is advertised and callable only when both profiles contain it. An entry can therefore only narrow the surface, and never grants a tool that the global profile hides. As with the global profile, an intersection never grants a permission: read, write, schema, administrative, database, user, and origin checks remain independent and mandatory.

A principal is keyed by its user name. An API token can be keyed either canonically as `apitoken:<token-name>` or by the bare token name, the same two spellings that `allowedUsers` accepts. When both spellings are present, the canonical `apitoken:` entry wins.

Example: a retrieval agent restricted to the `rag` tools on an otherwise unrestricted server

```
{
  "enabled": true,
  "profile": "all",
  "allowedUsers": ["root", "retrieval-agent", "retrieval-token"],
  "principalProfiles": {
    "retrieval-agent": "rag",
    "apitoken:retrieval-token": "rag"
  }
}
```

Profile names are case-insensitive on input and are serialized in lowercase. A `tools/call` for a tool excluded by the intersection is rejected with an error naming both profiles, so a client cannot bypass the filtered `tools/list` by invoking the tool directly. The `initialize` instructions follow the effective profile: they describe the `rag` surface when the intersection matches the `rag` profile, and otherwise describe the general or the restricted surface.

The runtime configuration endpoint merges `principalProfiles` by principal name. Set one principal to `null` to remove that entry, or set the top-level `principalProfiles` value to `null` to clear all per-principal profiles:

```
{
  "principalProfiles": {
    "retrieval-agent": null,
    "reporting-agent": "rag"
  }
}
```

5.6.4. Per-Database Scoping

Entries under `databases` further restrict the server-global MCP policy for named databases. An omitted local field inherits its global value. A local `false` can deny an operation, but a local `true` cannot enable an operation denied globally. Similarly, a local `allowedUsers` list is an additional restriction: an authenticated principal must satisfy both the global and local lists. ArcadeDB's native database authorization remains an independent mandatory check.

Each database override accepts `allowReads`, `allowInsert`, `allowUpdate`, `allowDelete`, `allowSchemaChange`, `allowAdmin`, and `allowedUsers`. Unknown fields are rejected rather than silently inherited. An override naming a database that does not currently exist produces a startup warning but is retained so it can apply if that database is created later.

Example per-database restrictions

```
{
  "enabled": true,
  "allowReads": true,
  "allowInsert": true,
  "allowedUsers": ["root", "tenant-agent"],
  "databases": {
    "tenant-graph": {
      "allowInsert": false,
      "allowedUsers": ["tenant-agent"]
    }
  }
}
```

The runtime configuration endpoint merges updates by database name. A non-null entry replaces the complete override for that database, so fields omitted from the replacement inherit the global policy. Set one database entry to `null` to remove that override, or set the top-level `databases` value to `null` to clear all database overrides:

```
{
  "databases": {
    "tenant-graph": null,
    "reporting": {
      "allowReads": true,
      "allowInsert": false
    }
  }
}
```

Database discovery, server status, and schema resources omit databases whose effective policy denies the authenticated principal or read access.

5.6.5. Configuration Endpoint

The MCP configuration can be managed at runtime via the `/api/v1/mcp/config` endpoint.

Get current configuration:

```
curl -u root:arcadedb-password \
  http://localhost:2480/api/v1/mcp/config
```

Update configuration (root user only):

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"enabled": true, "allowReads": true, "allowInsert": true}' \
  http://localhost:2480/api/v1/mcp/config
```

An update is written to `config/mcp-config.json` before it is applied. If the file cannot be written (a read-only configuration directory, such as a Kubernetes ConfigMap mount or a read-only root filesystem, or a full disk), the request answers HTTP `500` with an error naming the file, and the configuration in force is left unchanged. *(Since v26.12.1; before, the request answered `200` and applied the change in memory only, so the next restart reverted to the old file.)* On first start with a read-only configuration directory the server still starts with the default configuration: writing the initial file is best effort.

`config/mcp-config.json` is loaded all or nothing. A file that cannot be read or parsed, including one with a single malformed element in `allowedUsers` or `allowedOrigins` or a flag that is not a boolean, is rejected as a whole: MCP starts disabled with every setting at its default, the server logs a warning naming the file, and the file is left untouched so it can be fixed. *(Since v26.12.1; before, the settings read ahead of the malformed value could stay in force.)*

5.6.6. MCP Protocol Methods

The MCP server supports the following JSON-RPC 2.0 methods:

Method	Description
`initialize`	Handshake that returns the protocol version and server capabilities.
`notifications/initialized`	Client notification acknowledging initialization.
`tools/list`	Returns the list of available tools with their JSON schemas.
`tools/call`	Executes a specific tool by name with the provided arguments.
`resources/list`	Returns the readable resources. See Resources.
`resources/read`	Returns the contents of one resource, addressed by URI. See Resources.
`prompts/list`	Returns the available guided prompt templates. See Prompts.

Method	Description
`prompts/get`	Renders one prompt with its arguments substituted. See Prompts.
`ping`	Health check that returns a pong response.

The `initialize` response advertises all three surfaces under `capabilities`, as `tools`, `resources`, and `prompts`. None of them emits change notifications and resources cannot be subscribed to, so `listChanged` and `subscribe` are all false.

The response also carries an `instructions` string describing the surface the client actually received. It follows the effective tool profile, so a `rag` client is told about the retrieval and upsert tools and the guided prompts, while a client whose profile hides most of the surface is told to use only what `tools/list` returned.

Example: Initialize

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"protocolVersion":"2025-
03-26","clientInfo":{"name":"my-client","version":"1.0"}}}' \
  http://localhost:2480/api/v1/mcp
```

Example: List available tools

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":2,"method":"tools/list"}' \
  http://localhost:2480/api/v1/mcp
```

5.6.7. Available Tools

The MCP server exposes a version-dependent set of tools that AI clients can invoke. The commonly used tools described below include `list_databases`, `get_schema`, `query`, `execute_command`, `server_status`, `sample_records`, `vector_search`, `full_text_search`, `hybrid_search`, `upsert_entity`, and `upsert_relationship`. The operator tools `profiler_start`, `profiler_stop`, `profiler_status`, `get_server_settings`, and `set_server_setting` are described below as well. Call `tools/list` for the authoritative list, including the full input schema of every tool.

Tool Annotations

Since v26.11.1 every tool returned by `tools/list` carries MCP `annotations`: a `title` and the hints `readOnlyHint`, `destructiveHint`, `idempotentHint` and `openWorldHint`. AI clients use them to decide which calls they can run freely and which ones to confirm with the user first.

Tools	Read-only	Destructive	Idempotent	Open world
`list_databases`, `get_schema`, `sample_records`, `vector_search`, `full_text_search`, `hybrid_search`, `server_status`, `profiler_status`, `get_server_settings`	yes	no	yes	no
`query`	yes	no	yes	yes
`upsert_entity`, `upsert_relationship`	no	yes	no	no
`profiler_start`	no	yes	no	no
`profiler_stop`	no	no	no	no
`execute_command`	no	yes	no	yes
`set_server_setting`	no	yes	no	no

`query` and `execute_command` are "open world" because Cypher `LOAD CSV` and `IMPORT DATABASE` can fetch remote URLs. The upserts are destructive because they overwrite the properties of a record they match.

Annotations are hints for the client only. They never grant or restrict anything: the MCP configuration and the user's permissions are enforced by the server as before.

`list_databases`

Lists all databases accessible to the authenticated user.

Parameters: None

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"list_databases","argu
ments":{}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "databases": ["mydb", "analytics"]
}
```

`get_schema`

Retrieves the full schema of a database, including all types (vertex, edge, document), their

properties, and indexes.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"get_schema","argument
s":{"database":"mydb"}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "database": "mydb",
  "types": [
    {
      "name": "Person",
      "category": "vertex",
      "parentTypes": ["V"],
      "properties": [
        {
          "name": "name",
          "type": "STRING",
          "mandatory": true,
          "readonly": false,
          "notNull": true
        }
      ],
      "indexes": [
        {
          "name": "Person[name]",
          "type": "LSM_TREE",
          "properties": ["name"],
          "unique": true
        }
      ]
    }
  ]
}
```

`query`

Executes a read-only (idempotent) query against a database. The server performs semantic analysis to ensure the query does not modify data. If a write operation is detected, the request is rejected.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`language`	Yes	Query language: `sql`, `cypher`, `gremlin`, `graphql`, `mongo`.
`query`	Yes	The query string to execute.
`limit`	No	Maximum number of records to return, from 1 through 100,000 (default: 1000).

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"query","arguments":{"
database":"mydb","language":"sql","query":"SELECT FROM Person LIMIT 10"}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "records": [
    {"@rid": "#1:0", "@type": "Person", "name": "Alice"},
    {"@rid": "#1:1", "@type": "Person", "name": "Bob"}
  ],
  "count": 2
}
```

`execute_command`

Executes a non-idempotent command (write operations, schema changes, administrative commands). The server checks the command against the MCP configuration permissions before execution.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`language`	Yes	Command language: `sql`, `sqlscript`, `cypher`, `gremlin`, `graphql`, `mongo`.

Parameter	Required	Description
`command`	Yes	The command string to execute.
`limit`	No	Maximum number of records to return, from 1 through 100,000 (default: 1000).

Permission mapping:

The server analyzes the command semantics and checks against the MCP configuration:

Operation Type	Required Permission
INSERT	`allowInsert`
UPDATE	`allowUpdate`
DELETE	`allowDelete`
CREATE TYPE, CREATE PROPERTY, CREATE INDEX, etc.	`allowSchemaChange`
Administrative operations	`allowAdmin`
SELECT, MATCH	`allowReads`

`allowSchemaChange` also covers the types a write creates implicitly *(since v26.11.1)*: a Cypher `CREATE` or `MERGE` naming a label or relationship type that does not exist yet, through `execute_command` or the upsert tools, is refused when `allowSchemaChange` is off, while writes to existing types need only `allowInsert` / `allowUpdate`.

The check covers everything the command runs, not only its first keyword *(since v26.10.2)*. A write nested in a `LET`, a projection, a `FROM` subquery, a `CASE` or a script block needs the permission for that write, a DDL statement inside an `IF`, `FOREACH` or `WHILE` block needs `allowSchemaChange`, and `PROFILE` needs what the statement it wraps needs. In the same way the read-only `query` tool refuses a statement with a nested write.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"execute_command","arg
uments":{"database":"mydb","language":"sql","command":"INSERT INTO Person SET name =
'\''Charlie'\''"}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "records": [
```

```
    {"@rid": "#1:2", "@type": "Person", "name": "Charlie"}
  ],
  "count": 1
}
```

server_status

Returns server metadata including version, available query languages, and accessible databases.

Parameters: None

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"server_status","argum
ents":{}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "version": "26.10.2",
  "serverName": "ArcadeDB_0",
  "languages": ["sql", "sqlscript", "cypher", "gremlin", "graphql", "mongo"],
  "databases": ["mydb", "analytics"],
  "ha": {
    "clusterName": "arcadedb",
    "leader": "server1",
    "replicas": ["server2", "server3"]
  }
}
```

The `ha` field is only included when High Availability is enabled and the user has admin permissions.

sample_records

Returns the first records in storage order from selected database types so an agent can inspect actual property-value shapes before constructing a query. This is not a random or representative sample.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`types`	No	Up to 20 type names. Explicit names may include edge types. When omitted, ArcadeDB selects the first 20 vertex and document type names in alphabetical order and excludes edge types.
`limit`	No	Maximum records per type, from 1 through 20 (default: 3).

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"sample_records","argu
ments":{"database":"mydb","types":["Person","Company"],"limit":3}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "samples": {
    "Person": [
      {"@rid": "#1:0", "@type": "Person", "name": "Alice"}
    ],
    "Company": [
      {"@rid": "#2:0", "@type": "Company", "name": "Acme"}
    ]
  },
  "sampledTypes": 2,
  "availableTypes": 2,
  "recordsReturned": 2,
  "truncated": false
}
```

`availableTypes` always reports the number of eligible vertex and document types in the database, even when `types` is supplied. When `types` is omitted, `truncated` indicates that more than 20 eligible types exist. The generated queries are checked as read-only, and serialized records do not include vertex edge collections.

`vector_search`

Searches a dense `LSM_VECTOR` or sparse `LSM_SPARSE_VECTOR` index with a pre-computed query vector. ArcadeDB does not generate embeddings; the caller must supply the vector produced by its embedding model.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`indexName`	Yes	The name of an `LSM_VECTOR` or `LSM_SPARSE_VECTOR` index.
`queryVector`	Yes	Dense vector values, or sparse weights corresponding to `queryIndices`.
`queryIndices`	Sparse only	Sparse dimension identifiers corresponding to the weights in `queryVector`. Omit to use each `queryVector` position as its dimension.
`k`	Yes	Maximum results, from 1 through 1,000.
`efSearch`	No	Dense-index search beam width. Higher values can improve recall at additional cost. This option is rejected for sparse indexes.
`filter`	No	Read-only SQL `WHERE` predicate applied to a bounded candidate set.
`sparse`	No	Use `vector.sparseNeighbors` against an `LSM_SPARSE_VECTOR` index (default: false).

The optional `filter` is evaluated against each expanded neighbor row. Record properties are flattened into that row, and `@rid`, `@type`, `record`, plus `distance` for dense results or `score` for sparse results are also available.

Dense-vector example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"vector_search","argum
ents":{"database":"research","indexName":"Paper[embedding]","queryVector":[0.12,-
0.08,0.31],"k":20,"efSearch":100,"filter":"publishedYear >= 2024"}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "indexName": "Paper[embedding]",
  "sparse": false,
  "scoring": "distance_lower_is_better:COSINE",
  "candidateLimit": 160,
  "truncated": true,
  "count": 20,
  "results": [
    {
      "rid": "#12:34",
      "type": "Paper",
```

```
      "distance": 0.083,
      "properties": {
        "title": "Example paper",
        "publishedYear": 2025
      }
    }
  ]
}
```

Dense results expose only `distance`, where lower is better. Sparse results expose only `score`, where higher is better. The response-level `scoring` value also records the active dense similarity function or sparse scoring modifier.

Without a filter, `candidateLimit` equals `k`. With a filter, ArcadeDB examines up to `min(k * 8, 8000)` vector candidates before applying the predicate. `truncated` is true when the returned result window is full (`count` equals `k`), so more matches may exist. When it is false, fewer than `k` matches survived the current candidate window. For a filtered search, that does not prove that no matching vectors exist beyond `candidateLimit`; increasing `k` also expands the candidate window until the 8,000-candidate ceiling is reached.

`full_text_search`

Searches a full-text index and returns the matching records ranked by relevance score. Requires read permission.

Address the index either by `indexName`, or by `typeName` with optional `properties`. When both forms are supplied, `indexName` wins.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`queryText`	Yes	The full-text query. A blank query is rejected.
`indexName`	Either	The name of the full-text index, for example `Article[content]` or `Article[title,content]`.
`typeName`	Either	The type carrying the index, for example `Article`. An index declared on a supertype is named for the supertype.
`properties`	No	Indexed properties, used with `typeName`, given in the order the index declares them.
`limit`	No	Maximum results (default: 10).

Query syntax:

Form	Meaning
`+a +b`	Both terms required.
`a -b`	Excludes `b`.

Form	Meaning
`a b`	Matches either term.
`"exact phrase"`	All terms must appear in the same record. Term order is **not** enforced.
`pre*`	Prefix match.
`term~`	Fuzzy match.
`field:term`	Restricts to one property of a multi-property index.
`term^2`	Boosts a term.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"full_text_search","ar
guments":{"database":"research","indexName":"Paper[abstract]","queryText":"+graph
+retrieval -sql","limit":5}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "indexName": "Paper[abstract]",
  "similarity": "BM25",
  "count": 2,
  "results": [
    {
      "rid": "#12:34",
      "score": 4.71,
      "properties": {
        "title": "Graph retrieval over citation networks"
      }
    }
  ]
}
```

The `similarity` field reports whether scores are `BM25` or legacy `CLASSIC` coordination counts. The limit is pushed down into the index: a query made of OR-ed terms keeps only the best `limit` matches across all buckets (*since v26.12.1*), any other query keeps the best `limit` of each bucket, so the search never merges every match in the index. Because of that push-down, a record deleted concurrently between the index scan and the record read is skipped without being back-filled, so a search can legitimately return fewer than `limit` results.

`hybrid_search`

Retrieves records by fusing up to three retrieval legs into a single ranked list: a vector search, an optional full-text search, and an optional graph expansion. Requires read permission. As with `vector_search`, ArcadeDB does not generate embeddings.

The graph expansion is seeded from the records the other legs found and walks outward from them, so a neighbor of a strong match can itself rank. Expanded records carry the number of hops from their seed and the path back to it.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`vectorIndexName`	Yes	The name of an `LSM_VECTOR` or `LSM_SPARSE_VECTOR` index.
`queryVector`	Yes	Dense vector values, or sparse weights corresponding to `queryIndices`.
`k`	Yes	Maximum results after fusion, from 1 through 1,000.
`queryIndices`	Sparse only	Sparse dimension identifiers corresponding to the weights in `queryVector`.
`sparse`	No	Use `vector.sparseNeighbors` against an `LSM_SPARSE_VECTOR` index (default: false).
`efSearch`	No	Dense-index search beam width. Rejected for sparse indexes.
`filter`	No	Read-only SQL `WHERE` predicate applied to the vector leg's bounded candidate set.
`fulltextIndexName`	With `fulltextQuery`	The full-text index for the second leg.
`fulltextQuery`	With `fulltextIndexName`	The full-text query for the second leg.
`expand`	No	Graph expansion leg. See below.
`fusionStrategy`	No	`RRF`, `DBSF`, or `LINEAR` (default: `RRF`).
`weights`	No	Per-leg fusion weights, keyed by leg name.

Supplying only one of `fulltextIndexName` and `fulltextQuery` is rejected: half a leg is always a mistake, and ignoring it silently would return a plausible result set that quietly dropped what the caller asked for.

The `expand` object:

Field	Required	Description
`edgeTypes`	No	Edge types to traverse. Omit to traverse every edge type.
`direction`	No	`out`, `in`, or `both` (default: `out`).

Field	Required	Description
`maxDepth`	No	Hops to walk, from 1 through 3 (default: 1).

An unknown edge type is rejected, listing the available ones. Traversing a name that matches nothing would otherwise return an empty neighborhood without an error, silently degrading the search to plain two-way fusion. `maxDepth` above 3 is rejected rather than clamped, and the cap is enforced server-side regardless of the value sent. Graph expansion requires a vertex type; requesting it against a document type is rejected.

Fusion strategies:

`RRF` (Reciprocal Rank Fusion) is rank-only and ignores per-row scores. `DBSF` and `LINEAR` normalize per-source scores and therefore require a numeric score on every row of every source.

The graph expansion leg ranks by traversal order and carries no score, so **`DBSF` and `LINEAR` cannot be combined with `expand`** and that combination is rejected. Both remain available for vector plus full-text fusion.

Weights:

`weights` accepts `vector`, `fulltext`, and `expand`. The first two default to `1.0` and `expand` defaults to `0.5`, because under `RRF` a rank-one expanded record would otherwise score exactly what the nearest neighbor scores, letting an arbitrary one-hop neighbor tie the best semantic match.

A weight naming a leg the request does not ask for is rejected, as is an unknown key, a negative value, and a non-numeric value. A weight of `0.0` is accepted: it keeps a leg's records in the candidate set and in each result's `sources` while contributing nothing to the ranking.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"hybrid_search","argum
ents":{"database":"research","vectorIndexName":"Paper[embedding]","queryVector":[0.12,
-0.08,0.31],"fulltextIndexName":"Paper[abstract]","fulltextQuery":"graph
retrieval","expand":{"edgeTypes":["CITES"],"direction":"out","maxDepth":2},"fusionStra
tegy":"RRF","k":10}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "vectorIndexName": "Paper[embedding]",
  "sparse": false,
  "scoring": "distance_lower_is_better:COSINE",
  "fused": true,
```

```
  "fusionStrategy": "RRF",
  "fulltextIndexName": "Paper[abstract]",
  "legs": {
    "vector": { "count": 40 },
    "fulltext": { "indexName": "Paper[abstract]", "similarity": "BM25", "count": 27 },
    "expand": {
      "direction": "out",
      "edgeTypes": ["CITES"],
      "maxDepth": 2,
      "truncated": false,
      "seedCount": 61,
      "seedsTruncated": false,
      "count": 133
    }
  },
  "truncated": true,
  "count": 10,
  "results": [
    {
      "rid": "#12:34",
      "fusedScore": 0.0325,
      "sources": ["vector", "fulltext"],
      "properties": { "title": "Graph retrieval over citation networks" }
    },
    {
      "rid": "#12:99",
      "fusedScore": 0.0161,
      "sources": ["expand"],
      "depth": 1,
      "path": ["#12:34", "#12:99"],
      "properties": { "title": "Citation graphs as retrieval context" }
    }
  ]
}
```


`sources` names the legs that contributed to each result, so a fused score can be interpreted rather than taken on faith: it distinguishes a top semantic match from a record that arrived two hops from something relevant. `depth` and `path` appear only on results the expansion leg contributed, and `path` starts at the seed and ends at the result itself.

`fused` reports whether fusion actually ran. Fusion requires at least two sources, so a request naming neither `fulltextQuery` nor `expand` cannot be fused: that response sets `fused` to false and its results carry the vector leg's native `distance` or `score` instead of `fusedScore`. The full-text index name and similarity are reported under `legs` whenever that leg ran, including when it matched nothing.

The response carries several independent budgets. Top-level `truncated` means the result window filled (`count` equals `k`), so raising `k` may surface more. `legs.expand.truncated` means the traversal hit its fan-out cap, so the neighborhood was only partly explored; narrowing `edgeTypes` or `maxDepth` gives a more complete picture of the part that matters. `legs.expand.seedsTruncated` means the retrieval legs produced more candidate seeds than the traversal budget accepts, so expansion

started from a subset.

For pure two-way fusion without graph expansion, `vector.fuse` remains available through the `query` tool.

`upsert_entity`

Creates or updates a single vertex addressed by a match key, without duplicating it. Requires **both** insert and update permission, because a merge resolves to a create or an update depending on what already exists.

The vertex is matched, or created, by the `matchKeys` property/value pairs, and then any `setProperties` are written. Repeated calls with identical `matchKeys` resolve to the same vertex. Values are bound as parameters, so they are safe to pass verbatim; identifiers are quoted.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`typeName`	Yes	The vertex type. Created automatically if it does not exist, when `allowSchemaChange` is enabled.
`matchKeys`	Yes	property/value pairs used as the match key. Must be non-empty; values should be scalars.
`setProperties`	No	property/value pairs to write on the matched or created vertex.

Create a `UNIQUE` index on the `matchKeys` properties. Without one the match is a full type scan on every call, and two concurrent upserts carrying the same keys can each create a vertex, which is exactly the duplication the tool exists to prevent.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"upsert_entity","argum
ents":{"database":"memory","typeName":"Person","matchKeys":{"email":"ada@example.com"}
,"setProperties":{"name":"Ada Lovelace","role":"analyst"}}}}' \
  http://localhost:2480/api/v1/mcp
```

`upsert_relationship`

Creates or updates a single directed edge between two vertices, without duplicating it. Requires **both** insert and update permission.

Each endpoint is matched, or created, by its own match keys; then the edge of type `relType` between them is matched or created, and any `relProperties` are written. Repeated calls between the same

resolved endpoints with the same `relType` resolve to the same edge.

Parameters:

Parameter	Required	Description
`database`	Yes	The name of the database.
`fromType`	Yes	The source vertex type. Created automatically if it does not exist, when `allowSchemaChange` is enabled.
`fromMatchKeys`	Yes	property/value pairs identifying the source vertex. Must be non-empty.
`toType`	Yes	The destination vertex type. Created automatically if it does not exist, when `allowSchemaChange` is enabled.
`toMatchKeys`	Yes	property/value pairs identifying the destination vertex. Must be non-empty.
`relType`	Yes	The edge type. Created automatically if it does not exist, when `allowSchemaChange` is enabled.
`relProperties`	No	property/value pairs to write on the matched or created edge.

As with `upsert_entity`, create a `UNIQUE` index on each endpoint's match-key properties. Without one each endpoint match is a full type scan, and concurrent calls can create duplicate endpoints.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"upsert_relationship",
"arguments":{"database":"memory","fromType":"Person","fromMatchKeys":{"email":"ada@exa
mple.com"},"toType":"Project","toMatchKeys":{"code":"ANALYTICAL-
ENGINE"},"relType":"WORKS_ON","relProperties":{"since":1843}}}}' \
  http://localhost:2480/api/v1/mcp
```

The edge properties are written in a trailing assignment, so the edge is keyed by its endpoints and type alone and is never duplicated by a change to its properties.

`profiler_start`

Starts the query profiler, which records every query with its execution time and plan. Requires admin permission.

The profiler auto-stops after `timeoutSeconds`, so a session left running does not profile indefinitely. Use `profiler_stop` to end it early and collect the results, or `profiler_status` to check progress while it runs.

Parameters:

Parameter	Required	Description
`timeoutSeconds`	No	Recording duration, from 1 through 3,600 seconds (default: 60). The profiler stops itself when it elapses.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"profiler_start","argu
ments":{"timeoutSeconds":120}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "status": "started",
  "timeoutSeconds": 120,
  "message": "Query profiler started. It will auto-stop after 120 seconds. Use
profiler_stop to stop early and retrieve results."
}
```

Only one profiling session runs at a time. Starting the profiler while it is already recording does **not** restart it, extend it, or raise an error: the call returns `status: "already_recording"` and leaves the session untouched, so an agent retrying a start cannot silently discard the data collected so far.

`profiler_stop`

Stops the query profiler and returns the data captured during the session. Requires admin permission. Takes no parameters.

Results are aggregated per query and include execution counts, timing (minimum, maximum, average, and p99), and execution-plan step costs. They are also written to disk, so a session can be retrieved after the fact.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"profiler_stop","argum
ents":{}}}' \
  http://localhost:2480/api/v1/mcp
```

If the profiler was not recording, the call returns `status: "not_recording"` rather than an error, so stopping an already-stopped profiler is safe to retry.

`profiler_status`

Reports whether the profiler is recording and returns the current or most recent results. Requires admin permission. Takes no parameters.

Where `profiler_stop` ends the session, this reads it without disturbing it, so it is the safe call for an agent that wants to check progress mid-session. Alongside the captured queries and their timing statistics, the response carries server metric snapshots taken during the session.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"profiler_status","arg
uments":{}}}' \
  http://localhost:2480/api/v1/mcp
```

`get_server_settings`

Returns the server-level configuration settings with their current values, defaults, and descriptions. Takes no parameters.

This tool requires **read** permission, not admin permission, even though it belongs to the `admin` tool profile. Profile membership and permission gating are independent: see Tool Profiles.

Only settings whose scope is the server are returned; database-scoped settings are excluded.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"get_server_settings",
"arguments":{}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "settings": [
    {
```

```
      "key": "arcadedb.asyncWorkerThreads",
      "value": 7,
      "description": "Number of asynchronous worker threads. 0 (default) = available
cores minus 1",
      "overridden": true,
      "default": 0
    },
    {
      "key": "arcadedb.ha.clusterToken",
      "value": "*****",
      "description": "Token to authenticate a server joining the cluster",
      "overridden": true,
      "default": "*****"
    }
  ]
}
```

`value` is what this server is actually running on, and `default` the built-in default. `overridden` is true when the setting has been given an explicit value rather than inheriting its default - in the server configuration file, by an embedding application, or at runtime with `set_server_setting`.

Secret settings are masked as `*` in both `value` and `default`. The rule covers the HA cluster token and every key containing `password`, and it is applied by the server rather than by the client, so a secret is never serialized into the response in the first place.

`arcadedb.server.defaultDatabases` is masked differently, because its value merely **embeds** credentials rather than being one: the database names, user names and groups are kept, and only each password is replaced, so `mydb[jay:hunter2:admins]` is reported as `mydb[jay:*:admins]`.

`set_server_setting`

Updates a server configuration setting at runtime. Requires admin permission.

Changes take effect immediately. Whether they survive a restart depends on the individual setting, so treat this as a runtime adjustment rather than a way to persist configuration. Call `get_server_settings` first to see the available keys and their current values.

Parameters:

Parameter	Required	Description
`key`	Yes	The configuration key, for example `arcadedb.asyncWorkerThreads`.
`value`	Yes	The new value, as a string.

An unknown key is rejected by name rather than silently accepted, so a typo cannot masquerade as a successful update.

Example:

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"set_server_setting","
arguments":{"key":"arcadedb.asyncWorkerThreads","value":"8"}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "key": "arcadedb.asyncWorkerThreads",
  "previousValue": "7",
  "newValue": "8",
  "message": "Setting 'arcadedb.asyncWorkerThreads' updated successfully."
}
```

The response reports the replaced value so a change can be reverted without a separate read.

For a secret setting, `previousValue` is masked as `*`, using the same rule `get_server_settings` applies. Otherwise the setter would disclose through the value it replaced exactly what the getter withholds. The mask depends on the setting rather than on whether it currently holds a value, so an unset secret cannot be distinguished from a set one.

5.6.8. Resources

Besides tools, the MCP server exposes a read-only Resources surface. Every database publishes its schema at:

```
arcadedb://{database}/schema
```

The resource is served with the MIME type `application/json` and its `text` payload is the same document that the `get_schema` tool returns. A client that reads the resource at session start therefore obtains the schema without spending a tool call.

Example: list resources

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":1,"method":"resources/list"}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "resources": [
    {
      "uri": "arcadedb://mydb/schema",
      "name": "mydb schema",
      "description": "Schema of the ArcadeDB database 'mydb': types (vertex, edge,
document), properties, indexes, inheritance.",
      "mimeType": "application/json"
    }
  ]
}
```

Example: read a resource

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":2,"method":"resources/read","params":{"uri":"arcadedb://mydb/sc
hema"}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "contents": [
    {
      "uri": "arcadedb://mydb/schema",
      "mimeType": "application/json",
      "text": "{\"database\":\"mydb\",\"types\":[...]}"
    }
  ]
}
```

Resources are gated by the same policy as reads. `allowReads` must be enabled globally, the effective per-database policy must allow the authenticated principal to read that database, and ArcadeDB's native database authorization applies as usual. Tool profiles do not filter resources: a `rag` or `admin` profile changes the tool list only.

`resources/list` is a discovery call that most clients issue unprompted at session start, so a denial is reported as an empty `resources` array rather than as an error, and databases the principal cannot read are omitted. `resources/read` rejects an unknown database, an unauthorized database, and a malformed URI identically with JSON-RPC error `-32002` (`Resource not found`), so the surface cannot be used to probe which databases exist. When `allowReads` is disabled, `resources/read` returns error `-32600`.

5.6.9. Prompts

The MCP server also exposes a Prompts surface: guided templates that steer a model through a task using the retrieval and agent-memory tools, instead of leaving it to hand-write vector or `MERGE` syntax. A client typically offers them to the user as named commands.

Prompt	Purpose
`graphrag_query`	Answer a question over a database with its retrieval tools, citing the records the answer came from.
`build_knowledge_graph`	Extract entities and relationships from text and write them into a database without creating duplicates.

Both prompts take two required arguments.

Prompt	Argument	Description
`graphrag_query`	`database`	The name of the database to search.
	`question`	The question to answer from the contents of the database.
`build_knowledge_graph`	`database`	The name of the database to write into.
	`sourceText`	The text to extract entities and relationships from.

The rendered messages are static text with the arguments substituted verbatim, so `prompts/get` opens no database and reads no data.

`graphrag_query` directs the model to load the schema first, choose between `vector_search`, `full_text_search`, `hybrid_search`, and `query` according to the shape of the question, supply its own embedding vector because ArcadeDB does not generate one, and cite every claim with the `@rid` it came from.

`build_knowledge_graph` centers on match-key selection, which is what keeps repeated extraction from duplicating vertices. It directs the model to read the existing schema and reuse its types, choose the smallest stable match key per entity, prefer one backed by a `UNIQUE` index and report a missing one rather than creating it, then call `upsert_entity` once per entity and `upsert_relationship` once per relationship, reusing exactly the same match keys for the endpoints.

Availability

A prompt is a script that names tools by name, so it is offered only when the caller can actually run it. Each prompt is listed, and rendered, only when both conditions hold:

- every tool its text names is allowed by the caller's effective tool profile, and
- the permissions those tools require are granted server-globally.

Prompt	Tools named	Permissions required
`graphrag_query`	`get_schema`, `query`, `vector_search`, `full_text_search`, `hybrid_search`	`allowReads`

Prompt	Tools named	Permissions required
`build_knowledge_graph`	`get_schema`, `upsert_entity`, `upsert_relationship`	`allowReads`, `allowInsert`, `allowUpdate`

Neither prompt is therefore available under the `admin` profile, which contains none of the retrieval or upsert tools. Under `rag` and `all`, `graphrag_query` is available whenever reads are permitted, and `build_knowledge_graph` additionally requires the two write permissions, which are false by default.

These checks are evaluated against the server-global permission flags, because no database is resolved while a prompt is rendered. Per-database overrides still apply to every tool call the model subsequently makes.

Like `resources/list`, `prompts/list` is a discovery call most clients issue unprompted at session start, so a caller entitled to no prompts receives an empty `prompts` array rather than an error. Availability is then re-checked by `prompts/get`, for the same reason `tools/call` re-checks the profile after `tools/list` has already filtered: an unavailable prompt is refused with error `-32600` even when it is requested by name. An unknown prompt name, or a missing argument, returns `-32602`.

Example: list prompts

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":1,"method":"prompts/list"}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "prompts": [
    {
      "name": "graphrag_query",
      "description": "Answer a question over an ArcadeDB database using its retrieval
tools, citing the records the answer came from.",
      "arguments": [
        {"name": "database", "description": "The name of the database to search",
"required": true},
        {"name": "question", "description": "The question to answer from the contents
of the database", "required": true}
      ]
    }
  ]
}
```

Example: render a prompt

```
curl -u root:arcadedb-password \
```

```
  -X POST \
  -H "Content-Type: application/json" \
  -d
'{"jsonrpc":"2.0","id":2,"method":"prompts/get","params":{"name":"graphrag_query","arg
uments":{"database":"research","question":"Which papers cite the original RRF
paper?"}}}' \
  http://localhost:2480/api/v1/mcp
```

Response:

```
{
  "description": "Answer a question over an ArcadeDB database using its retrieval
tools, citing the records the answer came from.",
  "messages": [
    {
      "role": "user",
      "content": {
        "type": "text",
        "text": "Answer the following question using the ArcadeDB database
'research'.\n\n<question>\n..."
      }
    }
  ]
}
```

Each argument is substituted inside a delimiter block that marks it as data rather than procedure. Because the value is substituted verbatim, a value that itself carries the closing tag would end that block early and have the remaining text read as instructions. The server therefore suffixes both delimiters with an unpredictable token whenever the value contains such a tag, which moves the boundary instead of altering the value. An ordinary argument carries no such tag and renders against the constant delimiters shown above.

5.6.10. Transport

The same protocol surface is served over two transports: HTTP, and standard input and output. Tools, resources, prompts, and every permission and profile rule behave identically on both; only the envelope differs.

HTTP

The HTTP endpoint follows the Streamable HTTP transport rules of MCP 2025-03-26.

HTTP methods. JSON-RPC is carried over `POST` only. The server does not offer a Server-Sent Events stream, so any other method (including `GET`) is answered with `405 Method Not Allowed` and an `Allow: POST` header.

Requests and notifications. A request must carry an `id` that is a string or an integer; `null`, fractional and structured ids are rejected with JSON-RPC error `-32600`. A notification is a message with a `method` and no `id`: the server never answers one, and a `POST` that carried only notifications

returns `202 Accepted` with an empty body.

Batches. A top-level JSON array is accepted on both the HTTP and the stdio transport. Every element is dispatched independently and only the elements that are requests contribute a response, so a batch of notifications alone yields `202 Accepted` with no body and a mixed batch yields an array holding just the request responses.

```
curl -u root:arcadedb-password \
  -X POST \
  -H "Content-Type: application/json" \
  -d
'[{"jsonrpc":"2.0","id":1,"method":"ping"},{"jsonrpc":"2.0","id":2,"method":"tools/lis
t"}]' \
  http://localhost:2480/api/v1/mcp
```

Origin validation. To mitigate DNS rebinding, a request carrying an `Origin` header is accepted only when the origin is a loopback address, matches the host the request was addressed to, or is listed in `allowedOrigins`. Anything else is rejected with `403`. A request with no `Origin` is accepted, because non-browser MCP clients do not send one and the header is attached by the browser precisely in the case the check is meant to catch. Setting `allowedOrigins` to `["*"]` accepts any origin and disables this protection: use it only when the endpoint is fronted by a proxy that performs its own origin control.

Stdio

Clients that launch their MCP servers as a child process, rather than connecting to one over the network, speak JSON-RPC 2.0 over standard input and output with one message per line. The distribution ships that transport as `bin/mcp-stdio.sh`, which starts an ArcadeDB server and serves MCP on its stdin and stdout.

```
export ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=arcadedb-password"
bin/mcp-stdio.sh
```

The root password is mandatory: the process authenticates as `root` and exits with an error if neither `arcadedb.server.rootPassword` nor `arcadedb.server.rootPasswordPath` is set. `arcadedb.server.rootPasswordPath` names a file holding the password, such as a mounted Kubernetes Secret, so that the password never appears on a command line; one trailing line terminator is dropped, as for the server itself. `arcadedb.server.rootPassword` wins when both are set, and a file that cannot be read or is empty ends the process with an error that names its path *(since v26.11.1)*. Because the client owns stdout, the server redirects its own logging to stderr so that log lines cannot corrupt the JSON-RPC stream.

Exit status. The process exits non-zero on any fatal startup failure - a wrong root password, an HTTP port already in use, an unreadable database directory - and exits `0` only when it ended the way it is meant to, with the client closing its stdin. This matters when a supervisor launches it (a systemd unit, a container runtime, or the MCP client itself), since a zero status reads as "finished normally" and suppresses the restart and the alert.

Stdio mode enables the MCP server for the lifetime of the process regardless of the `enabled` flag, since the client started it precisely to speak MCP. Everything else in `config/mcp-config.json` still applies unchanged: permissions, tool profile, per-database overrides, and the `allowedUsers` list, which must admit `root`. Origin validation does not apply, because there is no HTTP request to carry an `Origin` header.

A line holding a top-level JSON array is dispatched as a batch, exactly as over HTTP. A message that produces no response, such as a notification or a batch of them, is answered with no output line at all.

Example client configuration

```
{
  "mcpServers": {
    "arcadedb": {
      "command": "/opt/arcadedb/bin/mcp-stdio.sh",
      "env": {
        "ARCADEDB_SETTINGS": "-Darcadedb.server.rootPassword=arcadedb-password"
      }
    }
  }
}
```

5.6.11. Listing and terminating tool calls

Since v26.10.2 a tool call that works on a database (`query`, `execute_command`, `sample_records`, `vector_search`, `full_text_search`, `hybrid_search`, `upsert_entity` and `upsert_relationship`) is listed while it runs by the HTTP `list queries` server command and by Cypher `SHOW TRANSACTIONS`, with `protocol mcp`:

- A `query` or `execute_command` call is listed as the statement it runs, under the language it names (`cypher` when it names none).
- Any other tool call is listed with language `mcp` and the tool name followed by its arguments, written as in the server log: for example `vector_search {database="research", indexName="Paper[embedding]", queryVector=[384 item(s)], k=20}`. A string argument is cut at 100 characters and an array is shown by its size.

The entry's `clientAddress` is the address of the agent that made the call over HTTP, or `stdio` for the stdio transport, so the work of two agents that share one principal can be told apart.

`terminate query`, `TERMINATE TRANSACTIONS` or the Studio Running Queries tab stop the call: the statements the tool runs are interrupted and the client receives an error result (`isError: true`). **root** sees and stops every call, any other user only their own.

Since v26.11.1 `get_schema` and a read of the `arcadedb://{database}/schema` resource are listed and can be terminated too, although they do not wait for a query admission slot: any tool that takes a `database` argument is listed. A schema read is listed with language `mcp` (for example `get_schema {database="mydb"}`) and a terminate stops it between one type and the next. The server-level tools (`list_databases`, `server_status`, the profiler and settings tools) are not listed.

5.6.12. Security

The MCP server enforces multiple layers of security:

- **Authentication** — All requests require valid ArcadeDB credentials (Basic Auth or Bearer token).
- **User authorization** — Users can only access databases they are authorized for in the ArcadeDB security configuration.
- **MCP permissions** — The MCP configuration controls which operations are allowed (reads, inserts, updates, deletes, schema changes, admin).
- **User allow-list** — The `allowedUsers` setting restricts which users can access the MCP server.
- **Per-database restrictions** — Optional database entries can further restrict operations and users without granting authority above the global policy.
- **Tool profiles** — Hidden tools are removed from discovery and rejected if invoked directly; profiles do not grant permissions.
- **Per-principal profiles** — An individual principal can be narrowed further; the effective profile is the intersection with the global one. See Per-Principal Profiles.
- **Prompt gating** — A prompt is offered, and rendered, only when the tools its text names are visible to the caller and the permissions those tools need are granted. See Prompts.
- **Origin validation** — Cross-origin browser requests are rejected to mitigate DNS rebinding. See [mcp-origin].
- **Semantic analysis** — Queries and commands are analyzed to detect their operation type and enforce the appropriate permission.
- **No scripting languages** — `query` and `execute_command` refuse scripting languages (`js`, `python`, ...): a script cannot be checked without running it. Since v26.11.1.
- **Error text in production mode** — In `production` server mode a tool error raised by the engine (for example a duplicated key refused by `execute_command`, whose message carries the stored key values) answers "The request failed. Check the server log for the details", and the detail is written to the server log. The validation of the tool arguments stays visible. Since v26.10.2; see Error details in production mode.

5.6.13. Using with AI Clients

The ArcadeDB MCP server is compatible with any MCP-compliant client. To connect an AI assistant to ArcadeDB, configure the client with the MCP endpoint URL and authentication credentials, or launch `bin/mcp-stdio.sh` as a child process for clients that prefer the stdio transport.

Example configuration for an MCP client:

```
{
  "mcpServers": {
    "arcadedb": {
      "url": "http://localhost:2480/api/v1/mcp",
      "headers": {
        "Authorization": "Basic cm9vdDphcmNhZGVkYi1wYXNzd29yZA=="
```

```
      }
    }
  }
}
```

Once connected, the AI client automatically discovers the available tools via `tools/list` and can use them to explore databases, query data, and execute commands within the configured permissions. A client that supports the other two surfaces should also read `arcadedb://{database}/schema` from Resources at session start, which supplies the schema without spending a tool call, and offer the Prompts its profile exposes.

Claude Code Plugin

Since v26.11.1 the ArcadeDB repository is also a Claude Code plugin marketplace. The `arcadedb` plugin connects Claude Code to your own ArcadeDB server and adds a skill that teaches the assistant how to work with ArcadeDB.

1. In Studio, create an API token (**Security** > **API Tokens**) with only the permissions the assistant needs.
2. In Studio, enable MCP (**Server** > **MCP**) and add `apitoken:<token name>` to the allowed users.
3. Install the plugin:

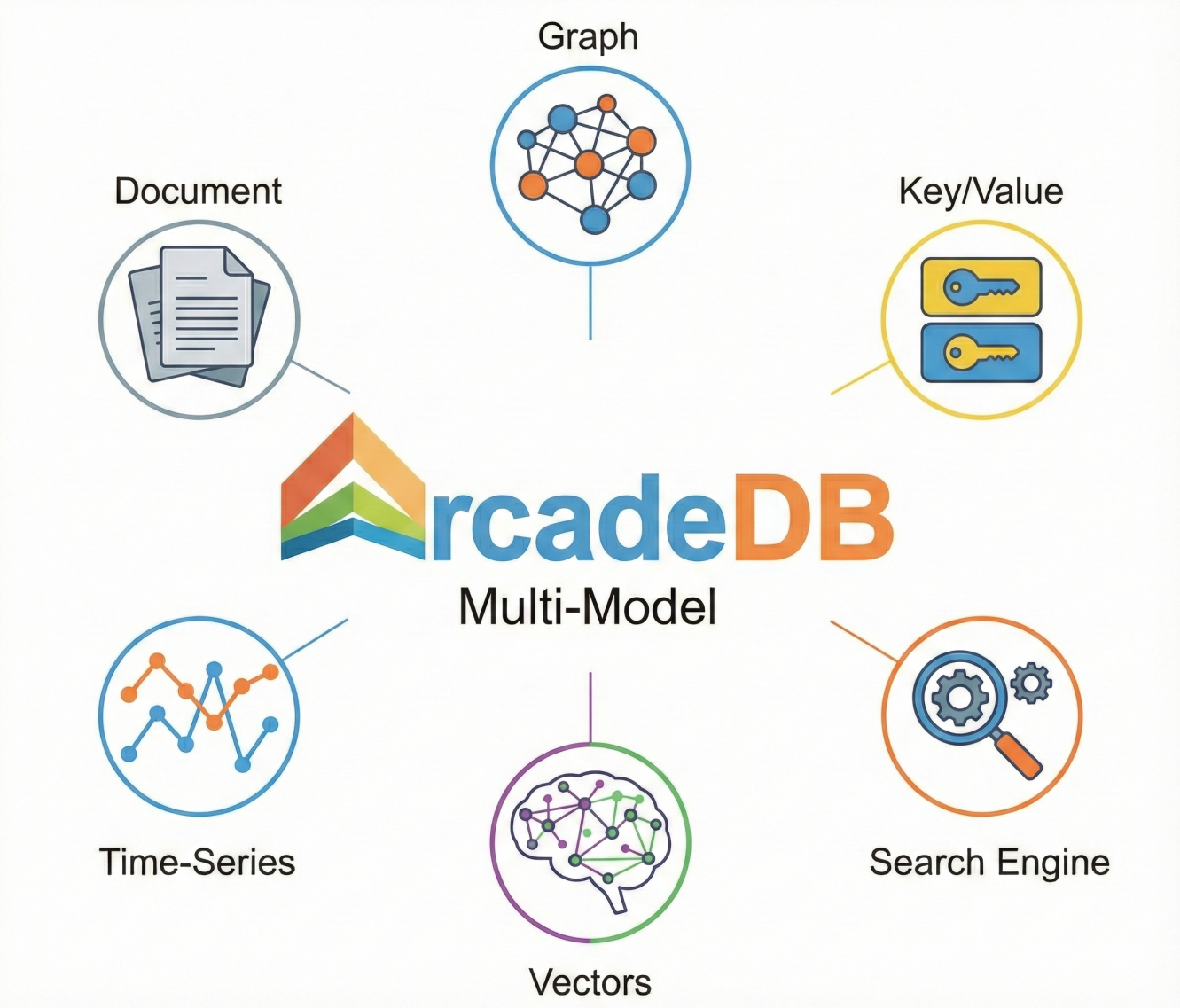
```
claude plugin marketplace add ArcadeData/arcadedb --sparse .claude-plugin claude-plugin
claude plugin install arcadedb@arcadedb
```

Claude Code asks for the server URL (for example `http://localhost:2480`, without `/api/v1/mcp`) and the API token, which it keeps in the operating system's credential store. Use `https://` for any server that is not on your own machine, because the token travels in an HTTP header.

Chapter 6. Data Modeling

6.1. Multi-Model Architecture

ArcadeDB is a multi-model database that natively supports documents, graphs, key-value, time series, full-text search, and vector embeddings within a single engine. All models share the same storage layer, the same transaction manager, and the same query infrastructure. This design eliminates the need for polyglot persistence — the practice of running separate specialized databases for different data models — and the operational complexity that comes with it.



ArcadeDB is a true **Multi-Model** DBMS, combining all the features of the supported models into one core. This is not just a thin layer of interfaces over a single underlying engine, as is common in other "multi-model" databases. The ArcadeDB engine itself was built from the ground up to support every model, which is why it can deliver speed and scalability that bolt-on approaches cannot.

6.1.1. Why Multi-Model Matters

Traditional architectures often pair a relational database with a graph database, a document store, a time series backend, and a vector index. Each system brings its own deployment, its own backup

strategy, its own security model, and its own failure modes. Keeping data consistent across these systems requires ETL pipelines, message queues, or change-data-capture infrastructure that adds latency and introduces synchronization bugs.

ArcadeDB collapses this stack into one process. A single ACID transaction can create a document, connect it as a vertex in a graph, and index its embedding for vector similarity — atomically. There is no replication lag between models because there is no replication; the data exists once.

6.1.2. The Latency Problem with Multiple Databases

With polyglot persistence, a single user request often triggers a **chain of sequential calls** across different databases. Each call adds network latency, serialization overhead, and wait time. The total response time is the *sum* of all calls.

The numbers below match the animation:

Polyglot Persistence (Sequential Calls)	**Database**	**Latency**
Your App → MySQL	Relational	~15ms
Wait… → MongoDB	Document	~20ms
Wait… → Neo4j	Graph	~12ms
Total: ~47ms+ (sum of all calls + serialization overhead)		

ArcadeDB (Single Call)	**Engine**	**Latency**
Your App → ArcadeDB (graph + doc + search in one query)	Multi-Model	~5ms
Total: ~5ms (one call, zero chaining, no serialization)		

With a multi-model database, your application makes a **single call** that can query graphs, documents, and full-text search at the same time. No chaining, no waiting, no redundant serialization. The result? Dramatically lower latency and simpler application code.

ArcadeDB natively supports **six data models**: Graph, Document, Key-Value, Full-Text Search, Vector, and Time-Series — all within a single, embeddable database engine.

6.1.3. Supported Data Models

Document Model

The data in this model is stored inside documents. A document is a set of key/value pairs (also referred to as fields or properties), where the key allows access to its value. Values can hold primitive data types, embedded documents, or arrays of other values. Documents are not typically forced to have a schema, which can be advantageous because they remain flexible and easy to modify. Documents are stored in collections, enabling developers to group data as they decide. ArcadeDB uses the concepts of "Types" and "Buckets" as its form of "collections" for grouping documents.

ArcadeDB's document model also adds the concept of a "Relationship" between documents. You can

decide whether to embed documents or link to them directly. When you fetch a document, all the links are automatically resolved by ArcadeDB. This is a major difference compared to other document databases, like MongoDB or CouchDB, where the developer must handle every relationship between documents themselves.

Types can operate in schema-full mode (all properties declared), schema-less mode (no schema constraints), or schema-hybrid mode where some properties are enforced while others remain free-form. Document types support inheritance, allowing a subtype to extend a parent type and inherit its properties.

Relational Model	Document Model	ArcadeDB Document Model
Table	Collection	Type or Bucket
Row	Document	Document
Column	Key/value pair	Document property
Relationship	not available	Relationship

Graph Model

A graph represents a network-like structure consisting of Vertices (also known as Nodes) interconnected by Edges (also known as Arcs). ArcadeDB's graph model is represented by the concept of a property graph, which defines the following:

- **Vertex** - an entity that can be linked with other vertices and has the following mandatory properties:
  - unique identifier
  - set of incoming edges
  - set of outgoing edges
  - label that defines the type of vertex
- **Edge** - an entity that links two vertices and has the following mandatory properties:
  - unique identifier
  - link to an incoming vertex (also known as head)
  - link to an outgoing vertex (also known as tail)
  - label that defines the type of connection/relationship between head and tail vertex

In addition to mandatory properties, each vertex or edge can also hold a set of custom properties, which makes vertices and edges look similar to documents. Edges are first-class citizens stored in their own types — not join tables or adjacency lists bolted onto a relational schema. ArcadeDB supports two kinds of edges: **lightweight edges** for simple relationships that carry no properties, and **regular edges** for relationships that need their own attributes such as weight, timestamp, or label. See Lightweight Edges for when to choose each, their measured cost, and the `LIGHTWEIGHT` and `UNIQUE` type declarations. Graph traversal operates at O(1) complexity per hop through index-free adjacency.

Relational Model	Graph Model	ArcadeDB Graph Model
Table	Vertex and Edge Types	Type
Row	Vertex	Vertex
Column	Vertex and Edge property	Vertex and Edge property
Relationship	Edge	Edge

Key/Value Model

This is the simplest model: everything in the database can be reached by a key, where the values can be simple or complex types. ArcadeDB supports documents and graph elements as values, providing a richer model than typical key/value stores.

For the full description, comparison table, and HTTP usage patterns, see the dedicated Key/Value page.

Search-Engine Model

The search engine model is based on a full-text variant of the LSM-Tree index. To index each word, the necessary tokenization is performed by the Apache Lucene library. A full-text index is created just like any other index in ArcadeDB and participates in the same transaction as the rest of the database. Results are ranked with BM25 by default, and can be fused with vector search in a single query, see BM25 at a glance.

Time-Series Model

ArcadeDB includes a native time-series engine designed for high-throughput ingestion and fast analytical queries over timestamped data. The Time-Series model is integrated directly into the multi-model core — the same database that stores graphs, documents, and key/value pairs can store and query billions of time-stamped samples with specialized columnar compression, SIMD-vectorized aggregation, and automatic lifecycle management.

Key capabilities:

- **Columnar storage** with Gorilla, Delta-of-Delta, Simple-8b, and Dictionary compression — as low as 0.4 bytes per sample
- **Shard-per-core parallelism** with lock-free writes
- **InfluxDB Line Protocol** ingestion for compatibility with Telegraf, Grafana Agent, and hundreds of collection agents
- **Prometheus remote_write / remote_read** protocol for drop-in Prometheus backend usage
- **PromQL query language** with native parser and HTTP-compatible API endpoints
- **SQL analytical functions** — `ts.timeBucket`, `ts.rate`, `ts.percentile`, `ts.interpolate`, and more
- **Continuous aggregates**, **retention policies**, and **downsampling tiers** for automatic data lifecycle
- **Grafana integration** via DataFrame-compatible endpoints

- **Studio TimeSeries Explorer** with query, schema inspection, ingestion docs, and PromQL tabs

For the full reference, including DDL syntax, ingestion methods, SQL functions, PromQL support, and Grafana integration, see Time Series.

Vector Model

ArcadeDB provides a high-performance vector indexing solution using the LSMVectorIndex, which is built on the JVector library. It implements both the hierarchical navigable small world (HNSW) algorithm and the Vamana (DiskANN) algorithm for efficient approximate nearest neighbor (ANN) search on multi-dimensional vector data. Vector indexes participate in ACID transactions, persist on disk with automatic compaction, and integrate directly into SQL queries through `vector.neighbors()`.

For SQL and Java examples, supported similarity functions, configuration parameters, quantization, and tuning advice, see the dedicated Vector page.

6.1.4. Unified Query Layer

All data models are accessible through multiple query languages that operate on the same underlying records:

- **SQL** — The primary query language, extended with graph traversal, time series aggregation, and vector similarity functions
- **Cypher** — Pattern-matching queries for graph traversal
- **Gremlin** — Imperative graph traversal language (Apache TinkerPop)
- **GraphQL** — Schema-driven queries over HTTP
- **MongoDB Query Language** — Document-oriented queries for migration compatibility
- **Redis Protocol** — Key-value operations over the Redis wire protocol

Because all languages read from and write to the same storage, a record created through SQL is immediately visible to a Gremlin traversal, and an edge created through Cypher can be queried with GraphQL.

6.1.5. Type Inheritance Across Models

ArcadeDB types follow an object-oriented inheritance model. A type can extend another type, inheriting all of its properties, indexes, and constraints. This applies uniformly to document types, vertex types, and edge types.

For example, a `Person` vertex type can serve as the parent for both `Employee` and `Customer` types. A query against `Person` automatically includes records from all subtypes, while a query against `Employee` returns only employees. This polymorphic behavior works identically regardless of which query language is used.

6.1.6. Schema Flexibility

ArcadeDB supports three schema modes that can be mixed within the same database:

- **Schema-full** — Every property is declared with a type and optional constraints. Attempts to store undeclared properties are rejected.
- **Schema-less** — No schema is defined. Records accept arbitrary properties with no validation.
- **Schema-hybrid** — Some properties are declared and enforced, while additional properties can be added freely.

This flexibility allows different types in the same database to use different schema strategies. A `Transaction` type might enforce strict schema-full validation, while a `Metadata` type might operate schema-less to accommodate evolving requirements.

6.1.7. Cross-Model Queries

The most powerful consequence of the multi-model architecture is the ability to combine models in a single query. A SQL statement can traverse graph edges, filter on document properties, aggregate time series data, and rank results by vector similarity — all in one pass, within one transaction.

```
SELECT FROM (
  MATCH {type: User, as: u}-Follows->{type: User, as: f}
  RETURN f
) WHERE vector.cosineSimilarity(f.embedding, ?) > 0.8
```

This query traverses the `Follows` graph to find users connected to a starting set, then filters the results to only those whose embedding vectors are semantically similar to a target. No ETL, no cross-database joins, no eventual consistency — the graph traversal and the vector comparison execute against the same records in the same transaction.

6.2. Graph Database

ArcadeDB is a native graph database; and in this section we explain what this means and how this relates to applications.

6.2.1. Graph Components

Essentially a graph is tuple, or pair of sets, of vertices (aka nodes) and edges (aka arcs), whereas the set of vertices contains an (indexed) set of objects, and the set of edges contains (at least) pairs specifying a respective edge's endpoints.

A particular type of graph is the directed graph, which is characterized by oriented edges, meaning each edge's pair is ordered. A simple example of a directed graph is given by:

with:

- Vertices: V = {v_1, v_2}

- Edges: E = {e_(1,2)}

6.2.2. Graph Database Types

There are two prevalent data models for graph databases (both usually directed):

1. **Triple store (RDF graph)**: data is represented as subject–predicate–object triples; all facts are uniform triples and semantics are standardized (RDF, RDFS/OWL, SPARQL).
2. **Property Graph**: vertices and edges carry labels (types) and arbitrary key/value properties directly attached to them.

ArcadeDB is a property graph database: vertices and edges have labels and can store arbitrary key/value properties, supporting compact storage, fast traversals, and flexible schema evolution within its multi-model, document-oriented engine.

Practically, a vertex or edge consists of an identifier (RID), a label (type), and properties (document), plus ordered endpoint references. The latter are here vertex properties of incoming and outgoing edges, instead of edge properties of ordered vertex pairs, for technical reasons.

6.2.3. Why (Property) Graph Databases?

- The modeled domain is already a network.
- Fast traversal of relations instead of costly joins in relational databases.
- Naturally annotated edges instead of inconvenient reification in RDF graphs.

6.2.4. See Also

- Super-Nodes (Hot Vertices): how ArcadeDB handles vertices with very high or fast-growing edge counts
  - Recommendation Engine — Use case combining graph traversals with collaborative filtering
  - Fraud Detection — Use case leveraging graph patterns to detect fraudulent activity
  - Gremlin API — Tinkerpop Gremlin query language for graph traversals
  - Cypher — OpenCypher query language for pattern matching on graphs

6.3. Time Series

ArcadeDB includes a native Time Series engine designed for high-throughput ingestion and fast analytical queries over timestamped data. Unlike bolt-on solutions, the Time Series model is integrated directly into the multi-model core — the same database that stores graphs, documents, and key/value pairs can store and query billions of time-stamped samples with specialized columnar compression, SIMD-vectorized aggregation, and automatic lifecycle management.

Key capabilities:

- **Columnar storage** with Gorilla (float), Delta-of-Delta (timestamp), Simple-8b (integer), and Dictionary (tag) compression — 0.4 to 1.4 bytes per sample

- **Shard-per-core parallelism** with lock-free writes
- **Block-level aggregation statistics** for zero-decompression fast-path queries
- **InfluxDB Line Protocol** ingestion for compatibility with Telegraf, Grafana Agent, and hundreds of collection agents
- **Prometheus remote_write / remote_read** protocol for drop-in Prometheus backend usage
- **PromQL query language** — native parser and evaluator with HTTP-compatible API endpoints
- **SQL analytical functions** — `ts.timeBucket`, `ts.rate`, `ts.percentile`, `ts.interpolate`, window functions, and more
- **Continuous aggregates** with watermark-based incremental refresh
- **Retention policies** and **downsampling tiers** for automatic data lifecycle
- **Grafana integration** via DataFrame-compatible endpoints (works with the Infinity datasource plugin)
- **Studio TimeSeries Explorer** with query, schema inspection, ingestion docs, and PromQL tabs

6.3.1. Creating a TimeSeries Type

Use `CREATE TIMESERIES TYPE` to define a new time series type. Every type requires a `TIMESTAMP` column, zero or more `TAGS` (low-cardinality indexed dimensions), and one or more `FIELDS` (high-cardinality measurement values).

```
CREATE TIMESERIES TYPE SensorReading
  TIMESTAMP ts PRECISION MILLISECOND
  TAGS (sensor_id STRING, location STRING)
  FIELDS (
    temperature DOUBLE,
    humidity DOUBLE,
    pressure DOUBLE
  )
  SHARDS 8
  RETENTION 90 DAYS
  COMPACTION_INTERVAL 1 HOURS
```

Minimal syntax (defaults for everything optional):

```
CREATE TIMESERIES TYPE SensorReading
  TIMESTAMP ts
  TAGS (sensor_id STRING)
  FIELDS (temperature DOUBLE)
```

Option	Default	Description
`TIMESTAMP <name>`	*(required)*	Name of the timestamp column

Option	Default	Description
`PRECISION`	`MILLISECOND`	Timestamp unit. Only `MILLISECOND` is accepted: timestamps are stored, ingested, retained and queried in milliseconds (see the note below)
`TAGS (…)`	*(none)*	Comma-separated `name TYPE` pairs for low-cardinality dimensions
`FIELDS (…)`	*(required)*	Comma-separated `name TYPE` pairs for measurement values
`CODEC <codec>`	derived from the type	Per-column compression codec, written after a tag or field type (`temperature DOUBLE CODEC GORILLA_XOR`) or after the timestamp. By default the timestamp uses `DELTA_OF_DELTA`, tags `DICTIONARY`, `DOUBLE`/`FLOAT` fields `GORILLA_XOR`, integer, boolean and date/time fields `SIMPLE8B`, and other fields `DICTIONARY`. *(Since v26.10.2)* a codec the sealed store cannot encode or read back unchanged is refused at creation: `NONE`, any timestamp codec but `DELTA_OF_DELTA`, `DELTA_OF_DELTA` on another column, and a numeric codec other than the column type's default (`SIMPLE8B` on a `DOUBLE`, `GORILLA_XOR` on a `LONG` or a text column). `DICTIONARY` is accepted on any tag or field. Earlier versions accepted such a codec, after which compaction of the type failed or the values read back changed. Types already created keep loading
`SHARDS`	`arcadedb.asyncWorkerThreads` (available cores minus 1)	Number of shards for parallel writes
`RETENTION`	*(none)*	Automatic deletion of data older than the specified duration (e.g., `90 DAYS`, `1 HOURS`)
`COMPACTION_INTERVAL`	*(none)*	Cuts sealed blocks at boundaries of this interval. A block that lies inside one query bucket is answered from its statistics without decompressing it, so set it to the bucket width of your most frequent aggregation (see Choosing the layout). Without it blocks are cut by size only and a block spanning several buckets is decompressed
`IF NOT EXISTS`	*(none)*	Skip creation if the type already exists, instead of failing. The statement returns one row either way, carrying a `created` flag: see the note on `IF NOT EXISTS` results

Since v26.10.2 `PRECISION` accepts only `MILLISECOND`, and `SECOND`, `MICROSECOND` and `NANOSECOND` are refused. Earlier versions accepted all four and honoured none of them: every ingestion endpoint (line protocol, Prometheus remote write, gRPC) converts its timestamps to milliseconds, and retention, downsampling and PromQL all count in milliseconds. On a `PRECISION SECOND` type fed with seconds, the first retention pass deleted every sample; on a `NANOSECOND` or `MICROSECOND` type fed

with values in that unit, nothing ever expired and PromQL answered nothing.

A type declared with another precision by an earlier version keeps loading and keeps its data, but its retention and downsampling are **suspended**, because its stored timestamps may be in either unit. A warning names each such type when the maintenance scheduler first meets it, and `SELECT FROM schema:types` reports `maintenanceSuspended: true` for it. If the type's timestamps are milliseconds (always the case for data written through the ingestion endpoints), confirm it to resume them:

```
ALTER TIMESERIES TYPE SensorReading PRECISION MILLISECOND
```

The statement changes only the declaration: no stored timestamp is rewritten, so `MILLISECOND` is the only unit it accepts. A type whose timestamps really are in another unit must have them converted to milliseconds first, by copying the samples into a new type, before retention is resumed. A JSONL export of such a type restores with its declaration and its suspension unchanged, into an embedded database.

A type stores its columns in the order they are declared, and that order matters: a sample is a positional row, so the column order is what an export records and what a restore reads back.

The `TIMESTAMP`, `TAGS` and `FIELDS` clauses may therefore be written in any order, and `TAGS` and `FIELDS` may each appear more than once. The declaration below creates a type whose columns are `f1`, `ts`, `t1`, `f2`, in that order:

```
CREATE TIMESERIES TYPE Interleaved
  FIELDS (f1 DOUBLE)
  TIMESTAMP ts
  TAGS (t1 STRING)
  FIELDS (f2 DOUBLE)
```

The Java schema builder accepts the same orders and keeps them, so a recipe rendered as SQL - which is what the **remote** schema builder does - creates the same type as the same recipe run against an embedded database. A JSONL export therefore restores into a remote database exactly as it does into an embedded one, whatever order its columns are in.

A statement written by an earlier version, with one clause of each kind in the order timestamp, tags, fields, parses and means exactly what it did.

6.3.2. Altering a TimeSeries Type

Add downsampling policies to automatically reduce resolution of old data:

```
ALTER TIMESERIES TYPE SensorReading
  ADD DOWNSAMPLING POLICY
```

```
    AFTER 7 DAYS GRANULARITY 1 MINUTES
    AFTER 30 DAYS GRANULARITY 1 HOURS
```

Remove all downsampling policies:

```
ALTER TIMESERIES TYPE SensorReading DROP DOWNSAMPLING POLICY
```

Confirm that the timestamps of a type declared with another precision by an earlier version are milliseconds, which resumes its retention and downsampling (see the precision note above):

```
ALTER TIMESERIES TYPE SensorReading PRECISION MILLISECOND
```

A TIMESERIES type stores only its declared `TIMESTAMP`, `TAGS`, and `FIELDS` columns. It cannot be given a `SUPERTYPE`, and it cannot be used as a `SUPERTYPE` for another type: any inherited property would be silently dropped by every write, so both directions are rejected. For the same reason, `CREATE PROPERTY` on a TIMESERIES type is also rejected unless the name matches an already-declared column.

Column Constraints

(Since v26.11.1) `ALTER PROPERTY` declares `MANDATORY`, `NOTNULL`, `MIN`, `MAX` and `REGEXP` on a column of a TIMESERIES type as on any property, and every write path enforces them: SQL `INSERT`, the InfluxDB line protocol, gRPC, Prometheus remote write and the JSONL import. The value checked is the one the column stores, read the way `SELECT` returns it:

```
ALTER PROPERTY SensorReading.temperature MIN -50
ALTER PROPERTY SensorReading.temperature MAX 150
ALTER PROPERTY SensorReading.sensor_id REGEXP 'sensor-[A-Z]'
ALTER PROPERTY SensorReading.sensor_id MANDATORY true
```

- `MIN` / `MAX` bound the value of a numeric column, the number of characters of a `STRING` column, and the instant of a `DATE` or `DATETIME` column, compared in the column's own precision (a `DATETIME_NANOS` column to the nanosecond);
- `REGEXP` is matched against the text of the value as `SELECT` returns it, a date as the date and not as its epoch count;
- `MANDATORY` and `NOTNULL` both require a value. A sample has no way to keep a column set to null apart from one left out, so a `MANDATORY` column refuses a null too, and a `NaN` in a `FLOAT` or `DOUBLE` column, which is how such a column stores no value;
- a `STRING` column, tag or field, stores a sample written without it as the empty text and returns it so, and that empty text is what `MIN` and `REGEXP` check: on a column declared `MIN 2` or `REGEXP '[a-z]+'` a sample that leaves the column out is refused, whether the constraint was declared before or after the sample. A numeric or date column keeps no value as no value, and no `MIN`, `MAX` or `REGEXP` applies to it.

A SQL `INSERT` that violates a constraint fails with a `ValidationException`, as for a document. Unlike a document type, a multi-row `INSERT` (`CONTENT [⋯]`, multi-row `VALUES`, `FROM SELECT`) refused at one row keeps the rows appended before it, because each sample is appended in its own shard transaction (see *Appends and Transactions*). On the line protocol, gRPC and Prometheus remote write the sample that violates it is dropped alone and named in `unreadableValues` (`<measurement>.<column> (tag|field|timestamp): <reason>`, the reason naming the constraint and never the value); the JSONL import applies its row-error policy. As for a document type, a constraint is refused when a sample already stored violates it (`Cannot set MAX 10: the sample at timestamp ⋯ violates it`). A sample cannot be updated, but it can be deleted by its time and its tags (see *Deleting Samples*), so the remedy is to delete the sample the message names and declare the constraint again:

```
DELETE FROM SensorReading WHERE ts = 1708430400000 AND sensor_id = 'sensor-A'
ALTER PROPERTY SensorReading.temperature MAX 150
```

Or, to keep the old type as it is, a new TIMESERIES type that declares the constraint, filled with the conforming samples:

```
CREATE TIMESERIES TYPE SensorReading2 TIMESTAMP ts TAGS (sensor_id STRING) FIELDS
(temperature DOUBLE, humidity DOUBLE)
ALTER PROPERTY SensorReading2.temperature MAX 150
INSERT INTO SensorReading2 FROM SELECT ts, sensor_id, temperature, humidity FROM
SensorReading WHERE temperature <= 150
```

A stored `STRING` column reads an absent value back as the empty text, so `MANDATORY`/`NOTNULL` cannot be checked against the samples already there, only against the new ones.

`DEFAULT` and `HIDDEN` are refused on a TIMESERIES column: no write path fills in an absent column and every read returns all of them, so either would be stored without any effect. `READONLY` is accepted, a sample being immutable anyway.

Before v26.11.1 these constraints were accepted and ignored by every write path, and `DEFAULT` and `HIDDEN` were stored without effect. A database written by an earlier version still opens with such a `DEFAULT` or `HIDDEN` (the schema load logs a warning naming the remedy, `ALTER PROPERTY ⋯ DEFAULT null` or `HIDDEN false`), and a `MIN`, `MAX`, `REGEXP`, `MANDATORY` or `NOTNULL` it carries is enforced from now on, on new samples only.

6.3.3. Compacting a TimeSeries Type

New samples land in each shard's mutable bucket, and the maintenance scheduler seals them into compressed blocks every 60 seconds. To seal them now, for example right after a bulk load or before measuring a query on the settled layout, run:

```
COMPACT TIMESERIES TYPE SensorReading
```

It runs the same compaction the scheduler runs, requires the `updateSchema` permission like `COMPACT INDEX`, and under HA runs on the leader. The result carries `mutableSamplesBefore` and `mutableSamples`, the samples still in the mutable buckets when it finished: rows appended while it ran stay there until the next pass.

The scheduler (and `COMPACT TIMESERIES TYPE`) also merges the small blocks a slow feed leaves behind into full-size ones, so the number of blocks follows the data volume and not the time. A read that is scanning the old blocks at the very moment they are merged is refused with a retryable error (HTTP `503`) instead of returning a partial answer: run it again. A merge rewrites the sealed file, so it needs free disk space of about the size of that file.

Since v26.10.2. Compaction, merge, retention and downsampling all rewrite a shard's sealed file, so on one shard they run one at a time. A `COMPACT TIMESERIES TYPE` issued while a retention or downsampling pass is running waits for it (and the other way round), instead of the two overwriting each other's work. Before, a retention pass that overlapped a compaction could be silently undone the next time the database was opened. Likewise, a rewrite that failed half-way no longer leaves a file behind that the next retention pass could resurrect deleted blocks from.

Reads do not need a compaction to be fast. A query for one tag, such as the newest reading of one host, skips every mutable page that holds no row of that tag without reading its rows, whether or not the tail has been sealed yet.

6.3.4. Dropping a TimeSeries Type

```
DROP TIMESERIES TYPE SensorReading
DROP TIMESERIES TYPE SensorReading IF EXISTS
```

6.3.5. Deleting Samples

A sample is never updated once written: `UPDATE` on a TIMESERIES type is refused (`Cannot UPDATE the samples of the TIMESERIES type 'SensorReading': a sample cannot be updated once written …`). To change a sample, delete it and insert it again. Samples leave a type in four ways:

- `DELETE FROM` removes the samples in a time range, optionally of some series only;
- `TRUNCATE TYPE` removes all of them and keeps the type;
- the `RETENTION` of the type removes the samples older than it;
- `DROP TIMESERIES TYPE` removes all of them, with the type.

```
-- every sample before a date
DELETE FROM SensorReading WHERE ts < '2024-01-01 00:00:00'

-- one series, in a time range
DELETE FROM SensorReading WHERE sensor_id = 'sensor-A' AND ts >= '2024-03-01 00:00:00'
AND ts < '2024-03-02 00:00:00'

-- one sample
DELETE FROM SensorReading WHERE ts = 1708430400000 AND sensor_id = 'sensor-A'
```

```
-- every sample, keeping the type
TRUNCATE TYPE SensorReading
```

`DELETE` answers `{count: n, transactional: false}`, the number of samples it removed and the reminder that a `ROLLBACK` will not put them back, and `TRUNCATE TYPE` the same two properties beside its usual result. The `WHERE` names the samples exactly, so it is a time range on the timestamp column (`<`, `<=`, `>`, `>=`, `=`, `BETWEEN`) and equalities on `TAG` columns, joined by `AND`. A predicate on a `FIELD` column, an `OR`, a `LIMIT`, a `BATCH` and `RETURN BEFORE` are refused with an error and delete nothing, rather than delete more than the statement says. `EXPLAIN` shows the range and the tags the statement takes. Deleting what is not there deletes nothing and rewrites nothing.

A sealed block that lies wholly inside the range is dropped without being read, so deleting a stretch of old data costs about what a retention pass costs. A block the range only crosses is rebuilt from the samples that stay, and so are the mutable pages of the shard; the statistics that answer `count`, `sum`, `min` and `max` without reading a block are recomputed with it.

Every statement that removes at least one sealed sample rewrites the sealed file of each shard it changes, as retention does, so its cost is about the size of that file whatever the range: many small deletes cost more than one statement that names the same samples. Under HA the leader ships the whole sealed file of each shard it changed, as it does for retention. A range that misses the mutable pages is decided from their header, with no page read; any other delete reads the mutable pages of the shard into memory once, as a compaction does, so it weighs what a compaction of the same pages weighs.

A statement that fails part way says so: the shards before the one that failed are already deleted, and running the statement again deletes the rest. Under HA the leader remembers, in memory, which shards it rewrote without publishing, and ships them with the next delete. If the leader restarts or loses the leadership between the failed statement and its rerun, the followers can keep samples the old leader deleted: `POST /api/v1/cluster/verify/{database}` compares the sealed files of the nodes and reports the difference, and a resync of the node that differs repairs it.

A delete is not part of the transaction it runs in, like the append of an `INSERT` (see *Appends and Transactions*). The samples are gone, durably, when the statement returns, and a `ROLLBACK` does not bring them back. The statement needs the `deleteRecord` permission on the type.

Readers are held off a shard while its files are rewritten, as they are during a retention pass. A long read that was scanning blocks the delete removed fails with a retryable error (HTTP `503`) instead of returning an answer that mixes the old blocks and the new: run it again. Under HA the delete runs on the leader and ships the rewritten pages and sealed blocks of every shard it changed in one unit, as a compaction does; while another maintenance pass holds the replication session it is retried for a few seconds and then refused with a retryable error that says nothing was deleted. A statement that failed half way can be run again (a crash between the two layers of a shard, the sealed blocks and the mutable pages, leaves the same state, and the count that the interrupted statement would have returned is lost); under HA the shards it got through are published before the failure is reported, so the followers hold what the leader holds. The mutable pages are compacted in place, row by row, whatever their number, so a delete holds no more memory than one row. A `DELETE` with

a tag filter decodes every sealed block that overlaps its range, one block at a time, and the shard stays held, as for a retention pass, for as long as that takes. Under HA the sealed file of each shard a delete changed is shipped to the followers whole, as a retention pass ships it, so the cost of a small delete on a shard with a very large sealed file is the size of that file.

A delete (`DELETE FROM` and `TRUNCATE TYPE`) marks the time range it removed samples from on every continuous aggregate over the type, and the aggregates are refreshed when the statement ends (when the transaction commits, if one is open), so the buckets they hold stop reflecting the samples that were deleted. *(Since v26.11.1, issue #10428.)* Before, the buckets behind the watermark kept the deleted samples for good. The `RETENTION` of the type does not: an aggregate outlives the raw samples it summarized. With no transaction open the refresh runs before the statement returns, so the time of a `DELETE FROM` includes the recomputation of the buckets it reached; inside a transaction it runs when the transaction ends, whether it commits or rolls back: the samples stay deleted either way, because they are not part of it, so the refresh they are owed is not conditional on the commit either. *(Since v26.11.1, issue #10687: before, a `ROLLBACK` dropped the refresh and the aggregate kept the deleted samples until something else refreshed it.)* A refresh that fails is logged and does not fail the statement: the range stays recorded, and the next refresh (the next `INSERT` triggers one, or `REFRESH CONTINUOUS AGGREGATE`) recomputes it.

A `DELETE` or `UPDATE` whose source is a subquery writes to the records that subquery returns, or to the record a projected `@rid` names (`DELETE FROM (SELECT @rid FROM Person WHERE …)`); any other row, such as a sample or a projection (`DELETE FROM (SELECT name FROM Person)`), is refused, since there is no record to write to.

Before v26.11.1 `DELETE FROM` a TIMESERIES type reported the matching samples as deleted and kept them, `UPDATE` reported `count: 0`, and `TRUNCATE TYPE` reported success: none of them removed or changed a sample. From v26.11.1 `UPDATE` is refused, and `DELETE FROM` (by time range and tags) and `TRUNCATE TYPE` remove the samples.

6.3.6. Ingesting Data

There are four ways to ingest data into a TimeSeries type, listed from fastest to most convenient.

InfluxDB Line Protocol (Recommended for High Throughput)

The fastest remote ingestion path. ArcadeDB exposes an InfluxDB Line Protocol-compatible HTTP endpoint that skips SQL parsing entirely.

```
POST /api/v1/ts/{database}/write?precision=ns
Content-Type: text/plain

SensorReading,sensor_id=sensor-A,location=building-1
temperature=22.5,humidity=65.0,pressure=1013.25 1708430400000000000
SensorReading,sensor_id=sensor-B,location=building-2 temperature=19.1,humidity=70.0
```

```
1708430400000000000
```

Line Protocol format: `<measurement>[,<tag>=<value>⋯] <field>=<value>[,⋯] [<timestamp>]`

A malformed line is not ingested; the rest of the payload is. When any line was malformed the endpoint answers `400` with a partial-write body: `written` and `dropped` counts (a malformed line counts as one dropped sample) and `malformedLines`, the 1-based numbers of the rejected lines in the payload (the first 100). A field with no value (`temp`, `temp=` or a line cut short after a field key, such as `weather temp=21.5,hum`) makes the whole line malformed. So does a `,` separator with nothing after it, in the tags or in the fields (`weather temp=21.5,` or `weather,city=rome, temp=1`).

A line cut right after a `,` separator is malformed since v26.10.2.

(Since v26.10.2) A tag with an empty value (`cpu,host= value=1`) makes the line malformed too, as InfluxDB rejects it with "missing tag value"; to write a sample without a tag, leave the tag out. Earlier versions stored it as `host=""`. The gRPC write RPCs refuse an empty-string tag value with `INVALID_ARGUMENT` (an unset value still means the tag is absent), and the Java remote clients refuse it before sending.

(Since v26.10.2) A sample carrying a tag or field key its type does not declare **in that role** (a misspelled tag such as `weather,citty=rome temp=21.5`, or a field sent as a tag) is dropped, because storing it would file the point under a different series than the one sent, with the declared tag left null. The other samples of the request are stored, and the `400` partial-write body names the key in `undeclaredKeys` as `<measurement>.<key> (tag|field)` (the first 100; `dropped` counts every sample), with a message naming the database setting below. The OpenAPI `TimeSeriesWriteError` schema carries the new property, and the gRPC `TimeSeriesWriteSummary` reports the same list as `undeclared_keys` (field 8) on the unary and streaming write RPCs. A sample that simply omits a declared key is still accepted. Earlier versions stored the point without the undeclared key and answered `204`.

The database setting `arcadedb.timeSeriesUndeclaredKeys` decides this for every time-series write: line protocol, gRPC, Prometheus remote write and SQL `INSERT`. `reject` (the default) drops and reports the sample as above (a failed statement for `INSERT`); `ignore` stores the sample with the undeclared keys discarded, which is what a schema-less producer such as Telegraf, sending extra tags the type was never meant to hold, needs:

```
ALTER DATABASE `arcadedb.timeSeriesUndeclaredKeys` 'ignore'
```

A value its column cannot store drops only the sample that carries it, which the `400` partial-write body names in `unreadableValues` as `<measurement>.<column> (tag|field): <reason>` (the reason names the column and its type, never the value; `dropped` is the count of samples, the list holds one entry per distinct reason and the first 100). The gRPC `TimeSeriesWriteSummary` carries the same list as `unreadable_values`. *(Since v26.10.2)* the line protocol and gRPC read a value as `INSERT` does, instead of failing the whole request with a server error:

- a `DATE` or `DATETIME` column takes a number as a count in the column's own unit, digits in quotes as the same count (so `"20230101"` is the count 20230101, not the date, as for a tag filter), and any other quoted date or datetime as the instant it names (`seen="2023-11-14 22:15:23"`). A number is

therefore not read as `INSERT` reads it (epoch milliseconds whatever the precision): see the note under *SQL INSERT* about the epoch counts these surfaces exchange;

- a numeric column takes a number its type can hold and numeric text; a number out of range (`300` in a `BYTE` column, a finite `1e300` in a `FLOAT`) is refused rather than truncated, while `NaN` and the infinities stay values of a floating point column;
- a `BOOLEAN` column takes a boolean and the text `true` or `false` in any case; anything else is refused, where before any text was stored as `false`;
- a `STRING` column takes a string of at most 256 UTF-8 bytes.

Precision parameter: `ns` (nanoseconds, default), `us` or `u` (microseconds), `ms` (milliseconds), `s` (seconds), in any case. An absent or empty `precision` means nanoseconds. *(Since v26.12.1)* any other value (`sec`, `seconds`, `h`, a typo) is refused with `400` and an `error` naming the accepted values, and nothing in the request is written, as in InfluxDB. Earlier versions answered `204` and read the timestamps as nanoseconds, so a timestamp in seconds was stored in 1970. A `ns` or `us` timestamp is stored at the millisecond that contains it, so `-1` ns (1969-12-31T23:59:59.999999999Z) is `-1` ms; the gRPC `TimeSeriesWrite` converts its points the same way. *(Since v26.12.1)* earlier versions rounded a timestamp before 1970 toward zero, so `-1` ns was stored as `0` ms and fell in the next time bucket.

Example with `curl`:

```
curl -X POST "http://localhost:2480/api/v1/ts/mydb/write?precision=ns" \
  -u root:password \
  -H "Content-Type: text/plain" \
  --data-binary 'SensorReading,sensor_id=sensor-A temperature=22.5 1708430400000000000
SensorReading,sensor_id=sensor-A temperature=22.6 1708430401000000000'
```

Example with Python:

```
import requests

lines = [
    "SensorReading,sensor_id=sensor-A temperature=22.5 1708430400000000000",
    "SensorReading,sensor_id=sensor-A temperature=22.6 1708430401000000000",
]

requests.post(
    "http://localhost:2480/api/v1/ts/mydb/write?precision=ns",
    auth=("root", "password"),
    headers={"Content-Type": "text/plain"},
    data="\n".join(lines),
)
```

If the type does not exist and auto-creation is enabled (`arcadedb.tsAutoCreateType=true`), the schema is inferred from the first line: measurement name becomes the type, tags become TAG columns, fields become FIELD columns with inferred types.

Prometheus Remote Write

ArcadeDB acts as a drop-in Prometheus remote storage backend. Configure Prometheus to write to ArcadeDB:

```
# prometheus.yml
remote_write:
  - url: "http://localhost:2480/api/v1/ts/mydb/prom/write"
    basic_auth:
      username: root
      password: password
```

The endpoint accepts the standard Prometheus `remote_write` Protobuf payload (snappy-compressed). Each time series is mapped to an ArcadeDB TimeSeries type named after the `name` label. Types are auto-created if they do not exist.

(Since v26.10.2) A label with an empty value is the same as an absent label, as in Prometheus: it is stored as absent and `{label=""}` selects it. A series carrying a label that the existing type does not declare as a tag is dropped and the request answers `400` naming the label (the other series of the request are stored); Prometheus does not retry a `400`. Set `arcadedb.timeSeriesUndeclaredKeys=ignore` to discard the extra label instead.

(Since v26.10.2) The endpoint implements remote write 1.0 (`prometheus.WriteRequest`) and answers as the remote-write specification asks of such a receiver:

- a `Content-Type` whose `proto=` parameter names another message (`application/x-protobuf;proto=io.prometheus.write.v2.Request`) is answered `415`, so a remote write 2.0 sender falls back to 1.0;
- a body that decodes to no remote write 1.0 field at all, typically a 2.0 message sent without the `proto=` parameter, is answered `400`. It used to be answered `204`, so the sender discarded the batch as delivered;
- a request carrying only metric metadata, or an empty `WriteRequest`, is still a no-op answered `204`;
- a malformed protobuf body is answered `400` instead of `500`.

(Since v26.12.1) PromQL honours the staleness marker Prometheus writes when a series ends (a target goes down, a pod is replaced): a series whose newest sample in the lookback is the marker is left out of an instant vector, so it no longer shows as `NaN`, turns `sum()` into `NaN` or counts in `count()`, and a range selector does not count the marker as a sample (`count_over_time()`). An ordinary `NaN` written by a client is still a value and is returned as `NaN`, as in Prometheus. SQL and the other `/ts` endpoints read the marker as an absent value, like any `NaN` (see Missing Values below). The marker is kept only in a `DOUBLE` column, which is what remote write creates for `value`: a `FLOAT` column, a `DOUBLE` field with the `DICTIONARY` codec and a downsampled block read it back as an ordinary `NaN`, so there an ended series is still answered as `NaN` until the lookback runs out (issue #12580).

If your collector or proxy sends an `X-Request-Id` header, the request body is taken into account when deciding whether a request is a repeat of a previous one.

Before, on `prom/write` and `prom/read` only, two requests carrying **different** samples under the same `X-Request-Id` were mistaken for a retry: the second was answered with the first one's `204` and its samples were never stored. This affected collectors that stamp one id per connection, per scrape loop or per retry batch rather than per distinct payload. Prometheus' own remote-write client does not send the header and was never affected.

Genuine retries still work as before: re-sending the identical body under the same id is applied only once.

SQL INSERT

Standard ArcadeDB SQL syntax works for TimeSeries types:

```
-- Single row
INSERT INTO SensorReading
  SET ts = date('2026-02-20 10:00:00', 'yyyy-MM-dd HH:mm:ss'),
      sensor_id = 'sensor-A',
      location = 'building-1',
      temperature = 22.5,
      humidity = 65.0

-- Batch insert
INSERT INTO SensorReading
  (ts, sensor_id, location, temperature, humidity)
  VALUES
    (date('2026-02-20 10:00:00', 'yyyy-MM-dd HH:mm:ss'), 'sensor-A', 'building-1', 22
.5, 65.0),
    (date('2026-02-20 10:00:01', 'yyyy-MM-dd HH:mm:ss'), 'sensor-A', 'building-1', 22
.6, 64.8),
    (date('2026-02-20 10:00:02', 'yyyy-MM-dd HH:mm:ss'), 'sensor-B', 'building-2', 19
.1, 70.0)

-- CONTENT syntax
INSERT INTO SensorReading
  CONTENT { "ts": 1771581600000, "sensor_id": "sensor-A", "temperature": 22.5 }

-- One sample per selected row
INSERT INTO SensorReading
  FROM SELECT ts, sensor_id, temperature FROM RawReading
```

(Since v26.11.1) `INSERT … FROM SELECT` appends one sample per selected row, with the same conversion and the same column constraints as `INSERT … SET`. Before, it failed with `Cannot retrieve a bucket for type … because there are no buckets associated` (`Transaction not begun` outside a transaction).

(Since v26.10.2) `INSERT` fails if it sets a property that is not a declared tag or field of the type (for example a misspelled tag), because the sample would otherwise be

filed under a different series (see undeclared keys). Set `arcadedb.timeSeriesUndeclaredKeys=ignore` on the database to discard such properties instead. The same setting governs line protocol, gRPC and Prometheus writes.

(Since v26.10.2) A `DATE`, `DATETIME`, `DATETIME_SECOND`, `DATETIME_MICROS` or `DATETIME_NANOS` tag or field behaves in SQL exactly like a document property of the same type. `INSERT` reads a value as such a property does: text and date/time objects as the instant they name, and a number as epoch milliseconds (epoch days for a `DATE`), whatever the column's precision. `SELECT` returns the value as the date/time object the property would return (a `java.time.LocalDateTime` for the `DATETIME` family by default), so a value read from a row can be written back as it is, and a `WHERE` condition on the column matches the same literal `INSERT` accepted, on a tag (pushed down to the storage engine) and on a field alike. An instant the column's precision cannot represent - for example a nanosecond count written to a `DATETIME_NANOS` column, which `INSERT` reads as milliseconds 53 million years from now - is refused with an error. Before, `SELECT` returned the raw epoch count in the column's own unit, which neither `INSERT` nor a filter read the same way, and such an instant was stored as `2262-04-11T23:47:16.854775807`. The engine API, line protocol, gRPC and HTTP surfaces keep exchanging these columns as epoch counts in the column's unit. This split is intended: SQL treats the column as its document twin, the wire protocols carry the count the column stores. So digits in quotes (`'1700000000000'`) are refused by `INSERT`, as by a `DATETIME` document property, while the line protocol and gRPC store them as that count. *(Since v26.11.1)* `INSERT` refuses a value such a column cannot take with a `CommandExecutionException` naming the column, like any other value it refuses, where it used to fail with a bare `IllegalArgumentException`.

The same literal, per column type and surface (*since v26.11.1*, pinned by a test that runs every cell). `1700000000000` is 2023-11-14 22:13:20 UTC in epoch milliseconds, and `1700000000` the same instant in seconds:

Literal in a column	SQL `INSERT`	Line protocol, gRPC (the wire)
`1700000000000` (a number) in `DATETIME`	the instant 2023-11-14 22:13:20	the same: a count in milliseconds
`1700000000` (a number) in `DATETIME_SECOND`	the count 1700000, the instant 1970-01-20 (a number is milliseconds)	the count 1700000000, the instant 2023-11-14 22:13:20 (a count in seconds)
`1700000000000` (a number) in `DATETIME_MICROS`, `DATETIME_NANOS`	the instant 2023-11-14 22:13:20	the count 1700000000000, the instant 1970-01-20 (microseconds) or 1970-01-01 00:28:20 (nanoseconds)
`1700000000000` (a number) in `DATE`	refused: no date is that many days away	stored: a count of days no date can stand for

Literal in a column	SQL `INSERT`	Line protocol, gRPC (the wire)
`'1700000000000'`, `'0'` (digits in quotes) in a `DATETIME` family column	refused: it is no date	stored as that count; a line protocol tag has no type, so a `DATETIME` tag can only arrive as text of digits
`'19675'` (digits in quotes) in `DATE`	the date 2023-11-14, read as days	the same
`'2023-11-14 22:13:20'`	the instant, in the column's precision	the same instant
`'not a date'`, `true`	refused	refused

An operator who writes one measurement from both places should write it the same way from both: an ISO date in text, or a number in the unit the surface reads (milliseconds in SQL, the column's unit on the wire). The refusal of SQL names the rule.

The timestamp column of a TimeSeries type is stored as a number, so `INSERT` does not accept an ISO-8601 string for it (`Cannot convert type String to LONG`): write an epoch value (as in the `CONTENT` example, milliseconds for a type with `PRECISION MILLISECOND`) or build it with the `date()` function, which interprets the text in the server's time zone. ISO-8601 strings are accepted in `WHERE` conditions on the timestamp column.

Java Embedded API

The fastest path — bypasses all protocol and SQL overhead:

```
TimeSeriesEngine engine = database.getSchema()
    .getTimeSeriesType("SensorReading").getEngine();

long[] timestamps = { 1708430400000000000L, 1708430401000000000L };
String[] sensorIds = { "sensor-A", "sensor-A" };
double[] temperatures = { 22.5, 22.6 };

database.transaction(() -> {
  engine.appendSamples(timestamps,
      new Object[] { sensorIds, temperatures });
});
```

`TimeSeriesEngine.appendSamples()` and `appendBatch()` are the engine's own API: they store the values as given, and are for data the application has already checked. `database.async().appendSamples()` is the checked one. *(Since v26.11.1)* It converts every value as `INSERT` does and holds it to the constraints of the type's columns (`MANDATORY`, `NOTNULL`, `MIN`, `MAX`, `REGEXP`) before it returns, on the calling thread: a value no column can store, or a sample a constraint refuses, raises a `ValidationException` that names the column and the reason, and the whole batch is refused, with nothing appended. Before, such a sample was stored (and then could not be removed, or declared against, when the constraint was declared again), or the batch was lost inside the async worker

with `waitCompletion()` returning normally.

The form that takes a `TimeSeriesBatch` (`database.async().appendSamples(type, batch)`) is checked the same way. *(Since v26.11.1)* A `TAG` or `STRING` text of more than 256 UTF-8 bytes, which a row cannot hold, is refused on the calling thread with a `ValidationException` that names the column and the limit, and the whole batch is refused; before, the call returned and the whole batch, the conforming rows included, was lost inside the async worker. The typed setters of a batch (`setLong()`, `setDouble()`, ...) store the primitive they are given. `TimeSeriesBatch.setValue()`, for a value you hold as an `Object`, converts it as `INSERT` does and raises an `IllegalArgumentException` naming the column for a value the column cannot store: a number past the range of an `INTEGER`, `SHORT` or `BYTE` column, text that is not `true` or `false` for a `BOOLEAN` column, text that is not a number for a numeric column, a text over the limit. Before, a `Long` past the `INTEGER` range was stored modulo 2^32, any text for a `BOOLEAN` column was stored as `false`, and numeric text failed with a bare `ClassCastException`.

Appends and Transactions

A TimeSeries append is not part of the transaction that contains it. This is true of every ingestion method above - SQL `INSERT`, the REST write endpoint, the line protocol, Prometheus remote write and the Java embedded API alike:

- the samples are committed as they are appended, so they are durable and visible to every other connection as soon as the append returns, before you commit anything;
- rolling the enclosing transaction back does **not** remove them.

Use a session or a transaction around an append for what it does do - it holds that session's lock and principal and keeps its idle timer from reaping it while you ingest - and not for atomicity with the samples.

The reason is the shard's own append lock. Every append writes the shard's header page, which holds the sample count and the min/max timestamps, so two appends to one shard always want the same page version; serializing them behind a lock the append itself releases at its own commit is what keeps them from conflicting. An append that wrote through the caller's transaction could not be serialized that way, because the commit would belong to the caller and a shard cannot hold that lock until an arbitrary user transaction ends without blocking every other writer of that shard. Both would then stage the same page and one would lose its **whole** transaction to a concurrent-modification error that is not the shard's to retry.

Ingestion Method Comparison

Method	Throughput	Overhead	Best For
Java Embedded API	~0.5-1 us/sample	None (direct)	Embedded applications
InfluxDB Line Protocol	~1-5 us/sample	Text parsing	Remote ingestion, Telegraf
Prometheus Remote Write	~2-10 us/sample	Protobuf + Snappy	Prometheus ecosystems

Method	Throughput	Overhead	Best For
SQL INSERT	~50-100 us/sample	SQL parsing + planning	Ad-hoc inserts, small batches

6.3.7. Querying Time Series Data

SQL Queries

Time series types support standard SQL `SELECT` with `WHERE`, `GROUP BY`, and `ORDER BY`. Time range conditions (`BETWEEN`, `>`, `>=`, `<`, `<=`, `=`) on the timestamp column are pushed down to the storage engine for efficient range scans. A time range on one branch of an `OR` never narrows the other branch, and two equalities on the same tag in one `AND` match nothing when the values differ, exactly as on a document type.

```
-- Basic range query
SELECT ts, sensor_id, temperature, humidity
FROM SensorReading
WHERE ts BETWEEN '2026-02-19' AND '2026-02-20'
  AND sensor_id = 'sensor-A'
ORDER BY ts

-- Aggregation with time bucketing
SELECT ts.timeBucket('1h', ts) AS hour,
       sensor_id,
       avg(temperature) AS avg_temp,
       max(temperature) AS max_temp,
       min(temperature) AS min_temp,
       count(*) AS sample_count
FROM SensorReading
WHERE ts BETWEEN '2026-02-19' AND '2026-02-20'
GROUP BY hour, sensor_id
ORDER BY hour
```

An aggregation that groups by the time bucket, by up to four `TAG` columns, or by both is answered by the storage engine in one pass (`AGGREGATE FROM TIMESERIES` in `EXPLAIN`), the shape of a per-host hourly report. Every projected tag must also be in the `GROUP BY`, and the aggregates are `avg`, `min`, `max`, `sum` and `count(*)` over numeric fields. A query outside that shape (a field in the `GROUP BY`, a projected tag that is not grouped by, `count(field)`, `DISTINCT`) gives the same answer through the generic row-at-a-time plan, which is slower.

An aggregation with no `GROUP BY` at all is answered by the storage engine as well, as one row over the queried range (`AGGREGATE FROM TIMESERIES ⋯ ungrouped` in `EXPLAIN`). Every sealed block lying wholly inside the range is answered from the sample count, minimum, maximum and sum it records, so only the blocks that cross a bound of the range are decompressed, and the cost of the query follows the number of blocks rather than the number of samples. Over a range holding no sample it answers one row, `0` for `count(*)` and `NULL` for every other aggregate, as SQL does.

```
SELECT count(*) AS samples, avg(temperature) AS avg_temp, max(temperature) AS max_temp
FROM SensorReading
WHERE ts >= '2026-02-19' AND ts < '2026-02-20'
```

Range conditions on a numeric `FIELD` are pushed down too: `>`, `>=`, `<`, `<=`, `=` and `BETWEEN` against a literal or a parameter, combined with `AND` (`FIELDS` in `EXPLAIN`). A sealed block whose recorded minimum and maximum rule a condition out is skipped without being read, a block whose whole range satisfies every condition is answered as if there were no condition (from its statistics, in an aggregation), and only the blocks in between are decompressed, with the condition tested on the decoded columns before any row is built. That applies to the aggregations above, so `SELECT count(*) … WHERE ts … AND temperature > 30` is answered by the storage engine, and to a plain row query, which then builds rows only for the samples that pass.

When the storage engine answers the whole `WHERE` of a row query - a time range, tag equalities and the field conditions above, combined with `AND` - the plan has no `FILTER ITEMS` step, and a projection made only of column names (aliases allowed) builds each row once, in the fetch (`PROJECTED` in `EXPLAIN`), instead of building a row of every column and copying it, as for `SELECT sensor_id, ts, temperature FROM SensorReading WHERE ts >= ? AND ts < ? AND temperature > 30`.

The comparisons pushed down are those SQL makes as plain numbers: a `DOUBLE` field against any number, and a `LONG`, `INTEGER`, `SHORT` or `BYTE` field against an integer. A missing measurement never satisfies one, as `NULL > 30` does not. Any other condition - on a `FLOAT` field, `<>`, an `OR`, a comparison between two columns, a function of the field, a decimal against an integer field - gives the same answer through the generic filter.

When upgrading to 26.10.2: an aggregation answered by the storage engine returns the same values and Java types as the generic plan. `min` and `max` return a value of the column's declared type (a date/time object for a `DATE` or `DATETIME` column), `sum` over a `LONG`, `INTEGER`, `SHORT` or `BYTE` field returns an integer that widens to a `BigDecimal` past the `long` range (`sum` over a `FLOAT` field a float), and `avg` is a `Double`, instead of the `Double` earlier versions returned for a query grouped by the time bucket or by tags. An integral column is aggregated exactly, as 64-bit integers with a 128-bit running total, so a value past 2^53 - an epoch-nanosecond instant, a 64-bit id - is no longer rounded.

TimeSeries SQL Functions

ArcadeDB provides a comprehensive set of `ts.*` SQL functions for time series analytics.

Function	Description
`ts.timeBucket(interval, timestamp [, options])`	Truncates a timestamp to the nearest interval boundary for `GROUP BY` aggregation. Intervals: `'1s'`, `'5m'`, `'1h'`, `'1d'`, etc. An interval (or an `offset`) whose milliseconds do not fit a 64-bit long, such as `'30500568905w'`, is refused with a "too large" error naming the largest amount of that unit *(since v26.12.1, issue #11986; earlier versions silently wrapped it to a different bucket width)*. Buckets are counted from the Unix epoch unless `options` moves the grid, see Moving the bucket grid.
`ts.first(value, timestamp)`	Returns the value corresponding to the earliest timestamp in the group.
`ts.last(value, timestamp)`	Returns the value corresponding to the latest timestamp in the group.
`ts.rate(value, timestamp [, counterResetDetection])`	Per-second rate of change. Optional 3rd parameter (`true`) enables Prometheus-style counter reset detection for monotonic counters.
`ts.delta(value, timestamp)`	Difference between the last and first values in the group.
`ts.movingAvg(value, window)`	Trailing moving average: returns a list with one element per input row, in row order, where each element averages the window of `window` rows ending at that row (`window` may also be given as `{window: n}`). A row whose value is `null` keeps its slot and contributes nothing to the windows it belongs to: each window averages only its non-null values, and a window holding no value at all answers `null`, so the list can contain `null` elements. Each window's average depends only on its own members. *(Since v26.10.2.)* Earlier versions dropped a `null` row from the list, which shifted every later average into an earlier row's slot.
`ts.interpolate(value, method [, timestamp])`	Gap filling. Methods: `'zero'`, `'prev'`, `'linear'`, `'none'`. The `'linear'` method requires the timestamp parameter.
`ts.correlate(a, b)`	Pearson correlation coefficient between two series.
`ts.percentile(value, percentile)`	Approximate percentile calculation (0.0-1.0). E.g., `ts.percentile(latency, 0.99)` for p99.
`ts.lag(value, offset, timestamp [, default])`	Window function: returns the value from a previous row.
`ts.lead(value, offset, timestamp [, default])`	Window function: returns the value from a subsequent row.
`ts.rowNumber(timestamp)`	Window function: sequential 1-based row numbering.
`ts.rank(value, timestamp)`	Window function: rank with ties, gaps after ties.

6.3.8. Moving the bucket grid

By default buckets are multiples of the interval counted from the Unix epoch, which was a

Thursday at 00:00 UTC: a `'1w'` bucket starts on Thursday and a `'1d'` bucket starts at 08:00 in UTC+8. The optional third parameter of `ts.timeBucket` moves the grid; use one of:

```
-- weeks start on Monday
SELECT ts.timeBucket('1w', ts, {'origin': '2024-01-01T00:00:00Z'}) AS week, sum(kwh)
FROM Energy GROUP BY week

-- days start at local midnight in UTC+8
SELECT ts.timeBucket('1d', ts, {'offset': '-8h'}) AS day, sum(kwh) FROM Energy GROUP
BY day
SELECT ts.timeBucket('1d', ts, {'timezone': '+08:00'}) AS day, sum(kwh) FROM Energy
GROUP BY day
```

- `origin`: any instant a bucket starts at (a bare instant as third parameter means the same).
- `offset`: a signed duration added to the epoch grid.
- `timezone`: local midnight (local Monday for `'1w'`) of a zone with a fixed offset such as `'+08:00'`. Zones with daylight saving are refused, because their days do not all have the same length.

The same grid is used by the aggregation push-down, the HTTP/gRPC/Grafana `bucketOrigin`, continuous aggregates (write the option in the defining query) and downsampling tiers, which take an `OFFSET` clause:

```
ALTER TIMESERIES TYPE SensorReading ADD DOWNSAMPLING POLICY AFTER 30 DAYS GRANULARITY
1 DAYS OFFSET -8 HOURS
```

Blocks already downsampled are not re-bucketed if you change the offset later.

Examples:

```
-- Rate of change with counter reset detection
SELECT ts.timeBucket('5m', ts) AS window,
       ts.rate(request_count, ts, true) AS requests_per_sec
FROM HttpMetrics
WHERE ts > '2026-02-20T10:00:00Z'
GROUP BY window

-- Percentile calculation
SELECT ts.timeBucket('1h', ts) AS hour,
       ts.percentile(latency_ms, 0.99) AS p99,
       ts.percentile(latency_ms, 0.50) AS median
FROM ServiceMetrics
GROUP BY hour

-- Gap filling with linear interpolation
SELECT ts.timeBucket('1m', ts) AS minute,
       ts.interpolate(temperature, 'linear', ts) AS temp
FROM SensorReading
```

```
WHERE ts BETWEEN '2026-02-20T10:00:00Z' AND '2026-02-20T11:00:00Z'
GROUP BY minute

-- Correlation between two fields
SELECT ts.correlate(temperature, humidity) AS correlation
FROM SensorReading
WHERE ts BETWEEN '2026-02-19' AND '2026-02-20'
```

Dedicated JSON Query Endpoint

A simplified REST endpoint is available for programmatic access and Grafana integration:

```
POST /api/v1/ts/{database}/query
Content-Type: application/json

{
  "type": "SensorReading",
  "from": "2026-02-19T00:00:00Z",
  "to": "2026-02-20T00:00:00Z",
  "columns": ["temperature", "humidity"],
  "tags": { "sensor_id": "sensor-A" },
  "aggregation": "AVG",
  "bucketInterval": "1h"
}
```

For raw (non-aggregated) queries, omit the `aggregation` and `bucketInterval` fields.

The optional `bucketOrigin` (epoch milliseconds, also available on the Grafana endpoint, on gRPC as `bucket_origin_ms` and in the Java client as `TimeSeriesQuery.bucketOrigin()`) moves the bucket grid like the `origin` option of `ts.timeBucket`, see Moving the bucket grid.

`from`, `to`, `bucketInterval` and `maxDataPoints` must be whole numbers. A value with a fractional part is refused with `400` and a message naming the member, rather than being rounded down to something you did not ask for. A bucket width computed by division therefore has to be rounded by the client, deliberately.

(Since v26.10.2) A `DATE` or `DATETIME` tag value given as text in a tag filter (the `tags` object of this endpoint and of the Grafana query endpoint, the `tag` parameter of `latest`, and the gRPC `TimeSeriesQuery` and `TimeSeriesLatest` RPCs) is read in the database's time zone, with its date formats, exactly as the `INSERT` that wrote the tag read it. Earlier versions read it in the JVM's default time zone, so on a server whose JVM zone differed from the database's the filter named another instant and selected nothing. In Java, `TimeSeriesGateway.buildTagFilter()` and `TimeSeriesGateway.andTag()` now take the `Database` as their first argument; code calling the old signatures must pass it.

To retrieve the most recent data point:

```
GET /api/v1/ts/{database}/latest?type=SensorReading&tag=sensor_id:sensor-A
```

Repeat `tag` once per tag column to narrow to a single series; every occurrence must match:

```
GET /api/v1/ts/{database}/latest?type=SensorReading&tag=sensor_id:sensor-
A&tag=region:eu
```

A tag name that is not a `TAG` column of the type is refused with **HTTP 400**, naming it and listing the tags the type actually declares. It used to be ignored, which meant a typo **widened** the query instead of failing it: `{"tags": {"sesnor_id": "sensor-A"}}` returned every row in the range, and `?tag=sesnor_id:sensor-A` the newest sample of any series - neither distinguishable from a filter that legitimately matched everything.

On `latest`, an occurrence that is not in `name:value` form (`?tag=sensor_id`) is refused for the same reason. An empty `?tag=` is still ignored, since that is what an empty form field produces.

This applies to `POST /api/v1/ts/{database}/query`, `GET /api/v1/ts/{database}/latest`, the Grafana query endpoint (reported as that panel's error, so one bad tag does not blank the others) and the gRPC `TimeSeries` RPCs, which answer `INVALID_ARGUMENT`. The PromQL endpoint is unaffected: there an unknown label has a defined meaning under Prometheus' rules and is not a malformed request.

Missing Values

A sample can be absent, which means "no measurement here" rather than a value of zero. You write one either as `NaN` or as `null`, and both are stored the same way on a `DOUBLE` or `FLOAT` column. Aggregations treat it that way:

- `MIN`, `MAX`, `SUM` and `AVG` skip `NaN` samples. Over a window that holds at least one real value they answer from the real values alone: `SUM` is the sum of the real samples and `AVG` divides it by the number of real samples, not by the number of samples. Over a window whose samples are **all** `NaN` they return `NaN`, meaning the window has no measurement to report.
- `count(*)` counts rows, so it counts `NaN` samples too. In SQL, `count(field)` counts only the rows where the field is not null, as on any other type, and returns a `Long`.
- A genuine `Infinity` or `-Infinity` sample is a measurement and is returned as such.
- Downsampling averages the real samples of each bucket; a bucket with no real sample is downsampled to `NaN`.

This is the semantics of SQL's `NULL` under `SUM`, `AVG` and `count(*)`. The PromQL endpoint is the one exception: its `sum`, `avg`, `sum_over_time` and `avg_over_time` propagate `NaN`, because that is what Prometheus does and a PromQL query is expected to answer what Prometheus would.

Over the REST and Grafana endpoints an absent value is serialized as JSON `null`, since JSON has no

NaN literal. A Grafana panel therefore draws a gap for such a bucket rather than a dip to zero. Over SQL it reads back as NULL, both as a projected column and as the result of an aggregate with nothing to aggregate:

```
SELECT value FROM Sensor                          -- an absent sample is NULL
SELECT avg(value) FROM Sensor GROUP BY bucket     -- a bucket with no real sample is
NULL
```

A NaN that arithmetic itself produces over real samples - the sum of +Infinity and -Infinity, for instance - is an undefined total rather than an absence, and is returned as NaN.

(Since v26.10.2) A LONG, INTEGER, SHORT, BYTE, BOOLEAN, DATE or DATETIME column records absence too: a null written to it, or a column a TimeSeriesBatch row leaves unset, reads back as null, and MIN, MAX, SUM, AVG and count(field) skip it as they skip NaN. These types have no value to spare for absence, so a row of the mutable buffer carries a small null bitmap and a sealed block a null bitmap per column. The row layout is fixed when the type is created, so a TimeSeries type created by an earlier version keeps storing such a null as 0 (or false), which counts as a real measurement: recreate the type (for example EXPORT and IMPORT) to record gaps in its integer columns.

An earlier version stored a null measurement as a real 0.0 on every column type. Samples already stored as zero that way stay zero after an upgrade - they are indistinguishable from a real measurement of zero.

Every sealed block records the count of real samples per column, which is what lets AVG be answered from the block statistics without decompressing the block. Blocks written by an earlier version are read as they are: a block whose recorded sum is NaN because one of its samples was NaN is decompressed for every aggregate but COUNT, and the next compaction, downsampling or retention rewrite of the block records both statistics. The sealed store format version moved from 0 to 1 for this.

Each sealed block records an identity of its own, which is what lets a long-running read - a wide query, a PromQL range, an EXPORT DATABASE - find its blocks again if the shard is rewritten underneath it. On a high-availability follower that happens whenever the leader ships a newly compacted store: before this, such a read returned the rows it had reached and stopped there silently, with no error. Blocks written by an earlier version have no identity recorded and one is derived from the block itself, so they behave the same way; the next compaction, downsampling or retention rewrite records it.

The sealed store format version moved to 2 for this, so a store in this format cannot be opened by an earlier version - deliberately, since the alternative is an older version reading it as if it held nothing. In a cluster, keep every node on the same version.

Which Columns Can Be Aggregated

`SUM`, `AVG`, `MIN` and `MAX` read the column's values, so they need a column that is stored as a number. That means a numeric `FIELD`: `DOUBLE`, `FLOAT`, `LONG`, `INTEGER`, `SHORT`, `BYTE`, `BOOLEAN` (as 1 and 0), and the datetime types.

A column that is not stored as a number is refused, and the message names it:

```
{ "error": "Aggregation SUM cannot be applied to column 'host': a TAG column of type
STRING is not stored as a number" }
```

Tags are always stored as text, whatever type they are declared with, so a tag cannot be aggregated even when it is declared `LONG`. The same applies to a `STRING` field and to the timestamp column itself.

`COUNT` is the exception and works on any column. The SQL push-down also behaves differently: rather than refusing the query it simply falls back to the generic aggregation path, so `SELECT … max(host) … GROUP BY` and `count(field) … GROUP BY` (which skips null values) still answer. Only `count(*)` is answered by the push-down.

`BOOLEAN` fields are stored as numbers, so a `SUM` over a boolean field answers the number of `true` samples, before compaction and after it.

When upgrading: a `BOOLEAN` field on a type created by an early version is stored as text rather than as a number, so `SUM`/`AVG` over it is refused. Recreate the type to aggregate such a field. `COUNT` is unaffected.

Aggregating from Embedded Java

Aggregation from embedded code goes through `TimeSeriesEngine.aggregateMulti`, which reads any number of columns in a single pass:

```
TimeSeriesEngine engine = database.getSchema()
    .getTimeSeriesType("SensorReading").getEngine();

List<ColumnDefinition> columns = ((LocalTimeSeriesType) database.getSchema()
    .getType("SensorReading")).getTsColumns();

// The position of 'temperature' inside an engine row, from its position in the
schema.
int temperature = TimeSeriesGateway.aggregationRowIndex(columns,
    TimeSeriesGateway.findColumnIndex("temperature", columns));

MultiColumnAggregationResult result = engine.aggregateMulti(fromTs, toTs,
    List.of(new MultiColumnAggregationRequest(temperature, AggregationType.AVG,
"avg_temp")),
    3_600_000L, null);

for (long bucket : result.getBucketTimestamps())
```

```
    System.out.println(bucket + " -> " + result.getValue(bucket, 0));
```

A bucket interval of `0` aggregates the whole range as one bucket, answering every sealed block wholly inside it from its statistics. The overloads of `aggregateMulti`, `aggregateGrouped` and `iterateQuery` that take a `FieldFilter` read only the samples that pass range conditions on numeric fields, judged on each block's statistics before it is read. A condition is built with `FieldFilter.range`, which returns `null` for a comparison it cannot evaluate exactly as SQL does (see the SQL Queries section above):

```
// 'temperature' by its index among the columns that are not the timestamp, the
numbering TagFilter uses
FieldFilter hot = FieldFilter.range(1, columns.get(2), 30, false, null, false); //
temperature > 30

MultiColumnAggregationResult hotSamples = engine.aggregateMulti(fromTs, toTs,
    List.of(MultiColumnAggregationRequest.count("n")), 0L, 0L, null, hot, null, 0);
```

`MultiColumnAggregationRequest.columnIndex` is the position of the value inside an **engine row**: `0` is the timestamp and `1` onward are the non-`TIMESTAMP` columns in schema order. That is not the same as the column's position in the schema unless the `TIMESTAMP` column is declared first, which is why the example above converts with `TimeSeriesGateway.aggregationRowIndex`.

When upgrading from an earlier version: code that passed a schema index must convert. Nothing will warn you — the request's shape is unchanged, so such code still compiles and simply aggregates a different column. No change is needed for a type whose `TIMESTAMP` column is declared first, and none at all for the SQL, HTTP, Grafana and gRPC surfaces, which resolve the column by name.

The single-column `TimeSeriesEngine.aggregate` and `TimeSeriesSealedStore.aggregate` methods have been removed. Use `aggregateMulti` with a one-element list.

6.3.9. Query Languages

A sample of a TimeSeries type is not a vertex, an edge or a record with an identity, so the languages that read the graph model do not see it as one. What each language does over a TimeSeries type:

Language	Reads the samples	Notes
SQL	Yes	Every statement on this page.
PromQL	Yes	See PromQL Query Language.

Language	Reads the samples	Notes
MongoDB QL	Yes	`find`, `count` and `insert`, and `deleteMany` by time range and tag equality. The timestamp is a date to a MongoDB client: `find` returns it as one and, as MongoDB does, does not match a number against it, so filter it with a date. A `deleteMany` whose filter compares the timestamp with a number is refused with an error, since it would remove samples that `find` and `count` with the same filter show none of. On a type declared with a precision other than `MILLISECOND` before it became the only one (a database restored from an old export), the unit of the timestamps is not known, so no condition on the timestamp is accepted until `ALTER TIMESERIES TYPE <name> PRECISION MILLISECOND` confirms it; a `deleteMany` by tags works meanwhile. A `deleteOne`, an update and an upsert are refused too.
GraphQL	Yes, through a query field	The query field of the type answers the samples, with filters. A `@relationship` that names a TimeSeries type is refused with an error naming the type, instead of answering an empty list.
Cypher	No: refused with an error	A pattern that names a TimeSeries type (a node label, a relationship type, a dynamic label, in a read or a write clause) fails with an error naming the type, instead of answering that there is no data. Patterns over the graph types are unaffected. Read the samples with SQL.
Gremlin	No: refused with an error	A traversal that asks for the label or the edge label of a TimeSeries type (`hasLabel`, `out`, `outE`, `both`, …) fails with an error naming the type, instead of answering that there is no data. A negation of the label (`hasLabel(neq(…))`) is answered, since no vertex carries it. Read the samples with SQL.

6.3.10. PromQL Query Language

ArcadeDB includes a native PromQL parser and evaluator, providing Prometheus-compatible query capabilities without requiring an external Prometheus server.

PromQL via HTTP API

The PromQL endpoints follow the Prometheus HTTP API format and return standard `{status: "success", data: {…}}` JSON responses.

Instant query:

```
GET
/api/v1/ts/{database}/prom/api/v1/query?query=avg(cpu_usage{host="srv1"})&time=1700000
```

```
000
```

The `time` parameter is Unix seconds (float). Defaults to current time if omitted.

(Since v26.12.1) Every sample of an instant query answer is stamped with the evaluation time `time`, as in Prometheus, including selectors, aggregations, operators and `offset`. The value is still the one of the newest stored sample in the lookback. Earlier versions stamped each sample with the time of the stored sample it came from, so `count(up)` at `T+46s` could answer at `T+30s` and the series of one answer could carry different times.

(Since v26.12.1) An instant query whose expression is a range selector, such as `cpu_usage[5m]`, `cpu_usage{host="srv1"}[5m] offset 1m` or `(cpu_usage[5m])`, answers the raw samples of the window as a `matrix`, as in Prometheus: one entry per series with its labels (including `name`) and the samples it stores in the window, each with its own timestamp. Earlier versions answered `{"status":"success","data":{}}`, with no `resultType` and no `result`.

Range query:

```
GET
/api/v1/ts/{database}/prom/api/v1/query_range?query=rate(http_requests_total[5m])&star
t=1700000000&end=1700003600&step=60
```

All timestamps in Unix seconds. The `step` parameter accepts a duration string (`60s`, `1m`) or seconds as a float.

`start`, `end`, `step` and `time` are validated before the query runs: a value that is not a number, is infinite (`NaN`, `Infinity`) or falls outside a sensible date range is refused with `400 bad_data`. A range that would produce more than 1,000,000 points is refused the same way - increase `step` or narrow the range.

Earlier versions accepted such values and could take a very long time to answer, or not answer at all, on an extremely wide range.

Since v26.10.2 selector windows are left-open, as in Prometheus: an instant selector evaluated at `t` reads the samples in `(t - lookback, t]` and a range selector `[5m]` reads `(t - 5m, t]`. A series whose last sample is exactly one lookback old is therefore stale, and `count_over_time(m[5m])` over a feed whose scrape interval divides five minutes no longer counts the sample sitting on the lower boundary. Earlier versions included that boundary sample (the Prometheus 2.x behaviour for range selectors, never Prometheus's behaviour for the lookback).

Instant and range queries read samples in a streaming fashion instead of loading the whole lookback window into memory first. A query over a wide time range now uses memory proportional to the number of series returned, not to the number of samples scanned to compute it.

Label discovery:

```
GET /api/v1/ts/{database}/prom/api/v1/labels
GET /api/v1/ts/{database}/prom/api/v1/label/{name}/values
GET
/api/v1/ts/{database}/prom/api/v1/label/{name}/values?start=1700000000&end=1700003600
GET /api/v1/ts/{database}/prom/api/v1/series?match[]=cpu_usage{host=~"srv.*"}
```

These endpoints follow the Prometheus data model, where a label with an empty value counts as absent. A tag that holds no value for some samples is therefore not offered as an empty entry by `label/{name}/values`, and `series` does not report a series whose only difference from another is an empty label. The generic `/api/v1/ts/{database}/query` endpoint is unaffected and still reports exactly what the samples hold.

`label/{name}/values` honours `start` and `end` (Unix timestamps in seconds, fractional values allowed), and answers only the values a sample carries inside that window. Querying the special `name` label likewise returns only the metrics holding a sample in the window.

Both are optional. A request sending neither covers the whole series and, for `name`, names every time-series type - including one holding no sample at all - exactly as before, so existing dashboards are unaffected.

Earlier versions ignored both parameters and always answered over the whole retention period, so a Grafana variable scoped to the last hour was offered every value the label had ever held.

`start` and `end` are validated the same way as on `query_range`: a value that is not a number, is infinite, or falls outside a sensible date range is refused with `400 bad_data`, as is an `end` earlier than `start`. The same validation now applies to `series`, which previously answered `500` for a malformed value.

Supported PromQL Features

Category	Supported
Vector selectors	`metric_name`, `metric{label="value"}`, `metric{label=~"regex"}`, `metric{label!="value"}`, `metric{label!~"regex"}`, and the `offset` modifier (`metric offset 1h`). *(Since v26.10.2)* A matcher on a tag declared with a non-string type is read in the tag's declared type, and `=`, `!=`, `=~` and `!~` share one rule on `/query`, `/series` and remote read, so `=` and `!=` always select complementary sets. `{id="007"}` selects a `LONG` tag holding 7, and so does `{id=~"007"}`: a pattern that is a plain literal selects what the equality it spells selects, which is a deliberate deviation from Prometheus, where label values are text and `=~"007"` never matches `7`. A real pattern is matched against the label text only, so `=~"0.7"` does not select 7. A `DATE` or `DATETIME` matcher value is read the way `INSERT` reads it, in the database's time zone.
Range selectors	`metric[5m]`, `metric[1h]`, `metric[30s]`, `metric[1d]`; units `ms`, `s`, `m`, `h`, `d`, `w`, `y`
Aggregations	`sum`, `min`, `max`, `avg`, `count`, `topk`, `bottomk`, with `by` / `without` clauses
Range-vector functions	`rate()`, `irate()`, `increase()`, `avg_over_time()`, `min_over_time()`, `max_over_time()`, `sum_over_time()`, `count_over_time()` *(Since v26.12.1)* As in Prometheus, `rate()` and `increase()` answer no sample for a series with fewer than two samples at distinct times in the window, and `irate()` answers none when the last two samples in the window share a timestamp, so aggregations such as `avg`, `min` or `count` over them ignore that series. Earlier versions answered `0` for it. The other range-vector functions still answer a series with a single sample. *(Since v26.12.1)* `rate()` and `increase()` extrapolate to the window boundaries as Prometheus 2.51 and later do: toward a window start that lies well before the first sample they extrapolate at most half the average sample spacing, even when the counter's projected zero point is farther back. Earlier versions extrapolated up to that zero point, the pre-2.51 Prometheus behavior, and could answer a higher value.
Math functions	`abs()`, `ceil()`, `floor()`, `round()` (with an optional `to_nearest` scalar). *(Since v26.10.2)* they accept only an instant vector, as in Prometheus: a scalar argument such as `abs(-1)` is refused with `expected type instant vector in call to function "abs", got scalar`. Earlier versions answered the scalar (`abs(-1)` returned `1`).

Category	Supported
Operators	`+`, `-`, `*`, `/`, `%`, `^`, comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`), logical (`and`, `or`, `unless`). _(Since v26.10.2)_ The `bool` modifier after a comparison (`a > bool 1`) keeps every sample and answers `0` or `1` instead of filtering, and drops `__name__` from the result. `bool` right after an operator is always the modifier, never a metric name, so it is refused after an operator that is not a comparison: `a + bool` is a parse error (`bool modifier can only be used on comparison operators, not on '+'`). Two instant vectors are matched one-to-one on their full label set, `name` excluded. Vector matching modifiers (`on`, `ignoring`, `group_left`, `group_right`) are not supported.

No other PromQL function is implemented, among them `quantile`, `delta()`, `idelta()`, `stddev_over_time()`, `sqrt()`, `clamp_min()`, `label_replace()`, `label_join()`, `absent()`, `histogram_quantile()`, `vector()`, `scalar()` and `time()`. A call to one is refused with `Unknown function: <name>` (HTTP `400` `bad_data`).

Refused expressions

(Since v26.10.2) The expressions below are refused as Prometheus refuses them, with Prometheus' message. Over HTTP the answer is `400` with `"errorType": "bad_data"`; the SQL `promql()` function fails with the same message. Earlier versions answered most of them with an empty result or a `NaN` scalar.

- A comparison between two scalars without `bool`: `1 > 0` gives `comparisons between scalars must use BOOL modifier`. Write `1 > bool 0`.
- `and`, `or` or `unless` with a scalar operand: `m and 1` gives `set operator "and" not allowed in binary scalar expression`.
- A range vector or a string as the operand of an operator: `m[5m] + 1`, `-m[5m]` and `"a" + 1` give `binary expression must contain only scalar and instant vector types` (for the unary minus, `unary expression only allowed on expressions of type scalar or instant vector`).
- A function or aggregation argument of the wrong type, or the wrong number of arguments: `sum(m[5m])` and `sum(1)` give `expected type instant vector in aggregation expression, got range vector` (or `scalar`), `rate(m)` gives `expected type range vector in call to function "rate", got instant vector`, and `abs()` gives `expected 1 argument(s) in call to "abs", got 0`.
- A range query (`query_range`) whose expression is a range vector or a string: `m[5m]` gives `invalid expression type "range vector" for range query, must be Scalar or instant Vector` (`"string"` for a string literal). An instant query still accepts both.

PromQL via SQL

The `promql()` SQL function calls the PromQL evaluator from within SQL queries:

```
-- Instant query at explicit time
```

```
RETURN promql('cpu_usage{host="srv1"}', 1700000000000)

-- Rate calculation
RETURN promql('rate(http_requests_total[5m])')

-- Scalar arithmetic
RETURN promql('2 + 3 * 4', 1000)
```

The function accepts 1-2 arguments: the PromQL expression (required) and an optional evaluation timestamp in milliseconds.

Prometheus Remote Read

Configure Prometheus to read from ArcadeDB for long-term storage:

```
# prometheus.yml
remote_read:
  - url: "http://localhost:2480/api/v1/ts/mydb/prom/read"
    basic_auth:
      username: root
      password: password
```

The endpoint accepts the standard Prometheus `remote_read` Protobuf query (snappy-compressed) and supports `=`, `!=`, `=~`, `!~` label matchers, applied with the same rules as the PromQL `/query` and `/series` endpoints (including the typed-tag rule in the *Vector selectors* row of the table above). A `name` matcher that is not an equality (for example `{name=~"cpu.*"}`) selects every metric it matches among those the user can read. An invalid or unsafe regular expression is refused with HTTP `400` rather than answered with an empty result *(since v26.10.2)*.

(Since v26.10.2) Like remote write, the endpoint reads the `proto=` parameter of the `Content-Type` header and answers `415` when it names a message other than `prometheus.ReadRequest`, and a malformed protobuf body is answered `400` instead of `500`.

6.3.11. Continuous Aggregates

Continuous aggregates are pre-computed time-bucketed rollups that are automatically refreshed when new data is inserted. They dramatically speed up common dashboard queries by maintaining materialized summaries.

```
-- Create a continuous aggregate
CREATE CONTINUOUS AGGREGATE hourly_temps AS
  SELECT ts.timeBucket('1h', ts) AS hour,
         sensor_id,
         avg(temperature) AS avg_temp,
         max(temperature) AS max_temp,
         count(*) AS cnt
  FROM SensorReading
  GROUP BY hour, sensor_id
```

```
-- Query the aggregate like any other type
SELECT * FROM hourly_temps
WHERE hour BETWEEN '2026-02-19' AND '2026-02-20'

-- Manual refresh
REFRESH CONTINUOUS AGGREGATE hourly_temps

-- Drop
DROP CONTINUOUS AGGREGATE hourly_temps
```

The defining query must reference a TimeSeries source type and contain a `ts.timeBucket()` call with a `GROUP BY` clause. The interval is checked when the aggregate is created, by the same rules as the query: an interval that is too large or not a positive amount of time is refused, and no aggregate is left behind, even when the source type is still empty *(since v26.12.1, issue #11986)*. After creation, every insert into the source type triggers an incremental refresh using watermark tracking — only new data since the last watermark is processed.

A refresh is never part of the caller's transaction. `REFRESH CONTINUOUS AGGREGATE`, `ContinuousAggregate.refresh()` and the first refresh of `CREATE CONTINUOUS AGGREGATE` run inside a transaction of their own, committed before they return, even when the caller has one open: a `ROLLBACK` of the caller's transaction does not take the rows of the aggregate back, exactly as it does not take the aggregate itself or the samples it reads back. The refresh an `INSERT`, `DELETE FROM` or `TRUNCATE TYPE` on the source type triggers inside a transaction runs once, when that transaction ends, whether it commits or rolls back, since the samples it answers for are written either way. *(Since v26.11.1, issue #10708. Before, a refresh inside a transaction joined it, and a `ROLLBACK` left the watermark advanced over rows that were gone: the buckets behind it were lost for good, with the aggregate reported `VALID`. A `ROLLBACK` also dropped the refresh an `INSERT` or a delete was owed.)*

Inspect continuous aggregates via:

```
SELECT FROM schema:continuousAggregates
```

This returns name, query, source type, bucket column, bucket interval, watermark timestamp, `watermarkSet`, status, and metrics for each aggregate. `watermarkSet` tells a watermark of `0` that has actually been reached — an aggregate whose newest bucket is the epoch — apart from an aggregate that has never been refreshed, which reports `0` as well.

The watermark is the bucket boundary of the newest row the last refresh produced. Each refresh recomputes the buckets at or after it from the source and replaces the backing rows of those buckets, so the incomplete bucket that was open when the previous refresh ran is always rebuilt rather than duplicated. A recomputed row is written over the row of its own bucket, in place, rather than deleted and inserted again.

The backing type is indexed on its bucket column (a `DATETIME` property with a `NOTUNIQUE` index), so a refresh reads only the rows of the buckets it replaces and the same index serves a range query on the aggregate (`SELECT FROM hourly_temps WHERE hour >= …`). *(Since v26.11.1.)* An aggregate created by an earlier version has no such index, and its refresh reads every row the aggregate holds: with

100,000 rows that is about 400 times the cost of the `INSERT` that triggers it. The database names each one in a warning when it opens, with the statements that index it, for example:

```
CREATE PROPERTY `hourly_temps`.`hour` DATETIME;
CREATE INDEX ON `hourly_temps` (`hour`) NOTUNIQUE
```

An index the bucket column leads (`(hour, sensor_id)`) serves as well. A defining query may carry its own `WHERE` clause, including a disjunction: the watermark filter is added as a separate conjunct around it, so `WHERE a OR b` keeps meaning `ts >= watermark AND (a OR b)`.

The watermark only reaches forward, so a bucket behind it is not recomputed by the refresh that follows an insert. `DELETE FROM` and `TRUNCATE TYPE` on the source type make up for it: they record, on every aggregate over the type, the time range they removed samples from, and the next refresh recomputes the buckets that range reaches and only those. The buckets behind the range, and the ones between it and the watermark, are not touched. The record is part of the definition of the aggregate in the schema, so it survives a restart, and under HA it is shipped to the followers in the replication unit of the delete. A bucket is found among the rows the aggregate holds, so the grid of the aggregate (an `origin`, an `offset` or a `timezone` given to `ts.timeBucket()`) needs no special care. A delete that removes nothing records nothing. Deletes that come before the next refresh runs are merged into one range, so two deletes far apart in time recompute the buckets between them too. A sample written with a timestamp behind the watermark is recorded the same way, as the range of buckets of the grid it reaches, and the next refresh recomputes those buckets, including a bucket that holds no row yet. The range of the late samples is kept apart from the range of the deletes, so a late sample written after a `DELETE FROM` or a `TRUNCATE TYPE` of the whole type still reaches its bucket, and the range of a delete is cleared once the refresh recomputed it. *(Since v26.12.1, issue #13049. Before, a late sample written after a delete of the whole type never reached its bucket, a* `REFRESH CONTINUOUS AGGREGATE` *did not repair it, and the range of the delete stayed recorded, recomputed by every later refresh.)* The `RETENTION` of the source type does not change the buckets that are already computed.

Renaming the source type (`ALTER TYPE SensorReading NAME Readings`) carries the aggregate with it: its stored source type and the `FROM` of its defining query are rewritten, so it keeps refreshing and `DROP TYPE` keeps refusing to drop the source. The **backing** type of an aggregate is named after the aggregate and cannot be renamed. *(Since v26.11.1.)*

The backing type is written by the refresh of the aggregate only. `INSERT`, `UPDATE` and `DELETE` on it, `TRUNCATE TYPE` on it (directly, or reaching it with `POLYMORPHIC` from a supertype), `TRUNCATE BUCKET` on one of its buckets and the removal of one of its buckets are refused, from SQL, the Java API and every protocol, naming `REFRESH CONTINUOUS AGGREGATE` and `DROP CONTINUOUS AGGREGATE`; a refused `TRUNCATE` deletes nothing. To change what the aggregate holds, change the samples of the source type: a delete there records the range to recompute, as described above. *(Since v26.11.1, issue #11020. Before, a* `TRUNCATE TYPE` *or a* `DELETE FROM` *of the backing type left the watermark where it was, so the refresh that followed recomputed only the buckets at or after it: every bucket behind it was lost for good while the aggregate reported* `VALID`*, and only dropping and creating the aggregate again brought them back.)* No type can extend the backing type either: `CREATE … TYPE … EXTENDS <aggregate>` and `ALTER TYPE … SUPERTYPE +<aggregate>` are refused, since the aggregate would answer the subtype's rows as buckets it computed. *(Since v26.11.1, issue #11204.)*

6.3.12. Retention Policies

Retention policies automatically delete data older than a specified duration. Set the retention period during type creation:

```
CREATE TIMESERIES TYPE SensorReading
  TIMESTAMP ts
  TAGS (sensor_id STRING)
  FIELDS (temperature DOUBLE)
  RETENTION 90 DAYS
```

A background maintenance scheduler (60-second interval) automatically enforces retention and downsampling policies on all TimeSeries types.

6.3.13. Downsampling Policies

Downsampling reduces the resolution of old data to save storage while preserving long-term trends. Multiple tiers can be defined to progressively reduce resolution as data ages:

```
ALTER TIMESERIES TYPE SensorReading
  ADD DOWNSAMPLING POLICY
    AFTER 7 DAYS GRANULARITY 1 MINUTES
    AFTER 30 DAYS GRANULARITY 1 HOURS
```

In this example, data older than 7 days is downsampled to 1-minute resolution, and data older than 30 days is further reduced to 1-hour resolution. The downsampling process aggregates values using AVG within each granularity bucket, preserving tag groupings.

(Since v26.12.1) A block of old data is downsampled whenever at least two of its samples with the same tag values fall in one bucket, however its other samples are spread. A block that holds at most one sample per series and bucket is already at the tier's resolution and is left as it is, keeping its original timestamps. Earlier versions decided on the average spacing of a block's samples instead, so a burst of samples inside one bucket stayed at raw resolution when the same block also held a distant sample, and a sparse block was moved onto the bucket starts.

Reads That Cross a Downsampling Pass

Downsampling replaces samples rather than deleting them, so a read already in flight when a pass runs cannot finish coherently: the rows it has already returned are the original samples, and the rows still to come are their coarser replacements. Joining the two would report the bucket at the crossing point at two different resolutions.

Such a read is therefore **refused** rather than allowed to return a short answer:

- Over HTTP — including PromQL and the Grafana endpoints — the response is `503`, so a client that already retries on `503` recovers on its own.
- Over gRPC, Bolt, MongoDB and Postgres the equivalent retryable status is returned.

- `EXPORT DATABASE` fails only the TIMESERIES type it was reading; every other type in the database is still exported.

Re-running the read succeeds and returns a complete answer at a single resolution, because downsampling is a scheduled maintenance event rather than something that happens per request.

A **retention** pass is different. It deletes old data outright, so a read that crosses one still completes and simply returns the rows that remain.

(Since v26.10.2) An unbounded read streams too: a SQL `SELECT` from a TIMESERIES type with no `LIMIT` in ascending order, the gRPC time-series stream without a row limit, and `TimeSeriesEngine.iterateQuery` in Java decode one sealed block per shard at a time instead of loading the whole range before returning the first row, so their memory no longer grows with the range (a 50-million-point range used to exhaust a 4 GB heap). They therefore follow the rules above: a block a retention pass deletes mid-read is skipped, and a downsampling pass mid-read refuses the read with the retryable error. Earlier versions held the type's compaction lock while loading the range, so neither pass could change what such a read returned.

What an Export Reports

The `EXPORT DATABASE` summary carries two counters for TIMESERIES data that changed underneath it:

Field	Meaning
`vanishedTimeSeriesBlocks`	Blocks a retention pass deleted while the export was reading them. Their samples are genuinely gone, so the export is still valid for the database as it now stands — but it is not the snapshot it would have been a moment earlier. Reported, never fatal.
`partialTimeSeriesTypes`	Types whose export was cut short by a downsampling pass. The rows already written for such a type are real but incomplete, and the export as a whole is reported as failed so the gap cannot pass unnoticed. Re-run the export for a complete result.

6.3.14. Grafana Integration

ArcadeDB provides Grafana DataFrame-compatible HTTP endpoints that work with the Grafana Infinity datasource plugin — no custom plugin is needed.

Endpoints:

Endpoint	Description
`GET /api/v1/ts/{database}/grafana/health`	Datasource health check
`GET /api/v1/ts/{database}/grafana/metadata`	Discovers TimeSeries types, fields, tags, and available aggregation types

Endpoint	Description
`POST /api/v1/ts/{database}/grafana/query`	Multi-target query returning Grafana DataFrame wire format

The query endpoint supports raw queries, aggregated queries (SUM/AVG/MIN/MAX/COUNT), tag filtering, field projection, and automatic bucket interval calculation from Grafana's `maxDataPoints` setting.

Grafana Infinity datasource configuration:

1. Install the Grafana Infinity datasource plugin.
2. Add a new Infinity datasource.
3. Set the base URL to `http://<arcadedb-host>:2480`.
4. Configure authentication (Basic Auth with ArcadeDB credentials).
5. Use the health endpoint for health checks.

6.3.15. Studio TimeSeries Explorer

The ArcadeDB Studio web interface includes a dedicated TimeSeries Explorer accessible from the main navigation sidebar.

The explorer provides four tabs:

- **Query** — Time range selector, aggregation controls, field checkboxes, interactive charts (Apache ECharts with zoom), data table with pagination, auto-refresh capability
- **Schema** — Type introspection with column roles (TIMESTAMP/TAG/FIELD badges), diagnostics (total samples, shards, time range), configuration details, downsampling tiers, per-shard statistics
- **Ingestion** — Documentation and examples for all four ingestion methods with a method comparison table
- **PromQL** — PromQL expression input, instant/range toggle, time controls, chart and table rendering of results

6.3.16. HTTP API Reference

Method	Endpoint	Description
`POST`	`/api/v1/ts/{database}/write?precision=<ns|us|ms|s>`	InfluxDB Line Protocol ingestion. Body: plain text lines. Returns 204 on success.
`POST`	`/api/v1/ts/{database}/query`	JSON query endpoint. Supports raw and aggregated queries with tag filtering and field projection. A `tags` name that is no `TAG` column of the type is refused with 400. A `DATE`/`DATETIME` tag value is read in the database's time zone.

Method	Endpoint	Description
GET	/api/v1/ts/{database}/latest?type=<name>&tag=<key>:<value>	Returns the most recent data point, with optional tag filter. Repeat tag to narrow across several tag columns; every occurrence must match. An occurrence that is malformed, or names no TAG column, is refused with 400. A DATE/DATETIME tag value is read in the database's time zone.
GET	/api/v1/ts/{database}/prom/api/v1/query?query=<promql>&time=<unix_seconds>	PromQL instant query.
GET	/api/v1/ts/{database}/prom/api/v1/query_range?query=<promql>&start=<s>&end=<s>&step=<s>	PromQL range query.
GET	/api/v1/ts/{database}/prom/api/v1/labels	List all label names (metric names and tag columns).
GET	/api/v1/ts/{database}/prom/api/v1/label/{name}/values	List distinct values for a label.
GET	/api/v1/ts/{database}/prom/api/v1/series?match[]=<selector>	Find series matching PromQL selector(s).
POST	/api/v1/ts/{database}/prom/write	Prometheus remote_write 1.0 endpoint (snappy-compressed Protobuf). Answers 415 to a remote write 2.0 Content-Type.
POST	/api/v1/ts/{database}/prom/read	Prometheus remote_read endpoint (snappy-compressed Protobuf).
GET	/api/v1/ts/{database}/grafana/health	Grafana datasource health check.
GET	/api/v1/ts/{database}/grafana/metadata	Grafana metadata discovery.
POST	/api/v1/ts/{database}/grafana/query	Grafana DataFrame query endpoint.

6.3.17. High Availability

In an HA cluster, TimeSeries data is handled as follows:

- **Mutable bucket data** (.tstb files) is replicated to all followers via ArcadeDB's standard page-replication protocol
- **Sealed store data** (.ts.sealed files) is replicated too, but by its own path: compaction, retention and downsampling run on the leader only, and the rewritten sealed file is shipped to the followers together with the change that clears the matching mutable data. Followers do not compact independently
- **Reads are consistent after failover**: the engine queries both sealed and mutable layers, so a newly promoted leader returns correct results even before its first compaction cycle
- **Compaction lag**: a node whose maintenance scheduler has not run yet (default: 60 seconds) serves the same results, just with more of the answer coming from the mutable layer
- **Backups and snapshots taken on a follower** are consistent. The sealed file and the clearing of

the mutable data arrive together, and a copy taken across that moment waits for the install to finish

6.3.18. Integrity Checking

If a type's sealed store fails to open at all (for example, a corrupted `.ts.sealed` file), the type stays visible in the schema instead of disappearing. `CHECK DATABASE` reports it as corrupted and names the file; any query, write, or REST/PromQL/Grafana call against the type fails with a clear error instead of silently recreating an empty type or returning wrong results. Earlier versions dropped the type from the schema silently on the next restart.

Since 26.12.1 every TIMESERIES type draws a random owner id when it is created. It is stored in the schema (`sealedOwner`) and written into the header of every `.ts.sealed` file of the type's shards (sealed format version 4, a 35-byte header with the owner at offset 27). When the engine opens the type, a sealed file at its path that was written for another type, another shard or another database (for example a file copied in by hand) is refused: the type stays registered, but its storage is reported unavailable as described above (a `SEVERE` log line, a finding of `CHECK DATABASE`, and an error saying why on every read and write), and the file is neither read nor overwritten. Move the foreign file away and reopen the database to restore the type. Renaming the type keeps its owner. A type of a database written before 26.12.1 has no owner and reads its files as before (issue #11274).

Restoring a whole database directory (the schema together with its files) and HA replication are unaffected. As with the earlier sealed format changes, a sealed file that 26.12.1 rewrites whole (compaction, retention, downsampling) is at version 4, which an older build cannot read, and an older build refuses at schema load a type created by 26.12.1 (`sealedFormatVersion` 4). See Upgrade ArcadeDB.

A TimeSeries type has neither record buckets nor indexes, so `CHECK DATABASE` walks it with a pass of its own that reads all three of the on-disk formats it owns:

- the **mutable bucket** (`.tstb`) — page 0's counters are reconciled against the samples its data pages actually hold
- the **tag dictionary** (`.tstd`) — the declared entry count against the entries walked
- the **sealed store** (`.ts.sealed`) — the header, the block directory, the block bounds, and the CRC32 of every block

```
CHECK DATABASE
CHECK DATABASE TYPE SensorReading
```

The CRC32 pass reads the whole sealed file. That is deliberate: a sealed block is otherwise verified only on the first read that touches it, so a block nothing queries is a block nothing verifies.

The DEEP tier

CHECK DATABASE DEEP adds the checks that have to decompress the blocks. A CRC proves the bytes are the bytes that were written; it proves nothing about whether they mean what the block claims. Three things the read paths answer queries from, without ever looking at a value, are verified only here:

Claim	What a wrong one does
Timestamps are sorted	The range iterator binary-searches them, so a query over the block silently returns a subset of the matching rows
Per-column min/max/sum	Aggregation push-down answers MIN/MAX/SUM/AVG from them without decompressing, so the query returns a wrong number
Distinct tag values	Block pruning skips a whole block whose declaration omits the value being filtered on, hiding those rows entirely

Each of those produces a wrong **answer** rather than an error, which is why nothing else in the engine reports them.

DEEP also reports a block written by an earlier version whose column holds only NaN samples: such a block stored an internal placeholder as its min/max, and aggregation push-down returns that placeholder as if it were a measurement. The check names the column and the remedy - recompacting or downsampling the block rewrites its header. Existing data is not corrected until then.

```
CHECK DATABASE DEEP
CHECK DATABASE TYPE SensorReading FIX DEEP
```

What FIX repairs

FIX repairs only what is **derived** from the data, and never a sample:

- the mutable bucket's page-0 counters (sample count, min and max timestamp)
- the sealed header's block count and global timestamp bounds
- the tail of a sealed append that did not complete

None of those is cosmetic. The global bounds are read straight out of the header when the file is opened rather than recomputed, so a range query pruned against a wrong bound silently misses data the file holds; and because new blocks are appended at the end of the file, a tail no reader can see also hides every block appended after it.

A sealed block that fails its CRC, or that DEEP finds inconsistent, is reported and left untouched. It is the only copy of the samples in it, so discarding it is an operator's decision - and in an HA cluster the sealed store is derived, so a node can rebuild one by recompacting from its replicated mutable pages.

The report carries totalTimeSeriesTypes, totalTimeSeriesShards, totalTimeSeriesSamples and

`totalTimeSeriesSealedBlocks` alongside the findings, so a clean result can be told apart from a pass that never ran. Findings appear in `warnings` with the offending type named in `corruptedTimeSeries`; repairs are counted in `repairedTimeSeries` and listed in `timeSeriesRepairs`.

6.3.19. Comparison with Other TimeSeries Databases

Feature	ArcadeDB	InfluxDB 3	TimescaleDB	Prometheus	QuestDB
License	Apache 2.0	MIT (core)	Apache 2.0 (core)	Apache 2.0	Apache 2.0
Multi-Model	Graph + Document + K/V + TimeSeries + Vector in one engine	TimeSeries only	Relational + TimeSeries (PostgreSQL extension)	Metrics only	TimeSeries only (SQL)
Query Languages	SQL, PromQL, MongoDB QL, GraphQL (query fields). Cypher and Gremlin refuse a TimeSeries type with an error, see Query Languages	SQL, InfluxQL	Full PostgreSQL SQL	PromQL	SQL (PG wire)
Ingestion Protocols	Line Protocol, Prometheus remote_write, SQL, Java API	Line Protocol, SQL	SQL (INSERT, COPY)	Prometheus scrape, remote_write	Line Protocol, SQL, CSV
Compression	Gorilla, Delta-of-Delta, Simple-8b, Dictionary	Parquet native (Delta, Dict, Snappy/ZSTD)	Gorilla, Delta-of-delta, Simple-8b, Dictionary, LZ4	Gorilla (~1.37 B/sample)	ZFS-level + Parquet for cold tier
Continuous Aggregates	Yes (watermark-based, auto-refresh on commit)	Materialized views	Yes (policy-based refresh)	Recording rules	No
Downsampling	Yes (multi-tier, automatic)	Via compaction	Via continuous aggregates + retention	Recording rules	No
Retention Policies	Yes (automatic, per-type)	Yes	Yes (per-chunk)	Yes (per-block)	Yes (per-partition)

Feature	ArcadeDB	InfluxDB 3	TimescaleDB	Prometheus	QuestDB
Grafana Integration	DataFrame endpoints (Infinity plugin)	Native plugin	PostgreSQL datasource	Native plugin	PostgreSQL datasource
PromQL Support	Native (parser + evaluator + HTTP endpoints)	No	Via adapter	Native	No
Embeddable (in-process)	Yes (Java library)	No	No (requires PostgreSQL)	No	No (separate process)
SIMD Aggregation	Yes (Java Vector API)	Via DataFusion (Arrow)	No	No	Yes (AVX2)
Prometheus Remote Write/Read	Yes	No	Via adapter	Native	No
Graph + TimeSeries Queries	Yes (native cross-model)	No	No	No	No

6.3.20. Choosing the layout: `COMPACTION_INTERVAL` and `SHARDS`

For a type that is bulk-loaded and then aggregated in fixed buckets (for example hourly averages), declare `COMPACTION_INTERVAL` with the bucket width of that aggregation and leave `SHARDS` at its default:

```
CREATE TIMESERIES TYPE Point TIMESTAMP ts TAGS (host STRING)
  FIELDS (uu DOUBLE, us DOUBLE, ui DOUBLE)
  COMPACTION_INTERVAL 1 HOURS
```

- A sealed block answers an aggregate from its min/max/sum statistics only when it lies inside ONE query bucket. Without an interval a block covers many hours and is decompressed (`PROFILE` reports it under `slow`); with `1 HOURS` the blocks are cut at the hour and are counted under `fast`.
- Other queries on the same type are not penalised. A last-point query (`ORDER BY ts DESC LIMIT 1`) walks the blocks newest-first and stops at the first block that has the row, and a finer bucket (`timeBucket('1m', ts)`) over a short range reads only the blocks of that range, so smaller blocks only mean less to decompress. Ingest is not slowed either: blocks are cut when the mutable data is compacted, not while samples are appended.
- The price is more, smaller blocks (each shard cuts at every interval boundary), so a very small interval on a long retention multiplies the block count. Pick the bucket of the aggregation you run most, not the smallest one you might ever run.
- `SHARDS` trades ingest parallelism for fragmentation: every shard cuts its own blocks, so doubling

the shards roughly doubles the blocks per interval, and queries merge one more stream. The default (available cores minus 1) suits concurrent writers; for a type loaded by one writer and queried more than written, fewer shards are enough.

- Run `COMPACT TIMESERIES TYPE <name>` after a bulk load: mutable samples are not in sealed blocks yet, and only sealed blocks are answered from their statistics.

6.3.21. Architecture

The TimeSeries engine uses a two-layer storage architecture:

- **Mutable bucket** — An append-only in-memory buffer backed by ArcadeDB's `PaginatedComponent`. New samples land here first. This layer is ACID-transactional and replicated in HA mode.
- **Sealed store** — Immutable, compressed columnar blocks on disk. The maintenance scheduler periodically compacts mutable data into sealed blocks using Gorilla, Delta-of-Delta, Simple-8b, and Dictionary codecs. Block-level min/max/sum statistics enable zero-decompression aggregation when an entire block falls within a single time bucket (for an aggregation with no `GROUP BY`, within the queried range), and let a range condition on a numeric field skip a block without reading it.

Since v26.10.2. A sealed block keeps the statistics of an integer column (`LONG`, `INTEGER`, `SHORT`, `BYTE`, `BOOLEAN` and the date/time types) exactly: the minimum and the maximum as 64-bit integers and the sum as a 128-bit integer, so `min`, `max` and `sum` of such a column past 2^53 are answered from the block headers, exactly, without decompressing anything. A block written by an earlier version is answered from its headers whenever its statistics are provably exact, and decompressed otherwise until the next compaction rewrites it. The sealed file format version is 3: an older ArcadeDB version refuses to open such a file, so keep this in mind before a downgrade.

Since v26.10.2. Integer columns (`LONG`, `DATETIME_NANOS` and the other integral types) hold the full 64-bit range. A block whose values do not all fit the compact Simple-8b packing (outside +/-2^59, such as epoch-nanosecond instants or snowflake ids) is stored as plain 64-bit values instead of making the compaction fail. Such a block is larger, and an older ArcadeDB version cannot read it, so keep this in mind before a downgrade. Before, compaction of such a type failed permanently.

Data is distributed across N shards (default: `arcadedb.asyncWorkerThreads`, the available cores minus 1) for parallel writes and reads. Each shard maintains its own mutable bucket and sealed store. Queries merge results across shards using a min-heap priority queue sorted by timestamp.

See Also

- Time Series Tutorial — Step-by-step hands-on guide to time series in ArcadeDB
- Realtime Analytics — Use case for streaming ingestion and real-time dashboards

6.4. Vector Search

ArcadeDB includes a native vector search engine for similarity-based retrieval of embeddings. Vector indexes are fully integrated into the SQL query engine and support ACID transactions, persistent storage, and automatic compaction.

6.4.1. How Vector Search Works

Vector search finds the nearest neighbors to a query vector in high-dimensional space. Instead of exact matching (like SQL `WHERE`), it finds the most *similar* items based on a distance or similarity metric.

Typical workflow:

1. Generate embeddings from your data using an external model (OpenAI, Sentence Transformers, etc.)
2. Store embeddings as vector properties on vertices or documents
3. Create a vector index on the property
4. Query with `vector.neighbors()` to find the k most similar items

6.4.2. LSMVectorIndex Architecture

ArcadeDB's vector index is built on two foundations:

- **LSM Tree storage** — ArcadeDB's proven LSM Tree architecture provides persistent, crash-safe storage with automatic compaction
- **JVector 4.0.0** — A high-performance vector search library that implements both HNSW (Hierarchical Navigable Small World) and Vamana (DiskANN) graph algorithms

The index stores vectors as a navigable graph where each node connects to its approximate nearest neighbors. Searches traverse this graph, narrowing in on the closest matches efficiently — typically in O(log n) time rather than O(n) brute-force scanning.

6.4.3. Flat vs Hierarchical Structure

The index supports two graph structures:

	Flat (default)	**Hierarchical**
Algorithm	Single-layer Vamana graph	Multi-layer HNSW with exponential decay
Build speed	Faster	10-20% slower
Disk usage	Baseline	5-15% larger
Best for	< 100K vectors, well-clustered data	100K+ vectors, 1024+ dimensions, diverse queries

Enable hierarchical mode with `addHierarchy: true` in the index metadata.

6.4.4. Similarity Functions

Three distance metrics are available:

Function	When to Use	Value Range
COSINE (default)	Text embeddings (BERT, GPT, Sentence Transformers). Direction matters, magnitude does not.	-1 to 1
DOT_PRODUCT	Normalized vectors where speed matters. 10-15% faster than COSINE.	Unbounded
EUCLIDEAN	Spatial data, point clouds, continuous measurements. Absolute distance matters.	0 to infinity

If your embeddings are already L2-normalized (unit vectors), use DOT_PRODUCT for best performance — it produces the same ranking as COSINE but skips the normalization step.

6.4.5. Quantization

Quantization reduces memory usage by compressing vector components at the cost of slight accuracy loss:

Type	Memory Reduction	Speed	Recall	Use Case
NONE	Baseline	Baseline	100%	Small datasets (< 10K vectors), maximum accuracy
INT8 (recommended)	4x (75% savings)	Lower latency than NONE; the gain depends on the data and the recall target (see Why INT8 can be faster)	95-98%	Best balance of speed and accuracy for most workloads
BINARY	32x (97% savings)	Lower latency than NONE; the gain depends on the data and the recall target (see Why INT8 can be faster)	85-92%	Massive datasets, approximate search with reranking

Type	Memory Reduction	Speed	Recall	Use Case
PRODUCT	16-64x for the PQ codes in memory (the stored vectors are still read to re-rank)	Search walks on the PQ codes, then re-ranks on the stored vectors	Close to NONE with the default re-rank	Very large datasets (100K+), enables zero-disk-I/O graph construction. INT8 is the better default unless the PQ codes are what you need

Use INT8 quantization for most use cases. It provides 4x memory savings with minimal accuracy loss and usually lower search latency (see Why INT8 can be faster). Only use NONE for very small datasets where maximum precision matters.

Why INT8 can be faster

Besides saving memory, quantization changes how vectors are read during search.

Without quantization (`NONE`), each node visited during graph traversal requires a full document lookup: read the record from disk, deserialize it, and extract the vector property. With INT8, vectors are stored in compact contiguous index pages and read directly — no document deserialization needed.

One benchmark with 500K 384-dimensional vectors (matching the `all-MiniLM-L6-v2` embedding model) measured the latencies below. Treat them as one setup, not a general figure. The gap between INT8 and NONE depends on the data, on whether the NONE vectors are served from memory or read from documents, and on the recall target, so measure it on your own data:

Quantization	Mean latency	p95 latency	Vector fetch path
NONE	3.50 ms	4.36 ms	Document lookup (random I/O)
INT8	**1.59 ms**	**1.94 ms**	Index pages (sequential I/O)

The difference can be larger under memory pressure. With NONE quantization, vector data is 4x larger, evicting more data from memory caches and forcing real disk I/O. INT8 keeps the working set small enough to stay in memory even with constrained resources.

Quantization is transparent — queries work identically regardless of quantization setting. The index automatically quantizes on insert and dequantizes on retrieval.

When using PRODUCT quantization, the graph build uses Product Quantization scores instead of exact vector distances. PQ codes are compact and stay in memory, eliminating disk I/O during graph construction. This is most effective on large datasets (100K+ vectors) where PQ quality is sufficient.

Since 26.11.1 a search of a PRODUCT index (`vector.neighbors`, `db.index.vector.queryNodes` and the fluent `select().nearestTo()`) walks the graph on the in-memory PQ codes, keeps `k * rerankOversample` candidates and scores those on the stored vectors, as BINARY does, so the distances it returns are exact and its recall is close to a NONE index. `rerankOversample` is an index setting (default 10) and a query overrides it with the `rerank` option of `vector.neighbors`; `0` returns the PQ scores alone, which reads no stored vector but reaches a recall of only about 0.7 on SIFT-1M with 32 subspaces, whatever the beam. Before 26.11.1 these searches scored every visited node on the stored vectors, as a NONE index does, so the PQ codes were used to build the graph and never to search.

The beam of a re-ranked search is `max(k * rerankOversample, efSearch)`, so an `efSearch` below `k * rerankOversample` changes nothing: with the default of 10 and `k` = 10, an `efSearch` of 10, 50 or 100 gives the same answer. Lower `rerank` (or `rerankOversample`) to make a PRODUCT search cheaper, not `efSearch`.

Since 26.11.1 a search of a BINARY index walks the graph on the one-bit codes of the vectors: each candidate is scored with a Hamming distance between its code and the code of the query, which is the similarity of their one-bit reconstructions, and the codes are kept in the index's search cache (about `dimensions / 8` bytes per vector, see `vectorIndex.searchCacheSize` for the exact sizing). The search then keeps `k * rerankOversample` candidates, or the whole `efSearch` beam when it is larger, and scores those on the stored vectors, so the distances it returns are exact, as before. `rerankOversample` is the same index setting a PRODUCT index uses (default 10), and the `rerank` option of `vector.neighbors` overrides it per query; a BINARY index always reranks at least one candidate per neighbor.

Before 26.11.1 a BINARY search rebuilt every candidate it visited as a float vector and scored that, which cost as much per candidate as a NONE index and filled the search cache with float vectors, and it reranked a fixed `4 * k` candidates whatever the `efSearch`, so a wider beam cost more without raising recall. An existing BINARY index now reranks `10 * k` candidates by default: recall is higher, and a search is still cheaper than before. Lower `rerankOversample` (or `rerank` on a query) for a cheaper and less accurate search, raise `efSearch` or `rerank` for a more accurate one.

The Java API `findNeighborsFromVectorApproximate(query, k)` and the select `approximate(true)` keep the PQ-scores-only answer, with no record read; add `rerank()` to the select, or pass `rerankOversample` to the method, to re-rank. Before 26.11.1 they could not.

The graph file of a PRODUCT index no longer stores the PQ codes of every node's neighbors (jvector's FusedPQ) unless the `fusedPQ` index setting is `true`; with it off the graph file is about the size of a NONE index's, and the codes live in the `.vecpq` file. Before 26.11.1 the codes were always written and nothing read them, which made a PRODUCT graph file about 8x the size of a NONE one. Files written earlier keep the codes and are read through them. `fusedPQ` pays off only when the PQ codes cannot stay in memory. `pqTrainingLimit` is the number of vectors, a random sample, the PQ codebook is trained on (default and maximum 128000); before 26.11.1 it was accepted and ignored.

Since 26.11.1 a graph rebuild of a PRODUCT index keeps the PQ codebook while it still fits the data: the codes of the surviving vectors are carried over and only the vectors added since are encoded. Before, every rebuild trained a new codebook and encoded every vector again. The codebook is

trained again, and every vector encoded again, when either test fails:

- **Growth**: the index holds more than `vectorIndex.pqRetrainGrowthFactor` times the vectors it held when the codebook was trained (default 2.0; 0 or less retrains on every rebuild, the behavior before 26.11.1).
- **Drift**: the mean relative reconstruction error (`|v - decode(encode(v))|^2 / |v|^2`) of a sample of up to 1024 vectors added since the training is more than `vectorIndex.pqRetrainErrorRatio` times the error measured on the training vectors (default 1.25; 0 or less disables this test). It applies once at least 32 vectors have been added since the training.

`REBUILD INDEX` always trains a new codebook. PQ training and encoding run on the index's graph-build pool, so `vectorIndex.graphBuildParallelism` bounds the CPU of the whole rebuild; before 26.11.1 PQ training ran on JVector's own default executors. See the database settings.

With `rerankOversample: 0`, or `rerank: 0` on a query, the distances returned by a PRODUCT-quantized index are the approximate PQ ones. All records are measured the same way, whether they were written before or after the last index rebuild.

A rebuild replaces the graph and its PQ codes a moment apart. A query landing in between answers from the exact vectors instead, so it returns normally — a single query may take longer while the rebuild finishes, and the fast path resumes on its own.

Table 1. Memory and performance example: 100K vectors with 384 dimensions

Quantization	Vector memory	Search latency
NONE	156 MB	Baseline
INT8	39 MB	Lower than NONE; depends on the data and the recall target
BINARY	5 MB	Lower than NONE, with lower recall; depends on the data and the rerank

6.4.6. Vector Encoding (Pre-Quantized Ingest)

The `encoding` index option controls the **wire and document-storage** representation of vectors. It is distinct from `quantization`, which controls the **index-internal** compression scheme:

Knob	What it changes	Trade-off
`encoding`	What the document column stores and what HTTP clients send	Wire payload, bucket bytes, client round trip
`quantization`	What the HNSW graph compresses internally	Index memory footprint, recall, search latency

Two encodings are supported:

- `FLOAT32` **(default)** — the `embedding` property is `ARRAY_OF_FLOATS` (4 bytes per dimension; before 26.11.1 a stored float took 5 bytes, see Data Types).
- `INT8` — the `embedding` property is `BINARY` (one signed byte per dimension). Callers using providers that emit int8 directly (Cohere `int8` endpoints, OpenAI `text-embedding-3-large` reduced precision, Sentence Transformers with int8 quantization) skip a precision-losing client-side `int8 → float32 → server` round trip. The HTTP payload and document bucket storage shrink 4x.

The HNSW graph still runs on `float32` internally; the engine dequantizes on the read path using `value / 127.0f` (Cohere/OpenAI calibration). Native int8 HNSW is tracked upstream at datastax/jvector#665 — once the JVector contract widens, no schema change will be needed.

```
-- INT8-encoded ingest. Property MUST be BINARY.
CREATE PROPERTY Doc.embedding BINARY;
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1024,
  "similarity": "COSINE",
  "encoding": "INT8"
};
```

The factory rejects two misconfigurations at index-creation time so the failure surfaces immediately rather than as silent incorrect scaling later:

- `encoding=INT8` with a non-`BINARY` property (or `encoding=FLOAT32` with a `BINARY` property) — the property type and encoding must agree.
- `encoding=INT8` combined with `quantization=INT8` — redundant: the property is already byte-quantized at the wire level, so the index-internal scalar quantizer would re-quantize the dequantized floats. Pick one (`encoding=INT8` for payload/storage savings, `quantization=INT8` for index-internal compression), not both.

When a Cohere-style int8 source is **not** the origin of the bytes, note that Java's `byte` range `[-128, 127]` includes a value the Cohere/OpenAI calibration never emits. The dequantizer clamps `-128` up to `-127` to keep the result inside `[-1, 1]` for COSINE; a one-time WARNING is logged the first time a `-128` byte is encountered in a process so operators can investigate non-conforming sources.

6.4.7. Store Embeddings in an `EXTERNAL` Property

A vector embedding is typically the largest field on a record — a `FLOAT32` 1,536-dim embedding alone weighs 6 KB, an order of magnitude bigger than the topology (id, label, edge lists) it sits next to. Stored inline, every traversal, scan, or projection that does not touch the embedding still pages those 6 KB into the buffer cache. On a graph of a few million vertices that is enough to evict the useful topology working set long before the dataset is exhausted.

Declare the embedding property as `EXTERNAL true` so the bytes move to a paired **external bucket**. The main record then carries only an 8-byte pointer; the embedding is loaded lazily, on the queries that actually need it (vector search, similarity scoring, hybrid retrieval). See External property storage for the full description, including the per-bucket pairing, compression, and tiered-storage

options.

```
-- FLOAT32 embedding stored in a paired external bucket.
CREATE PROPERTY Doc.embedding ARRAY_OF_FLOATS (EXTERNAL true);
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1536,
  "similarity": "COSINE"
};

-- Same idea with INT8 wire/storage encoding -- still external, 4x smaller payload.
CREATE PROPERTY Doc.embedding BINARY (EXTERNAL true);
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1024,
  "similarity": "COSINE",
  "encoding": "INT8"
};
```

When you should opt in:

- The embedding is large compared to the rest of the record (any modern dense embedding qualifies).
- Most of your workload is not vector search — graph traversals, filters by other properties, or full-record reads that project everything except the embedding all benefit immediately.
- You plan to put the heavy payload on a different volume (e.g. embeddings on bulk storage, topology on NVMe) — set `arcadedb.externalPropertyBucketPath` before creating the property; see Tiered storage.

When inline storage is still the right default:

- Records are tiny and the embedding is your hottest field (every query reads it). The extra pointer lookup is a per-read cost that adds up.
- You are running a vector-only workload where every query projects `embedding`. There is nothing to save.

To migrate an existing populated type to external storage in one step, use `ALTER PROPERTY … EXTERNAL true` followed by `REBUILD TYPE`; see Migrating existing records.

6.4.8. Key Parameters

Parameter	Default	Purpose
`dimensions`	(required)	Must match your embedding model output size
`similarity`	`COSINE`	Distance metric: COSINE, DOT_PRODUCT, or EUCLIDEAN

Parameter	Default	Purpose
`encoding`	`FLOAT32`	Wire / document-storage encoding: FLOAT32 (`ARRAY_OF_FLOATS` property) or INT8 (`BINARY` property, byte per dim, 4x smaller payload). See Vector Encoding.
`quantization`	`NONE`	Index-internal compression: NONE, INT8, BINARY, or PRODUCT. INT8 is recommended for most use cases.
`efSearch`	sized from the graph	Search beam width at query time. Controls recall vs speed trade-off. See efSearch and the Default Beam.
`rerankOversample`	10	`PRODUCT` and `BINARY`: candidates per requested neighbor that a search re-ranks on the stored vectors, which makes the returned distances exact. `0` returns the PQ scores alone, without reading a stored vector (a `BINARY` index re-ranks at least one candidate per neighbor). A query overrides it with the `rerank` option of `vector.neighbors`. See [product-quantization-search] and [binary-quantization-search]. Since 26.11.1.
`pqTrainingLimit`	128000	`PRODUCT` only: vectors (a random sample) the PQ codebook is trained on, at most 128000.
`fusedPQ`	false	`PRODUCT` only: store the PQ codes of each node's neighbors in the graph file (jvector FusedPQ). Makes the graph file several times larger; useful only when the PQ codes cannot stay in memory. Since 26.11.1.
`maxConnections`	32	Graph connections per node, per layer (Vamana degree). Higher = better recall, more memory. This is **not** doubled at the base layer like hnswlib's `M`: to match an hnswlib `M`, set `maxConnections = 2 * M`. See note below.
`beamWidth`	100	Search depth during build. Higher = better index quality, slower builds
`addHierarchy`	false	Enable multi-layer HNSW for large/complex datasets
`storeVectorsInGraph`	false	Co-locate vectors in graph file for faster retrieval at large scale
`buildGraphNow`	true	Build the HNSW graph immediately at index creation time. Set to `false` for deferred (lazy) building on first search.

A rebuild of the graph is always a full one — there is no incremental update, so any rebuild reindexes every vector currently in the index, not just the ones written since the last build. `close()` normally rebuilds and persists the graph before shutting down, so a later reopen finds a ready-to-search graph rather than rebuilding on first open. For an index of 1,000 vectors or more (the same threshold mutation-triggered rebuilds use), a session that closes with unpersisted writes pending instead defers that rebuild rather than blocking `close()` on it: the vectors themselves are already durably persisted independently of the graph, so nothing is lost, but the graph's on-disk topology is left stale or absent. The deferred rebuild then runs the next time that index is actually searched, after any reopen. A session that closes and never searches the index again pays nothing at all. Below 1,000 vectors, `close()` keeps rebuilding synchronously as before.

That first query after a reopen does not wait for the rebuild. If the persisted graph is behind only because vectors were added after it was last built — and ArcadeDB can verify on disk that the graph still describes exactly the records it was built from — the graph is put straight back to work and only the missing vectors are searched separately, while the full rebuild runs in the background. Results are complete from the first query: a vector written after the last build is found whether it is already in the graph or still waiting to be folded into it. This applies to any session, including one that inserts before it searches.

Deleting a vector does not force a graph rebuild on the first query after the next restart, however large the index. Neither does updating one, which is a delete followed by an insert.

The graph records what its nodes stand for, in a small file stored beside it, so nothing has to be worked out again on open and nothing has to be rebuilt. A deleted vector becomes a node that searches skip. Those skipped nodes are removed later by the ordinary background rebuild, on the schedule and within the memory limits it already uses, rather than all at once on whichever query happens to run first.

An index created by an earlier version rebuilds once on its first open after the upgrade, writes that file, and is covered from then on. Nothing needs to be run by hand. Searches return the same results either way.

Indexes using `PRODUCT` quantization are the exception and still rebuild after a deletion, because their compressed vectors are tied to the same node numbering the rebuild reassigns.

`vectorIndex.maxConcurrentRebuilds` (1 by default) limits how many vector index graphs are rebuilt at the same time anywhere in the JVM, because concurrent rebuilds are what exhaust the heap. That limit covers the rebuilds ArcadeDB starts in the background and also a rebuild that a **query** has to run itself, because the index has no usable graph on disk — and that is the rebuild every index of a database runs at once, on the first query to reach each of them after a restart.

Such a query waits its turn like any other rebuild. The first query to need a graph of 1,000 vectors or more takes one of the `maxConcurrentRebuilds` slots and holds it for the build; the others wait, and answer as soon as a slot frees. Smaller indexes are exempt and keep building immediately, since they cannot threaten the heap and would only be delayed by whatever larger rebuild is running. Before the build starts, the page cache is asked to give up whatever heap the build needs, the same way a background rebuild does.

That exemption is itself bounded. "Too small to matter" describes one rebuild and says nothing about how many run at once. Exempt rebuilds draw on a shared allowance of 1,000 vectors per `maxConcurrentRebuilds` slot: while the allowance lasts they keep building immediately, and once it is spent the next one waits for a slot like a large rebuild, because together they have become one. A thousand indexes of a few vectors each therefore still cost nothing, while a handful close to the 1,000-vector line already count. The counter `smallRebuildsOverBudget` in the index statistics reports how often a small rebuild had to wait for this reason — distinct from `searchRebuildsQueuedForPermit`, which says some **other** index was holding the slot.

A query is not **refused** a rebuild for want of a slot, only delayed by one: with no graph there is nothing to search, and answering from the vectors written since the last graph alone would return an incomplete result presented as a complete one. If no slot frees within `vectorIndex.rebuildPermitTimeoutMs` (ten minutes by default), the query builds anyway and logs a warning, because by then the slot is being held by a rebuild that is stuck rather than slow.

The heap is a different matter. Before v26.12.1, a query-triggered rebuild that did not fit the heap allowed by `vectorIndex.rebuildMaxHeapPercent` — even after giving up the whole page cache — logged "It is built anyway" and built, and on a heap that full the build ended in an `OutOfMemoryError` that stopped every database of the server. Since v26.12.1 such a build is first made to fit: the page cache gives up what the build needs, and when that is not enough the build runs with a smaller build cache (never below 100,000 vectors, or the whole index if it is smaller), reading the rest of its vectors back from disk. The graph is the same, only the build is slower; it is counted as `searchRebuildsWithReducedCache`. Only when the graph itself does not fit is the query refused, with a `VectorGraphBuildRefusedException` that names the estimate and the setting. It is retryable (HTTP 503, and the equivalent on every other protocol): nothing was built, and the same query re-issued once the heap has room builds the graph and answers. Refusals are counted as `searchRebuildsRefusedForMemory`, and a warning is logged once until a build gets through. Setting `vectorIndex.rebuildMaxHeapPercent` to `0` disables the check, as it does for background rebuilds. The index statistic `lastGraphBuildCacheSize` reports the build cache the last build actually ran with.

Since v26.10.2 that wait no longer outlives the query. A vector search — `vector.neighbors()` in SQL, `db.index.vector.queryNodes` in Cypher, `select().nearestTo()` in the Java API — stops within a fraction of a second when its statement is cancelled (a client that disconnects, a gRPC deadline, a query terminated from the running queries list) or reaches its timeout

(`arcadedb.command.timeout`, a SQL `TIMEOUT` clause, `Select.timeout()`), whether it is waiting for a slot, waiting for a build another query started, or walking the graph. It fails with the usual timeout or termination error. The build itself is not abandoned: it runs in the background, keeps its slot until it finishes, and the next search uses the graph it publishes. A command timeout shorter than the build therefore delays the first successful search on that index until the build completes, but never prevents it.

A schema reload starts every index cold, and since v26.10.2 that no longer lands on the first search. A reload replaces every index with a freshly built instance with no location map and no graph resident, which on an HA follower happens for every schema entry its incremental refresh declines (see how-to/operations/ha.pdf). The vector indexes that had a graph resident before the reload are therefore warmed in the background, on one thread and one after the other, once the schema has been quiet for `vectorIndex.warmUpAfterSchemaReloadDelayMs` (1 s by default, `-1` to disable): the location map is read and the graph is loaded from its persisted copy, or rebuilt when that copy is stale, without a search having to ask. A burst of DDL statements costs one warm-up, not one per statement, a search that arrives meanwhile joins the load in progress, and an index nobody searched is left alone. The Java API exposes the same step as `LSMVectorIndex.warmUp()`.

The counters `searchRebuildsQueuedForPermit`, `searchRebuildsWithoutPermit`, `searchRebuildsWithReducedCache` and `searchRebuildsRefusedForMemory` in the index statistics report how often each happened. Persisting the graph before a restart — a clean `close()`, or `REBUILD INDEX` — avoids the whole situation: there is then nothing to rebuild on the first query.

A transaction that fails and is rolled back — most often with a concurrent-modification error, when two transactions write the same record — leaves the vector index exactly as it was.

Note also that rewriting a vector property with a value identical to the stored one never marks anything deleted: ArcadeDB compares the two embeddings and skips the index when they are equal. Avoiding redundant writes in your application therefore does not reduce rebuilds; reducing write conflicts does.

The same holds for sparse vector indexes (`LSM_SPARSE_VECTOR`): a sparse vector index holds the entries of a transaction back until the transaction has actually committed, so a rollback has nothing to undo.

Searching a vector index from inside an open transaction is safe: what the index keeps afterwards is built only from committed data.

A search issued inside a transaction also finds the rows that transaction has written but not yet committed, for both dense (`LSM_VECTOR`) and sparse (`LSM_SPARSE_VECTOR`) indexes. You can insert or re-embed rows and search them in the same unit of work.

Full-text indexes merge their own transaction's writes the same way.

The two guarantees hold together: what the search **returns** includes your uncommitted rows, while what the index **stores** is still built only from committed data. Roll the transaction back and both the rows and their embeddings are gone, with nothing left behind in the index.

Rewriting one record's embedding several times inside a single transaction leaves the index holding the **last** value written.

Vectors written since the last rebuild are searched by scanning them one by one, so that part of a query grows with how many are waiting, while the graph search it supplements grows only with the logarithm of the index size.

A rebuild is also triggered when the scan grows past what the graph search itself costs — measured, from the nodes searches actually visit, not estimated — and once those scans have cost as much as the rebuild will. The second condition is what keeps the first affordable: the extra rebuild work this can introduce is bounded by the query work it removes. Because it is evaluated when queries run, a write-only workload is unaffected and keeps the existing rebuild schedule.

If you are upgrading a deployment that both writes and queries heavily, expect more background rebuild activity than before — that is the fix doing its job, and it is the same deployment that was paying for the scan. Set `vectorIndex.maxDeltaScanRatio` to 0 to restore the previous behaviour exactly, lower it to keep queries faster at the cost of more rebuilds, or raise it for the opposite trade.

Opening a database does not read a vector index at all. Both structures the index keeps in memory — the graph and the map of where each vector lives — are built the first time something actually uses that index: a search, an insert, a count. Before this, every open rebuilt the location map by reading through every page of every `LSM_VECTOR` index, which took time proportional to the number of indexed vectors (around 1.4 seconds for 10 million) and was paid again on each open, even by a process that only opened the database to write one document or to read an unrelated type.

Results, counts and stats are unchanged — only the timing is. The first operation that touches an index pays the load once; on a session that searches, that was already the smaller half of the work, since loading or rebuilding the graph dominates it. A session that never touches a vector index never pays for it.

The first search after a reopen no longer re-reads the document of every indexed vector to rebuild the graph's ordinal map, which took about 2.8 seconds at 1 million vectors and 18 seconds at 10 million. When nothing was deleted and the persisted graph still matches the index, the map saved next to the graph is verified and reused instead. Indexes with deletions or vectors added since the graph was saved use the previous paths.

Since v26.11.1 this applies to `PRODUCT`-quantized indexes too, provided their `.vecpq` file records which graph its PQ codes were built for. A `.vecpq` file written before v26.11.1 records none, so such an index keeps the previous path until its graph is next rebuilt; `REBUILD INDEX` does it at once. A `PRODUCT` index with deletions always uses the previous path, because its PQ codes need a dense ordinal space.

To pay the load at startup instead of on the first query, call `warmUp()` on the `LSMVectorIndex` (embedded API), or run one throwaway search. The `graphLoadsFromOrdinalMap` index stat counts how many loads took the fast path.

The location map is also where a vector index learns which vector ids are already taken, so a load that does not read every entry cannot safely give out new ones. Before v26.10.2 such a load was kept for the life of the process anyway, and the inserts that followed could take ids that unread entries already held: two records then shared one vector id, and one of them dropped out of search results (`CHECK DATABASE` reports it as a vector id that a load gives to another record).

Since v26.10.2 a load stopped from outside — an `OutOfMemoryError`, an interrupted thread, a database closing under it — is discarded and done again on the next access. A load that finds a page it cannot read keeps serving searches from what it read, logs a `SEVERE` message, and refuses new vectors (an insert, or an update that changes a vector) until the index is rebuilt; deleting stays possible. `CHECK DATABASE` lists such an index among the corrupted ones, so `CHECK DATABASE FIX` (or `REBUILD INDEX <name>`) rebuilds it and lifts the refusal.

6.4.9. efSearch and the Default Beam

The `efSearch` parameter is the beam width of the graph search: how many candidate nodes the search keeps while it walks the vector graph. Higher values find more accurate results but take longer. The beam is never narrower than `k`.

When `efSearch` is not set (neither on the index nor per query), ArcadeDB sizes the beam from the graph and runs the search once:

1. For a graph under 10,000 nodes the beam is `max(k, 100)`.
2. For a larger graph the beam is `max(2 × k, 100)`. Before 26.11.1 it was `max(2 × k, 20)`, so a 10-neighbor query on a large graph walked a beam of 20 and lost recall.

There is no second pass that widens the beam when a search comes up short: a wider beam would visit the same nodes. A search that still returns fewer than `k` results is topped up by a brute-force scan instead. The default beam does not adapt to hard queries. If recall matters, set `efSearch` explicitly (see below): a larger value trades latency for recall.

Setting efSearch

You can set efSearch at three levels:

Per-query (highest priority) — pass as the 4th argument to `vector.neighbors()`, either positionally or via the named options map:

```
-- Higher efSearch for a critical query that needs maximum recall (positional)
SELECT expand(vector.neighbors('Doc[embedding]', [...], 10, 500))

-- Lower efSearch for a latency-sensitive query (positional)
SELECT expand(vector.neighbors('Doc[embedding]', [...], 10, 30))

-- Options map form (extensible; also supports `filter`)
SELECT expand(vector.neighbors('Doc[embedding]', [...], 10, { efSearch: 500 }))
```

Per-index — set in the index metadata at creation time:

```
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  dimensions: 1024,
  similarity: 'COSINE',
  efSearch: 200
}
```

Default beam — when neither per-query nor per-index efSearch is specified, the beam is sized from the graph as described above.

The default beam suits most workloads. If recall matters, set efSearch explicitly and raise it until the recall you measure on your own queries is enough. Set it lower only if you have strict latency requirements. On a version before 26.11.1, the default beam of a graph of 10,000 nodes or more is narrow, so set efSearch there when recall matters.

6.4.10. Filtered Search

Vector search can be combined with a logical filter on the same type by passing a `filter` option containing the allowed RIDs. The HNSW traversal restricts itself to that set, so non-matching vectors are skipped without decoding.

```
-- Find the 10 most similar documents within a specific tenant and category
SELECT vector.neighbors(
         'Document[embedding]',
         :queryVector,
         10,
         { filter: (SELECT @rid FROM Document WHERE tenantId = 'acme' AND category =
'finance') }
       )
```

The `filter` value accepts a list of RIDs, RID strings, or any `Identifiable`. It can be produced by a subquery, a query parameter, or built programmatically. A subquery may project the RID (`SELECT @rid FROM …`), the whole record (`SELECT FROM …`), or a list of RIDs that is flattened one level (`(SELECT list(@rid) AS l FROM …).l`); a `LET` variable bound to such a subquery works the same way. Projecting anything that is not a RID raises an error rather than silently filtering nothing.

A `filter` narrower than `vectorIndex.prefilterMaxSelectivity` (20% of the index's live vectors by default) bypasses the HNSW walk automatically: ArcadeDB scores the filtered candidates directly instead, which gets faster as the filter narrows rather than slower. Above that threshold, a selective filter can still starve the beam trying to fill `k` from a narrower-than-expected pool; combine `filter` with a higher `efSearch` to preserve recall in that range, or lower `vectorIndex.prefilterMaxSelectivity` to route more filters through the direct-scoring path. See Settings Reference.

Partition-aware filterable HNSW

When the type uses `partitioned(<key>)` as its bucket selection strategy and the query's `WHERE` clause binds the partition key to a literal, the planner narrows `vector.neighbors` / `vector.sparseNeighbors` to a single per-bucket HNSW graph automatically — no `filter` option needed:

```
SELECT vector.neighbors('Document[embedding]', :queryVector, 10) AS neighbors
  FROM Document
 WHERE tenant_id = 'acme'
```

With the partition strategy in place, this query touches only `tenant_id = 'acme'`s HNSW graph instead of fanning out across every bucket. The optimisation kicks in transparently for both dense (`vector.neighbors`) and sparse (`vector.sparseNeighbors`) search and composes with explicit `filter` lists when both are present.

The two approaches are complementary:

- `partitioned(<key>)` + a `WHERE` predicate on the key prunes the **bucket set** before the HNSW traversal even starts. Best for high-cardinality scope keys (tenant, customer, region) where every query naturally carries the predicate.
- `filter: (SELECT @rid ⋯)` prunes **individual RIDs** during the HNSW traversal. Best for low-cardinality or query-shaped filters that don't map cleanly onto a partition key.

For the multi-tenant SaaS case, partition-aware HNSW is usually the right primitive. See Schema design 101: choosing a bucket strategy for picking the partition key.

6.4.11. Multi-Modal Search

A single vertex type can have multiple vector indexes on different properties:

```
CREATE INDEX ON Product (imageEmbedding) LSM_VECTOR METADATA {dimensions: 512,
similarity: 'COSINE'}
CREATE INDEX ON Product (textEmbedding)  LSM_VECTOR METADATA {dimensions: 768,
similarity: 'COSINE'}
```

Query each index independently to search by image similarity, text similarity, or combine scores.

6.4.12. Integration with Other Models

Vector search combines naturally with ArcadeDB's other data models:

- **Graph + Vectors** — Find similar items, then traverse relationships to discover connected context (Graph RAG pattern)
- **Full-text + Vectors** — Hybrid search combining keyword matching with semantic similarity (Knowledge Graph pattern)
- **Time Series + Vectors** — Detect behavioral anomalies by comparing embedding patterns over time (Fraud Detection pattern)

6.4.13. SQL Example

Create a vector index and query it:

```
-- Create vertex type and property
CREATE VERTEX TYPE Document;
CREATE PROPERTY Document.content STRING;
CREATE PROPERTY Document.embedding ARRAY_OF_FLOATS;

-- Create vector index with 384 dimensions using COSINE similarity
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  dimensions: 384,
  similarity: 'COSINE'
};

-- Query for the 10 nearest documents
-- Returns rows with the document's properties, @rid, @type and .distance (0 =
identical for COSINE);
-- the full document is reachable as .record
SELECT expand(vector.neighbors('Document[embedding]', :queryVector, 10))
```

6.4.14. Java Example

Create and query a vector index programmatically:

```
import com.arcadedb.index.lsm.LSMVectorIndex;
import com.arcadedb.index.lsm.LSMVectorIndexBuilder;
import com.arcadedb.index.vector.VectorSimilarityFunction;

// Create index programmatically
final LSMVectorIndexBuilder builder = new LSMVectorIndexBuilder(
    database,
    "Document",
    new String[]{"embedding"})
    .withDimensions(384)
    .withSimilarity(VectorSimilarityFunction.COSINE)
    .withMaxConnections(16)
```

```
    .withBeamWidth(100);

final LSMVectorIndex index = builder.create();

// Query the index using SQL
final ResultSet resultSet = database.query("sql",
    "SELECT expand(vector.neighbors('Document[embedding]', ?, 10))",
    queryVector);
```

6.4.15. Configuration Parameters

When creating LSMVectorIndex instances, the following parameters can be configured:

- `dimensions`: The dimensionality of the vectors (must match your embedding model output)
- `similarity`: The distance function for similarity calculation (COSINE, DOT_PRODUCT, EUCLIDEAN, etc.)
- `maxConnections`: Maximum number of graph connections per node, per layer (default: 32, increase for better recall)
- `beamWidth`: Beam width for approximate nearest neighbor search (default: 100, increase for more accurate results)

`maxConnections` is the Vamana per-layer graph degree used by the underlying engine. It is applied to every layer, **including the base layer** — it is **not** doubled at the base layer the way hnswlib (used by Chroma and most HNSW tools) treats its `M` parameter, which allocates `2*M` links at the base layer.

If you are porting a configuration from an hnswlib-based system, set `maxConnections = 2 * M` to get the same graph density (for example, hnswlib `M=16` corresponds to `maxConnections=32`). Leaving it at half the intended value silently halves base-layer density and lowers recall. The default was raised from 16 to 32 to match hnswlib `M=16` out of the box; the extra density costs proportionally more build-time heap.

6.4.16. Supported Similarity Functions

Measure	Name	Type
`COSINE`	Cosine Similarity	L_2
`DOT_PRODUCT`	Inner Product	L_2
`EUCLIDEAN`	Euclidean Distance	L_2

For more information on vector embeddings, see the Vector Embeddings section.

6.4.17. Sparse Vector Search

In addition to dense embeddings, ArcadeDB supports **sparse** vector retrieval via the `LSM_SPARSE_VECTOR` index. Sparse vectors carry only the non-zero dimensions of a high-dimensional

space and are produced by learned-sparse retrieval models such as SPLADE, BGE-M3, OpenSearch's `opensearch-neural-sparse-encoding-multilingual-v1`, and BM25-as-sparse-vector pipelines.

Looking for BM25 keyword ranking? That is provided by the `FULL_TEXT` index, which uses BM25 by default, see BM25 at a glance. Fuse both with dense vectors using `vector.fuse`.

Sparse retrieval excels where dense embeddings collapse semantically distinct queries into the same point ("freeze card after theft" vs "unfreeze card after travel"), because the per-token weights expose the discriminating terms directly.

Storage layout

The index uses a posting-list inverted structure on the LSM-Tree backbone with composite key `(int dim_id, RID rid, float weight)`. It inherits ACID transactions, WAL, replication, and compaction from the LSM-Tree, just like every other ArcadeDB index. Per-document data lives in two parallel array properties:

- an `ARRAY_OF_INTEGERS` of non-zero dimension ids, and
- an `ARRAY_OF_FLOATS` of the matching weights.

Both arrays must have the same length on every record. Weights must be non-negative; negative weights would corrupt the per-term upper bound used to prune during retrieval and are rejected at write time. All standard sparse models (BM25, SPLADE, BGE-M3, Cohere sparse) emit non-negative weights, so this is not a real constraint in practice.

Dimension ids must be whole numbers in the integer range (`0` to `2147483647`). A declared `ARRAY_OF_INTEGERS` property converts what it is given on write, as every integer property does, so `[0, 2.5]` is stored and indexed as `[0, 2]`. A record stored before its property was declared keeps the list it was given: the index refuses such a record when its list holds a fractional id (`2.5`) or one past the integer range (`5000000000`), naming the record and the property, so building the index over existing rows fails on it instead of indexing a dimension the record does not hold. `2.0` is the dimension `2` and is accepted.

Up to v26.10.2 the index truncated such an id (`2.5` became dimension `2`, `5000000000` wrapped to `705032704`). Creating or rebuilding a sparse vector index over a type holding rows written that way now fails, naming the first such record: correct its list (or remove it) and run the build again. Deleting or updating a record indexed the old way keeps working, and retracts the postings it was indexed with.

Schema setup

```
CREATE DOCUMENT TYPE Doc;
CREATE PROPERTY Doc.tokens  ARRAY_OF_INTEGERS;
CREATE PROPERTY Doc.weights ARRAY_OF_FLOATS;

-- LSM_SPARSE_VECTOR index. `dimensions` declares the vocabulary cap (0 = open-ended).
-- `modifier: 'IDF'` enables Robertson-Sparck-Jones IDF weighting at query time.
```

```
CREATE INDEX ON Doc (tokens, weights) LSM_SPARSE_VECTOR
  METADATA { "dimensions": 105000, "modifier": "IDF" };
```

Querying

Use `vector.sparseNeighbors` to retrieve top-K records by sparse dot product:

```
SELECT expand(`vector.sparseNeighbors`(
    'Doc[tokens,weights]',
    :queryIndices, :queryValues,
    50
))
```

Options map (5th argument) supports `filter` (allowed-RIDs whitelist), `groupBy`, `groupSize` — see Group-By Retrieval below.

Top-K algorithm

Retrieval uses document-at-a-time Block-Max MaxScore. Query terms are split into an **essential** set and a **non-essential** one: a document matching only non-essential terms cannot reach the current K-th best score, so those terms stop generating candidates and are only point-probed for documents the essential terms produce. Per-block maximum weights then let whole ranges of postings be skipped without decoding them. Only dot-product similarity is supported (cosine on sparse vectors is conventionally handled by L2-normalizing both the query and stored vectors at insert time).

Earlier releases used pivot-based WAND. It was replaced because learned-sparse queries carry many terms with relatively flat weights, which keeps every term in the traversal and ends up decoding nearly every block. MaxScore prunes on the term side instead, which is the dimension that discriminates on this kind of data.

Large indexes additionally split a query across RID ranges scored in parallel, which is exact — it changes latency, not results.

Posting-weight quantization is configurable via the `weightQuantization` index `METADATA` key (`INT8` — the default — `FP16`, or `FP32`), trading index size against the precision of the stored weights. The quantized weights only select candidates: a search rescores `k * rescoreOversample` of them exactly from the records' own weights (default `2` for `INT8` and `FP16`), so the returned ranking and scores are exact and do not depend on how the data was committed or compacted. See `CREATE INDEX` for the values and their trade-offs.

Durability of recent writes

New postings are kept in memory and sealed into a segment once about one million accumulate, or when the database closes. Since v26.11.1 each sparse index also writes them, in the same transaction as the records, to a small posting log (two files per index, `<index>_plog0` and `<index>_plog1`, extension `.sparselog`), so after a crash they are replayed and a committed record is always found by `vector.sparseNeighbors`. Before, the postings not yet sealed were lost by a crash until a `REBUILD INDEX`. The log costs roughly 5-10% of write time on a sparse-heavy workload.

On an HA cluster the leader writes the same log, and every server fills its in-memory postings from the log pages it receives. So every server answers sparse searches with the latest writes without waiting for a flush, and a new leader starts with the postings the previous one had not sealed. An index created on a cluster before v26.11.1 gets its posting log with a `REBUILD INDEX`.

Segment compaction and deletes

Sealed segments are merged in the background by a size-tiered policy: segments of comparable posting count that sit next to each other in write order are merged into one, which keeps the per-query merge fan-in bounded without rewriting the whole index on every flush.

A delete is recorded as a **tombstone** in the segment that is current when it happens, not in the segment holding the posting it removes, so which of two segments is newer is what decides whether a document is visible. Compaction preserves that order: a merged segment takes the position of its newest input rather than moving to the front, and a partial merge only ever takes segments that are adjacent in write order.

A query validates the segment data it reads as it goes. A damaged segment is reported as an error naming the index and segment rather than silently producing wrong results; the fix is `REBUILD INDEX` on the named index.

Databases created by an earlier version may hold sparse-vector segments that were merged under the previous ordering, which could let a partial compaction outrank the segment holding a tombstone and make a deleted document reappear in `vector.sparseNeighbors` results (issue #6379). Upgrading fixes the ordering for all new merges, but a segment already written under the old rule cannot be repaired automatically — the information about where it should have sat is gone.

The server identifies such segments when the index is opened and logs a warning naming the index and the `REBUILD INDEX` to run. The marker is stored in the segment and inherited by any later merge, so it survives the affected segments being compacted away by ordinary write traffic. If a database was created by an earlier version and its sparse-vector index has seen deletes or updates, run once per affected index:

```
REBUILD INDEX `Doc[tokens,weights]`
```

This collapses the existing segments into a single correctly ordered segment. Indexes that have only ever been appended to are unaffected.

This is also queryable at any time, not only visible as a one-time log line at startup: `schema:indexes` (and Studio's index view) report the affected index in `upgradeWarning`, and its stats include an `untrustedSegments` count:

```
SELECT name, upgradeWarning FROM schema:indexes WHERE upgradeWarning IS
NOT NULL

SELECT untrustedSegments FROM (SELECT expand(stats) FROM schema:indexes
```

```
WHERE name = 'Doc[tokens,weights]')
```

`untrustedSegments` drops back to `0` once `REBUILD INDEX` has been run, so it can also be used to confirm the fix took effect.

6.4.18. Hybrid Search with `vector.fuse`

`vector.fuse` combines two or more ranked sub-pipelines into a single ranked top-K, server-side, in one query. It generalises the dense+sparse hybrid pattern but works with any source that yields `(@rid, $score)` rows: dense `vector.neighbors`, sparse `vector.sparseNeighbors`, full-text `SEARCH_INDEX`, or any plain `SELECT ⋯ ORDER BY ⋯ LIMIT N`.

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Doc[dense]', :denseVec, 50),
    `vector.sparseNeighbors`('Doc[tokens,weights]', :qIdx, :qVal, 50),
    { fusion: 'RRF', groupBy: 'source_file', groupSize: 1 }
)) LIMIT 10
```

Fusion strategies

`vector.fuse` supports three strategies via the `fusion` option:

Strategy	When to use
`RRF` (default)	Reciprocal Rank Fusion. `score = sum_i weight_i / (k + rank_i)`. Rank-only, indifferent to score scales — best when sources have wildly different score distributions (e.g. cosine in [-1, 1] plus BM25 in [0, +inf)). Default `k = 60`.
`DBSF`	Distribution-Based Score Fusion (Qdrant 1.11+). Per-source scores normalised to [mean - 3sigma, mean + 3sigma], then weighted sum. Useful when sources produce comparable, roughly Gaussian distributions and you want score magnitude (not just rank) to influence fusion.
`LINEAR`	Per-source min-max normalisation, then weighted sum. Use when you have already-tuned weights from offline relevance experiments. Requires every source row to carry a numeric score.

`vector.fuse` assumes "higher = better" for every source. It auto-flips the `distance` field exposed by `vector.neighbors` to a similarity at extract time so dense and sparse sources fuse without manual rescaling. If you build a custom source via `SELECT @rid, <expr> AS score FROM ⋯`, expose `score` with similarity semantics for `LINEAR` and `DBSF` to behave correctly. `RRF` is rank-only so it tolerates either convention.

Fused rows come back by descending fused score; rows with exactly the same fused score (common with `RRF`, when two documents swap ranks across two sources) are ordered by ascending RID, so the order is stable from one run to the next. Before 26.12.1 their order was arbitrary.

Weighted fusion

Per-source weights tilt the fused ranking:

```
-- Dense matters 3x more than sparse for this corpus.
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Doc[dense]', :denseVec, 50),
    `vector.sparseNeighbors`('Doc[tokens,weights]', :qIdx, :qVal, 50),
    { fusion: 'RRF', weights: [3.0, 1.0] }
)) LIMIT 10
```

Three-way fusion (dense + sparse + full-text)

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Doc[dense]', :denseVec, 100),
    `vector.sparseNeighbors`('Doc[tokens,weights]', :qIdx, :qVal, 100),
    (SELECT @rid, $score FROM Doc WHERE SEARCH_INDEX('Doc[content]', :keywords) = true
LIMIT 100),
    { fusion: 'RRF', k: 60 }
)) LIMIT 10
```

Bounding the cost of a full-text leg

`vector.fuse` fuses every row its sources produce and, without a `limit` option, loads every fused candidate. A dense or sparse leg is bounded by its `k`. A full-text leg without a `LIMIT` is not: it yields one row per match, a common keyword in a large corpus can match thousands of rows, and the query then slows down with the match count instead of with `k`. The same holds when the fused rows feed `vector.rerank`.

Bound the full-text leg with a `LIMIT` and the fusion with `limit`:

```
SELECT expand(`vector.rerank`(
    `vector.fuse`(
        `vector.neighbors`('Doc[dense]', :denseVec, :fetch),
        (SELECT @rid, $score FROM Doc WHERE SEARCH_INDEX('Doc[content]', :keywords) =
true LIMIT :fetch),
        { fusion: 'RRF', limit: :candidates }
    ),
    :denseVec, 'dense', :k
))
```

`:fetch` is the number of rows each leg contributes, `:candidates` the number of fused rows handed to `vector.rerank`, and `:k` the number of results (for example 100, 50 and 10). A `SEARCH_INDEX` subquery without an `ORDER BY` returns its matches by descending score, so its `LIMIT` keeps the best-scoring ones. Bounding it costs no relevance.

Since 26.11.1, `SEARCH_INDEX(…)` can also be passed directly as a source, as in

`vector.fuse(vector.neighbors(…), SEARCH_INDEX('Doc[content]', :keywords), { fusion: 'LINEAR' })`. It carries the score of each match, so `DBSF` and `LINEAR` accept it. It cannot take a `LIMIT`, though, and every match gets fused: prefer the subquery when the match count can be large. In the same release `vector.rerank` stopped reloading records that `vector.fuse` and `vector.neighbors` rows already carry, and now keeps only the best `k` while scanning.

Outer composition

The fused output is itself a result set, so it composes with outer `WHERE`, `ORDER BY`, `LIMIT`, joins, and projections:

```
-- Score threshold + projection.
SELECT title, source_file, score
FROM (
  SELECT expand(`vector.fuse`(
      `vector.neighbors`('Doc[dense]', :denseVec, 50),
      `vector.sparseNeighbors`('Doc[tokens,weights]', :qIdx, :qVal, 50),
      { fusion: 'RRF' }
  ))
)
WHERE score >= 0.02
ORDER BY score DESC
LIMIT 10
```

6.4.19. Group-By Retrieval

`vector.neighbors`, `vector.sparseNeighbors`, and `vector.fuse` all accept `groupBy` and `groupSize` options for diversification at retrieval time, mirroring Qdrant's `query_points_groups` semantics. Common pattern: best chunk per source document, deduped at the index level rather than over-fetched and post-partitioned in the application.

```
-- Top 10 distinct source files, best matching chunk from each.
SELECT expand(`vector.neighbors`(
    'Doc[embedding]', :queryVec, 10,
    { groupBy: 'source_file', groupSize: 1 }
))
```

The third positional argument (`limit` / `k`) becomes the **max number of distinct groups** when `groupBy` is set. Total returned rows are bounded by `limit * groupSize`. `groupSize` defaults to `1`.

`groupBy` accepts dotted nested-field paths — `metadata.author`, `provenance.source.id`, etc. Each segment after the first descends one level via `Map`-typed properties or embedded documents; a missing segment lands the row in the `null` group.

`groupBy` composes with the existing `filter` option:

```
SELECT expand(`vector.neighbors`(
```

```
    'Doc[embedding]', :queryVec, 10,
    { filter: [#5:0, #5:1, #5:2], groupBy: 'source_file', groupSize: 1 }
))
```

Grouping is integrated into the index traversal proper, not applied as a post-filter on an over-fetched candidate pool:

- **Sparse** (`vector.sparseNeighbors`) — the Block-Max MaxScore loop runs with a per-group min-heap. A new group opens only while the heap holds fewer than `limit` distinct keys; within each group, a candidate replaces the group's worst member when its score is higher. The pruning threshold tightens to the global per-group worst score once every group has reached `groupSize`, so whole posting-list regions that cannot beat any group's current worst member are skipped. `allowedRIDs` is applied inline — no over-fetch. When a group only earns its place after some of its members were already passed over, a second pass restricted to that group fills it, so every returned group holds its `groupSize` best members.
- **Dense** (`vector.neighbors`) — the HNSW traversal admits on liveness (and, with `filter`, allow-list membership) only; the group cap is applied afterward to the walk's score-ordered output, so the `groupSize` members kept per group are genuinely its best and the `limit` groups kept are genuinely the nearest (an earlier design gated group membership inside the traversal's `Bits` filter, but `Bits` is score-blind, so groups were admitted in visit order and the truly nearest group could be locked out of its own answer — this was replaced by the current score-ordered cap). The search budget is sized to `limit * groupSize`, and the walk resumes automatically — no re-query needed — while more distinct groups are still wanted and the graph has more to give; raise `efSearch` for wider coverage if a query still comes up short of `limit` groups. When `filter` is present and narrower than `vectorIndex.prefilterMaxSelectivity` (20% of the index by default), the HNSW walk is skipped entirely: the filtered candidates are scored directly and the same group cap applied to that output.

The answer is always the `limit` groups whose best member ranks highest, each with its `groupSize` best members. This also holds for a type with several buckets (each bucket is searched separately and a bucket is asked again for the members of a winning group it did not return) and inside a transaction, where the answer includes the transaction's own uncommitted records exactly as it will after the commit.

Pathological combinations of `limit` and `groupSize` (e.g. `limit=1000, groupSize=1000` would size a per-group state past the cap) are still rejected with an explicit `over-fetch budget exceeded` error to bound memory.

6.4.20. Further Reading

- Vector Search Tutorial — Step-by-step hands-on guide
- Vector Embeddings How-To — Index creation, tuning, and best practices
- Java Vector API — Programmatic vector index management
- SQL Vector Functions — All 40+ vector SQL functions
- Sparse Vector Search and Hybrid Search with `vector.fuse` — This section

6.5. Geospatial Index

ArcadeDB's geospatial index enables spatial queries directly from SQL using the `geo.*` function namespace. Geometries are stored as Well-Known Text (WKT) strings in regular `STRING` properties — no special column type is required.

The index is built on ArcadeDB's LSM-Tree storage engine, which means ACID guarantees, write-ahead logging, high availability replication, and automatic compaction all apply without any extra configuration. Internally, the index decomposes each geometry into GeoHash tokens using a `RecursivePrefixTreeStrategy`, stores each token in the underlying LSM-Tree, and applies an exact Spatial4j predicate as a post-filter when queries are evaluated.

6.5.1. Creating a Geospatial Index

Create a `STRING` property, then add a `GEOSPATIAL` index to it:

```
CREATE VERTEX TYPE Location
CREATE PROPERTY Location.coords STRING
CREATE INDEX ON Location (coords) GEOSPATIAL
```

The index type keyword is `GEOSPATIAL`. Only `STRING` properties are supported (WKT values are stored as strings).

Configuring Precision

The precision level controls the GeoHash grid resolution used to decompose geometries. Higher precision means more tokens per geometry, finer resolution, and a larger index.

```
CREATE INDEX ON Location (coords) GEOSPATIAL
  METADATA { "precision": 9 }
```

Precision	Cell size (approx.)	Typical use
5	~4.9 km × 4.9 km	Country / region level
7	~153 m × 153 m	City / neighborhood level
9	~4.8 m × 4.8 m	Street-address precision
11 (default)	~2.4 m × 2.4 m	High precision — building footprints, GPS tracks

Changing precision after data has been indexed requires rebuilding the index with `REBUILD INDEX`.

6.5.2. Inserting Geometries

Store any valid WKT string in the indexed property. Pass it as a plain string literal or use the `geo.asText()` function to convert a geometry object back to WKT:

```
-- Insert a point by WKT literal
INSERT INTO Location SET name = 'Eiffel Tower', coords = 'POINT(2.2945 48.8584)'

-- Insert a polygon by WKT literal
INSERT INTO Location SET name = 'Paris Centre', coords =
  'POLYGON((2.2945 48.8584, 2.3522 48.8566, 2.3488 48.8791, 2.2945 48.8584))'

-- Use geo.asText() to store a constructed geometry
INSERT INTO Location SET name = 'Arc de Triomphe',
  coords = geo.asText(geo.point(2.2950, 48.8738))
```

Records with null or unparsable WKT values in the indexed property are silently skipped during indexing.

6.5.3. Querying with Spatial Predicates

Use any `geo.*` predicate in a `WHERE` clause. When a geospatial index exists on the referenced field, the query planner automatically uses it — no hint or special syntax is required.

```
-- All locations within a bounding polygon
SELECT name FROM Location
WHERE geo.within(coords, geo.geomFromText('POLYGON((2.28 48.85, 2.40 48.85, 2.40
48.89, 2.28 48.89, 2.28 48.85))')) = true

-- All locations that intersect a line
SELECT name FROM Location
WHERE geo.intersects(coords, geo.geomFromText('LINESTRING(2.29 48.85, 2.35 48.87)')) =
true

-- All locations within 500 m of a point (full-scan fallback — see note)
SELECT name FROM Location
WHERE geo.dWithin(coords, geo.geomFromText('POINT(2.2945 48.8584)'), 0.0045) = true
```

`geo.dWithin` and `geo.disjoint` always perform a full scan because their semantics cannot be efficiently mapped to the GeoHash candidate set. All other predicates use the geospatial index automatically.

6.5.4. How Index-Accelerated Queries Work

1. The SQL query planner detects a spatial predicate function implementing `IndexableSQLFunction` in the `WHERE` clause.
2. It calls `allowsIndexedExecution()`, which returns `true` when the first argument is a bare field reference and a `GEOSPATIAL` index exists on that field.
3. The index decomposes the query shape into GeoHash cells and looks up candidate record IDs in the LSM-Tree.
4. The predicate function's exact Spatial4j implementation post-filters the candidates to remove

false positives (GeoHash cells are rectangular approximations).

The result is correct — the index is a superset filter, and the exact predicate is always applied afterward.

6.5.5. geo.* Functions Reference

Geospatial functions are organised into two groups: **constructor / accessor functions** (pure computation, no index) and **spatial predicate functions** (used in `WHERE` clauses, index-accelerated where possible).

All functions accept either WKT strings or geometry objects produced by other `geo.*` functions. All predicate functions return `null` when either argument is `null` (SQL three-valued logic).

Constructor and Accessor Functions

geo.geomFromText()

Parses any WKT string into a geometry object that can be passed to other `geo.*` functions.

Syntax: `geo.geomFromText(<wkt-string>)`

```
SELECT geo.geomFromText('POINT(2.2945 48.8584)') AS geom
SELECT geo.geomFromText('POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))') AS geom
```

Throws `IllegalArgumentException` for malformed WKT.

---

geo.point()

Creates a 2D point from longitude and latitude (or X and Y) coordinates.

Syntax: `geo.point(<x>, <y>)`

```
SELECT geo.point(2.2945, 48.8584) AS pt
```

---

geo.lineString()

Creates a line string from a list of points.

Syntax: `geo.lineString([<point>*])`

```
SELECT geo.lineString([geo.point(0, 0), geo.point(10, 10), geo.point(20, 0)]) AS line
```

---

geo.polygon()

Creates a polygon from an ordered list of points. The first and last point must be identical to close the ring.

Syntax: `geo.polygon([<point>*])`

```
SELECT geo.polygon([
  geo.point(0, 0), geo.point(10, 0),
  geo.point(10, 10), geo.point(0, 10),
  geo.point(0, 0)
]) AS poly
```

---

geo.buffer()

Returns a new geometry that is a buffer of the given distance (in degrees) around the input geometry.

Syntax: `geo.buffer(<geometry>, <distance-in-degrees>)`

```
-- Buffer 0.01 degrees (~1.1 km) around a point
SELECT geo.buffer(geo.point(2.2945, 48.8584), 0.01) AS buffered
```

Distance is expressed in degrees. To approximate metres, divide by 111,000 (1 degree ≈ 111 km at the equator).

---

geo.envelope()

Returns the minimum bounding rectangle (envelope) of a geometry as a polygon WKT. A bounding rectangle that crosses the antimeridian is returned as a `MULTIPOLYGON` split at 180, e.g. `MULTIPOLYGON ( 170 0, 180 0, 180 10, 170 10, 170 0, -180 0, -170 0, -170 10, -180 10, -180 0)`, so every `geo.*` function reads it back as that rectangle and not as the band between its two meridians. `geo.asGeoJson()` of an `ENVELOPE` that crosses the antimeridian is likewise a `MultiPolygon` split at 180. The envelope of an empty geometry (e.g. `POINT EMPTY`) is `POLYGON EMPTY`, as in PostGIS. *(Since v26.12.1; before, it was a polygon of NaN coordinates)*

Syntax: `geo.envelope(<geometry>)`

```
SELECT geo.envelope(geo.geomFromText('LINESTRING(0 0, 10 5, 20 0)')) AS bbox
```

---

geo.distance()

Returns the great-circle distance between two geometries, using the Haversine formula. For two points it is the distance between them; for any other pair of shapes it is the distance between their two closest points, and 0 when they intersect (a point inside a polygon is at distance 0 from it). Two shapes facing each other across the antimeridian are measured the short way round. The closest points of two shapes that are not points are found on the longitude/latitude plane and then measured on the sphere, so the result is close to, but not always equal to, the shortest distance on the sphere: the plane picks the nearest edge in degrees, and a degree of longitude is shorter the nearer the pole. For two points it is exact. When either geometry is empty (e.g. `POINT EMPTY`) the result is `NULL`, as in PostGIS, and `geo.dWithin()` answers `false`. *(Since v26.12.1; before, it was NaN)*

Syntax: `geo.distance(<geometry1>, <geometry2> [, <unit>])`

The optional `unit` parameter accepts `'m'` (default), `'km'`, `'mi'` or `'nmi'`.

Before 26.11.1 a shape that was not a point was measured from the centre of its bounding box.

```
-- Distance in kilometres between two points on a record
SELECT geo.distance(coords, geo.geomFromText('POINT(2.3522 48.8566)')) AS dist_km
FROM Location

-- Distance in metres
SELECT geo.distance(geo.point(0, 0), geo.point(1, 1), 'm') AS dist_m
```

---

geo.area()

Returns the area of a geometry in square degrees. The area is measured on the longitude/latitude plane, as OGC WKT, PostGIS `geometry` and Shapely do, so a 10 x 10 degree square is 100 wherever it lies: it is not the area on the surface of the Earth. Holes are left out, the area of a multi-part geometry is the sum of its parts, a rectangle that crosses the antimeridian is measured the short way round, and points and lines have an area of 0.

Syntax: `geo.area(<geometry>)`

Before 26.12.1 the result was an estimate that was neither square degrees nor the area on the sphere: the spherical area of the shape's bounding box scaled by the fraction of the box the shape fills.

```
SELECT geo.area(geo.geomFromText('POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))')) AS area
-- Returns: 100.0
```

geo.asText()

Converts a geometry object to its WKT string representation.

Syntax: `geo.asText(<geometry>)`

```
SELECT geo.asText(geo.point(2.2945, 48.8584)) AS wkt
-- Returns: 'POINT (2.2945 48.8584)'
```

---

geo.asGeoJson()

Converts a geometry object to a GeoJSON string. An empty geometry has no position: `geo.asGeoJson('POINT EMPTY')` returns `{"type":"Point","coordinates":[]}` and `geo.asGeoJson('POLYGON EMPTY')` returns `{"type":"Polygon","coordinates":[]}`. *(Since v26.12.1; before, an empty point failed with an error)*

Syntax: `geo.asGeoJson(<geometry>)`

```
SELECT geo.asGeoJson(geo.point(2.2945, 48.8584)) AS geojson
-- Returns: '{"type":"Point","coordinates":[2.2945,48.8584]}'
```

---

geo.x()

Extracts the X coordinate (longitude) from a point geometry. An empty point (`POINT EMPTY`) returns `NULL`. *(Since v26.12.1; before, it returned NaN)*

Syntax: `geo.x(<point>)`

```
SELECT geo.x(geo.geomFromText('POINT(2.2945 48.8584)')) AS longitude
-- Returns: 2.2945
```

---

geo.y()

Extracts the Y coordinate (latitude) from a point geometry. An empty point (`POINT EMPTY`) returns `NULL`. *(Since v26.12.1; before, it returned NaN)*

Syntax: `geo.y(<point>)`

```
SELECT geo.y(geo.geomFromText('POINT(2.2945 48.8584)')) AS latitude
-- Returns: 48.8584
```

Spatial Predicate Functions

All predicate functions accept a field reference or WKT string as the first argument and a geometry (from any `geo.*` constructor) as the second. When a `GEOSPATIAL` index exists on the referenced field, the query planner uses it automatically.

Function	Semantics	Index used?
`geo.within(g, shape)`	`g` is fully contained within `shape`	Yes
`geo.intersects(g, shape)`	`g` and `shape` share at least one point	Yes
`geo.contains(g, shape)`	`g` fully contains `shape`	Yes
`geo.equals(g, shape)`	`g` and `shape` are geometrically identical	Yes
`geo.crosses(g, shape)`	`g` crosses the boundary of `shape`	Yes
`geo.overlaps(g, shape)`	`g` and `shape` overlap (share interior but neither contains the other)	Yes
`geo.touches(g, shape)`	`g` touches the boundary of `shape` but interiors do not intersect	Yes
`geo.disjoint(g, shape)`	`g` and `shape` share no points	No — full scan
`geo.dWithin(g, shape, dist)`	`g` is within `dist` degrees of `shape`	No — full scan

`geo.disjoint` cannot use the index because the index stores records that *intersect* indexed cells; records absent from the index are exactly those that are disjoint. `geo.dWithin` falls back to a full scan because correct proximity indexing requires expanding the query shape into a bounding circle first — planned as a future enhancement.

geo.within()

Returns `true` if the first geometry is fully within the second. `geo.within(a, b)` always answers the same as `geo.contains(b, a)`, and a geometry is within itself (or within the same area written with different vertices), as in OGC `ST_Within`. Before 26.12.1 a geometry was not within itself, so an indexed `geo.within(coords, <shape>)` left out the record whose `coords` is that shape.

```
SELECT name FROM Location
WHERE geo.within(coords, geo.geomFromText(
  'POLYGON((2.28 48.84, 2.42 48.84, 2.42 48.90, 2.28 48.90, 2.28 48.84))'
)) = true
```

geo.intersects()

Returns `true` if the geometries share at least one point.

```
SELECT name FROM Location
WHERE geo.intersects(coords, geo.geomFromText('LINESTRING(0 0, 10 10)')) = true
```

geo.contains()

Returns `true` if the first geometry fully contains the second. A geometry contains itself, two equal points included, as in OGC `ST_Contains`. Before 26.12.1 two equal points did not contain each other, and the same area written in two ways (for example, a box with an extra vertex) could answer `false` depending on the argument order.

```
SELECT name FROM Zone
WHERE geo.contains(boundary, geo.geomFromText('POINT(2.2945 48.8584)')) = true
```

geo.dWithin()

Returns `true` if the first geometry is within the given distance (in degrees of great-circle arc) of the second. The distance is measured between the two shapes, as `geo.distance()` measures it: 0 when they intersect, otherwise between their closest points. Before 26.11.1 it was measured between the centres of the two shapes, so a point inside a large polygon could be found farther from it than the given distance.

```
-- All locations within 0.01 degrees (~1.1 km) of the Eiffel Tower
SELECT name FROM Location
WHERE geo.dWithin(coords, geo.geomFromText('POINT(2.2945 48.8584)'), 0.01) = true
```

geo.disjoint()

Returns `true` if the geometries share no points.

```
SELECT name FROM Location
WHERE geo.disjoint(coords, geo.geomFromText(
  'POLYGON((0 0, 5 0, 5 5, 0 5, 0 0))'
)) = true
```

geo.equals()

Returns `true` if the geometries are geometrically identical.

```
SELECT * FROM Location
WHERE geo.equals(coords, geo.geomFromText('POINT(2.2945 48.8584)')) = true
```

geo.crosses()

Returns `true` if the geometries cross (share some but not all interior points, and the intersection is of lower dimension than the geometries).

```
SELECT name FROM Road
WHERE geo.crosses(path, geo.geomFromText('LINESTRING(0 5, 10 5)')) = true
```

geo.overlaps()

Returns `true` if the geometries overlap (share interior but neither contains the other).

```
SELECT name FROM Zone
WHERE geo.overlaps(boundary, geo.geomFromText(
  'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))'
)) = true
```

geo.touches()

Returns `true` if the geometries share boundary points but their interiors do not intersect.

```
SELECT name FROM Parcel
WHERE geo.touches(boundary, geo.geomFromText(
  'LINESTRING(0 0, 10 0)'
)) = true
```

6.5.6. Practical Examples

Points of Interest Near a Landmark

```
-- All POIs within a 500 m bounding box of the Eiffel Tower
SELECT name, coords
FROM PointOfInterest
WHERE geo.within(coords, geo.buffer(geo.point(2.2945, 48.8584), 0.0045)) = true
ORDER BY geo.distance(coords, geo.geomFromText('POINT(2.2945 48.8584)'))
LIMIT 20
```

Delivery Zones Containing an Address

```
SELECT zone_name, carrier
FROM DeliveryZone
WHERE geo.contains(boundary, geo.geomFromText('POINT(2.3522 48.8566)')) = true
```

Fleet Vehicles Inside a Geofence

```
SELECT vehicle_id, last_seen
FROM Vehicle
WHERE geo.within(last_position,
  geo.geomFromText('POLYGON((2.28 48.85, 2.42 48.85, 2.42 48.90, 2.28 48.90, 2.28
48.85))')
) = true
  AND last_seen > date('2026-01-01', 'yyyy-MM-dd')
```

Building Envelopes Intersecting a Road Segment

```
SELECT building_id, address
FROM Building
WHERE geo.intersects(footprint, geo.geomFromText(
  'LINESTRING(2.2900 48.8600, 2.3100 48.8700)'
)) = true
```

Combining Geospatial and Graph Queries

Graph traversal and spatial predicates compose naturally in the same query:

```
SELECT expand(outV())
FROM Has
WHERE geo.within(outV().location, geo.geomFromText(
  'POLYGON((2.28 48.84, 2.42 48.84, 2.42 48.90, 2.28 48.90, 2.28 48.84))'
)) = true
```

6.5.7. Notes and Limitations

- Geometries must be stored as WKT strings in `STRING` properties. No dedicated geometry column type is introduced; WKT is the only supported format at storage level.
- GeoJSON is not a storage format, but `geo.asGeoJson()` converts any geometry to GeoJSON for output.
- 3D geometry and raster data are not supported.
- Coordinates are longitude/latitude, and a shape means the area its coordinates enclose on the longitude/latitude plane, as OGC WKT, PostGIS `geometry`, Shapely and GeoJSON (RFC 7946) read it: a polygon wider than 180 degrees of longitude, or a polar cap such as `POLYGON -180 80, 180`

`80, 180 90, -180 90, -180 80`, is the area written.

- A shape that crosses the antimeridian is written either as a `MULTIPOLYGON` split at 180, or with longitudes running past it, e.g. `POLYGON 170 0, 190 0, 190 10, 170 10, 170 0` for the 20 degrees around the antimeridian: such a shape is split there automatically. A polygon written `170 .. -170` is the 340-degree band between those meridians.
- A coordinate out of range (a latitude past +/-90, a longitude past one turn of the globe) is not a geometry: the `geo.*` functions refuse it, and a write into a property with a `GEOSPATIAL` index fails rather than store a value the index would skip. Building or rebuilding the index over records written before still skips such a value, with a warning naming the record.

Before 26.11.1 a polygon wider than 180 degrees was read as crossing the antimeridian, so it became its complement and a polar cap enclosed nothing, and an invalid coordinate was stored, skipped by the index, and made every later scan of the type fail. A `GEOSPATIAL` index holding a polygon written across the antimeridian as `170 .. -170` has to be rewritten in one of the two forms above and rebuilt.

A `GEOSPATIAL` index created before 26.11.1 is not used by queries until it is rebuilt: it may hold the complement of a polygon wider than 180 degrees, which would leave a matching record out of the answer. The `geo.*` predicates scan the type instead, which is always correct, and the index reports the advice (in the log when the database opens, and in `schema:indexes`). Run `REBUILD INDEX` on it to go back to indexed queries. * Changing the precision level of an existing geospatial index requires rebuilding the index:

```
REBUILD INDEX `Location[coords]`
```

- A `GEOSPATIAL` index cannot be marked `UNIQUE`.

For related commands, see:

- `CREATE INDEX`
- `DROP INDEX`
- `REBUILD INDEX`
- Indexes

6.6. Full-Text Index and BM25 Ranking

ArcadeDB's full-text index is built on Apache Lucene and supports configurable analyzers, advanced Lucene query syntax, multi-property indexing, native **BM25 relevance scoring** (with tunable parameters and per-field boosts), and "More Like This" similarity search.

BM25 at a glance

- **BM25 is the default** ranking for every newly created `FULL_TEXT` index. No configuration is needed: `ORDER BY $score DESC` returns results ranked by Okapi BM25.

- **Tunable**: `bm25_k1` (term-frequency saturation, default `1.2`) and `bm25_b` (document-length normalization, default `0.75`), see Similarity Models: BM25 and CLASSIC.
- **Per-field boosts**: weight a `title` over a `body` with `title_boost`, or boost terms at query time with the caret syntax, see Per-Field Boosts.
- **Explainable**: `EXPLAIN` and `PROFILE` show the `k1`/`b` parameters and the `df`/`idf` of every query term, see Inspecting Scores with EXPLAIN / PROFILE.
- **Hybrid retrieval**: fuse BM25 keyword relevance with dense vectors and learned-sparse vectors in a single query, and expand the graph from the fused results, see Hybrid Retrieval: Fusing BM25 with Vector Search.
- **Existing indexes** created before BM25 keep the legacy `CLASSIC` ranking until rebuilt, see Getting BM25 on Existing Indexes.

BM25 keyword ranking is a property of the `FULL_TEXT` index. It is not the same thing as the `LSM_SPARSE_VECTOR` index, which stores learned-sparse embeddings (SPLADE, BGE-M3) produced outside the database, see concepts/vector-search.pdf.

6.6.1. Creating a Full-Text Index

```
CREATE INDEX ON Article (content) FULL_TEXT
```

Multiple properties can be indexed together:

```
CREATE INDEX ON Article (title, body) FULL_TEXT
```

6.6.2. Configuring the Analyzer

Pass a `METADATA` block to choose a Lucene analyzer:

```
CREATE INDEX ON Article (content) FULL_TEXT
  METADATA {
    "analyzer": "org.apache.lucene.analysis.en.EnglishAnalyzer",
    "allowLeadingWildcard": false,
    "defaultOperator": "OR"
  }
```

Metadata Options

Option	Type	Default	Description
`analyzer`	string	`org.apache.lucene.analysis.standard.StandardAnalyzer`	Lucene analyzer class used for both indexing and querying

Option	Type	Default	Description
`index_analyzer`	string	—	Override analyzer used only at index time
`query_analyzer`	string	—	Override analyzer used only at query time
`allowLeadingWildcard`	boolean	`false`	Allow `*term` wildcard queries
`defaultOperator`	`"OR"` | `"AND"`	`"OR"`	Default operator between terms when none is specified
`<field>_analyzer`	string	—	Per-field analyzer override, e.g. `"title_analyzer"`
`similarity`	`"BM25"` | `"CLASSIC"`	`"BM25"`	Relevance model (see Relevance Scoring). `BM25` is the default for newly created indexes; `CLASSIC` is the legacy term-coordination (match-count) model
`bm25_k1`	float	`1.2`	BM25 term-frequency saturation (must be >= 0). Higher values let repeated terms keep increasing the score
`bm25_b`	float	`0.75`	BM25 document-length normalization, in `[0, 1]` (`0` disables length normalization)
`<field>_boost`	float	`1.0`	Per-field BM25 boost multiplier applied to field-qualified matches on that field (must be >= 0), e.g. `"title_boost"`

Common Analyzers

Analyzer Class	Description
`org.apache.lucene.analysis.standard.StandardAnalyzer`	General-purpose tokenizer (default)
`org.apache.lucene.analysis.en.EnglishAnalyzer`	English stemming and stop words
`org.apache.lucene.analysis.core.SimpleAnalyzer`	Lowercase only, no stop words
`org.apache.lucene.analysis.core.WhitespaceAnalyzer`	Split on whitespace only

Per-Field Analyzers

For multi-property indexes, each field can use a different analyzer:

```
CREATE INDEX ON Article (title, body) FULL_TEXT
  METADATA {
    "analyzer": "org.apache.lucene.analysis.standard.StandardAnalyzer",
    "title_analyzer": "org.apache.lucene.analysis.en.EnglishAnalyzer"
  }
```

Here `title` uses the English analyzer while `body` falls back to the standard analyzer.

A per-field analyzer applies to the words stored for that field and to every search qualified to it: `title:run` in `SEARCH_INDEX()`, `SEARCH_FIELDS(['title'], …)` and `title CONTAINSTEXT …`. With the index above, `title:run` finds a title containing "Running" and `title:running` does not. A search that names no field reads every property and is analyzed with the analyzer of the index, as is the document length used for BM25 scoring. On a one-property index, the `<field>_analyzer` of its property replaces the analyzer of the index.

Before 26.12.1 the `<field>_analyzer` options were stored but ignored. An index created by an older version with one of these options keeps analyzing every property with the analyzer of the index, so it answers exactly as before, and the server logs a warning when the database opens. Run `REBUILD INDEX` on it to apply its per-field analyzers.

6.6.3. Searching

A full-text index is used by `CONTAINSTEXT` and by the search functions below. Exact comparisons (`=`, `IN`, `CONTAINS`, `CONTAINSANY`, `CONTAINSALL` in SQL, property equality in Cypher and Gremlin, `lookupByKey()` in Java) never use it, because it matches words rather than whole values: they always return exact matches. Add a regular index on the property if you also need fast exact lookups.

SEARCH_INDEX()

Searches a named full-text index using Lucene query syntax.

```
SELECT * FROM Article
WHERE SEARCH_INDEX('Article[content]', 'java programming')
```

Signature: `SEARCH_INDEX(indexName, query)`

Parameter	Type	Description
`indexName`	string	The full index name as shown in the schema, e.g. `Article[content]`
`query`	string	A Lucene query string

(since v26.11.1): `SEARCH_INDEX` and `SEARCH_FIELDS` refuse a third argument with an error; before, it was ignored.

SEARCH_FIELDS()

Finds the full-text index automatically from field names, without needing to know the index name.

```
SELECT * FROM Article
WHERE SEARCH_FIELDS(['title', 'body'], 'database tutorial')
```

Signature: `SEARCH_FIELDS(fieldNames, query)`

Parameter	Type	Description
`fieldNames`	array of strings	Fields to search; a full-text index covering these fields must exist
`query`	string	A Lucene query string

(since v26.10.2): the search is restricted to the given fields. When the index chosen covers more properties than the ones listed, every unqualified clause of the query (a term, a phrase, a prefix, a wildcard, a fuzzy term) is searched in the listed fields only, as if it were written `field:clause` for each of them and ORed; a clause the query qualifies itself keeps its own field. `SEARCH_FIELDS(['title'], 'graph')` on an index over `title, body` is therefore the same search, with the same `$score`, as `SEARCH_INDEX('Article[title,body]', 'title:graph')`, and listing every property of the index is the same as `SEARCH_INDEX` on it. Before v26.10.2 a term found only in an unlisted field matched too.

When several full-text indexes cover the fields, the one over the fewest properties is used. A `SEARCH_FIELDS` condition in `WHERE` is answered by the index, like `SEARCH_INDEX`, with the same lazy loading and pushdowns (see Result Order, LIMIT and Cancellation), rather than by a scan of the type.

Cypher: db.index.fulltext.queryNodes()

Full-text search is also available from Cypher, so it can be combined with graph pattern matching in the same query. This is the Neo4j-compatible equivalent of `SEARCH_INDEX()`.

```
CALL db.index.fulltext.queryNodes('Article[content]', 'java programming')
YIELD node, score
RETURN node.title AS title, score
ORDER BY score DESC
```

Signature: `db.index.fulltext.queryNodes(indexName, query [, options])`, returning one row per match with `node` and `score` (BM25 relevance, higher is better).

For a full-text index declared on an edge type, use `db.index.fulltext.queryRelationships(indexName, query [, options])`, which yields `relationship` and `score` instead.

Results are ordered by descending score, ties broken by record id, so the optional `options` map can page through them: `skip` drops that many top rows (default `0`) and `limit` caps the rows returned (default unlimited). Both must be non-negative integers.

```
CALL db.index.fulltext.queryNodes('Article[content]', 'java programming', {skip: 20,
limit: 10})
YIELD node, score
RETURN node.title AS title, score
```

Coming from Neo4j, you may reach for the `analyzer` option. ArcadeDB rejects it with an error rather than ignoring it: the analyzer is the one stored in the index

> metadata when the index was created, and there is no per-query override.

(since v26.11.1): the `where` and `filter` options restrict the matches before `skip` and `limit` apply, as the same options of `db.index.vector.queryNodes` do. A `WHERE` after the `CALL` only sees the rows the procedure returned, so with a `limit` and a filter that rejects the best matches it answers too few rows, or none (Neo4j behaves the same way); the `where` option answers `limit` rows of the filtered matches instead. `where` is a map of property to value (`{tenant: $t}`) or a SQL condition on the record (`'year >= :minYear'`, named parameters only); `filter` is a list of the record ids, or records, the search may answer. A `where` map that a regular index can answer is decided on the matches before any record is loaded; otherwise each match is tested on its record. The scores are those of the unfiltered search.

```
CALL db.index.fulltext.queryNodes('Doc[content]', $q, {limit: 10, where: {tenant:
$t}})
YIELD node, score
RETURN node.title AS title, score
```

```
CALL db.index.fulltext.queryRelationships('Cites[note]', 'java')
YIELD relationship, score
RETURN relationship.note AS note, score
ORDER BY score DESC
```

6.6.4. Lucene Query Syntax

Both `SEARCH_INDEX` and `SEARCH_FIELDS` accept standard Lucene query syntax.

Boolean Operators

Syntax	Meaning
`java programming`	Either term (OR, default)
`+java +programming`	Both terms required (AND)
`java -python`	`java` required, `python` excluded
`java AND programming`	Explicit AND
`java OR python`	Explicit OR

```
-- Requires both terms
SELECT * FROM Article WHERE SEARCH_INDEX('Article[content]', '+java +programming')

-- Excludes documents about python
SELECT * FROM Article WHERE SEARCH_INDEX('Article[content]', 'java -python')
```

An excluded term (`-python`) removes its matches from the group it is written in, as in Lucene: `(java -python) database` still returns a document holding `python database`, through `database`, and `java`

`-(database -tutorial)` keeps a document holding all three words. A query or group made only of excluded terms (`-python`) is an ArcadeDB extension: it returns every indexed document that does not match them *(since v26.12.1: before, `java -python` returned every document of a bucket holding no `java`, less the `python` ones, and an excluded term inside a group removed its matches from the whole query).*

Phrase Queries

Wrap a phrase in double quotes to require terms to appear in order:

```
SELECT * FROM Article WHERE SEARCH_INDEX('Article[content]', '"machine learning"')
```

Wildcard Queries

Syntax	Matches
`data*`	database, datastore, dataset…
`dat?base`	database, datXbase…
`*base`	Requires `allowLeadingWildcard: true`

```
SELECT * FROM Article WHERE SEARCH_INDEX('Article[content]', 'data*')
```

Wildcard, prefix, fuzzy and regexp terms are normalized by the index's analyzer, the same way the indexed tokens were: lower-cased with the default `StandardAnalyzer`, left as they are with a case-preserving analyzer such as `WhitespaceAnalyzer` (where `Fo*` matches `Foo` but not `foo`).

Fuzzy Queries

Append `~` to match terms within an edit distance:

```
SELECT * FROM Article WHERE SEARCH_INDEX('Article[content]', 'database~')
-- also matches terms within edit distance 2 (Lucene default), e.g. "databaSe",
"databasee"
```

Field-Qualified Queries (Multi-Property Indexes)

For indexes over multiple fields, restrict a term to a specific field:

```
-- Only match "database" in the title field
SELECT * FROM Article WHERE SEARCH_INDEX('Article[title,body]', 'title:database')

-- Combine field-specific and general terms
SELECT * FROM Article WHERE SEARCH_INDEX('Article[title,body]', '+title:"multi model"
-nosql')
```

A qualifier applies to every form of a term, phrases included: `title:"multi model"` matches the phrase in `title` only. A property indexed `BY ITEM` (for example `keywords BY ITEM` on a list) is qualified with its plain name, `keywords:graph`.

An unqualified clause (a term, a prefix, a wildcard, a fuzzy or a regexp term) searches the words of every field and never the field names: `ti*` matches `tiger`, not every record with a `title`. A word that itself holds a colon (the default analyzer keeps `title:java` as one word) is searched by escaping the colon, `title\:java`, while `title:java` searches `java` in `title` only. *(Since v26.12.1: before, an unqualified prefix such as `ti*` also matched every record with a non-empty field whose name starts with it, and `title:java` also matched a record holding the text `title:java` in another field. An index created before v26.12.1 keeps words of the form `<property>:⋯` in the old form until `REBUILD INDEX <name>`; the wrong prefix, wildcard, fuzzy and regexp answers are fixed with no rebuild.)*

Boosting Terms (Caret)

Append `^N` to a term, phrase, field-qualified term, or parenthesized group to weight it more (or less, with `N < 1`) when scoring. This follows the Lucene/Elasticsearch convention and applies to BM25 indexes.

Syntax	Meaning
`java^3`	Boost the term `java` 3x
`java^0.5`	De-emphasize the term `java` (any `N < 1` lowers its weight)
`"machine learning"^2`	Boost the phrase 2x
`title:java^5`	Boost a field-qualified term 5x
`(java OR python)^3`	Boost a whole parenthesized group 3x

```
-- Make "java" count three times as much as "database" in the ranking
SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[content]', 'java^3 database') ORDER BY $score DESC

-- Boost a field-qualified term, combined with a required term
SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[title,body]', '+database title:java^5') ORDER BY $score
DESC

-- Boost a whole OR group
SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[title,body]', '(title:java OR title:python)^3
body:cooking') ORDER BY $score DESC
```

The query-time caret combines multiplicatively with any configured per-field boost (see Relevance Scoring): the effective weight is `caret * field_boost`.

6.6.5. Relevance Scoring

Similarity Models: BM25 and CLASSIC

Full-text indexes rank results with one of two similarity models, selected by the `similarity` metadata option:

Model	Description
`BM25` (default)	Industry-standard Okapi BM25 ranking: combines term frequency (TF), inverse document frequency (IDF, so rare terms count more than common ones) and document-length normalization (shorter documents matching the same term rank higher). Tunable via `bm25_k1` and `bm25_b`.
`CLASSIC`	Legacy term-coordination model: the score is simply the number of distinct query terms a document matches. Kept for backward compatibility.

`BM25` is the default for newly created indexes. Configure it explicitly, optionally tuning `bm25_k1` /`bm25_b`:

```
-- BM25 with tuned parameters
CREATE INDEX ON Article (content) FULL_TEXT
  METADATA { "similarity": "BM25", "bm25_k1": 1.2, "bm25_b": 0.75 }

-- Keep the legacy match-count scoring
CREATE INDEX ON Article (content) FULL_TEXT
  METADATA { "similarity": "CLASSIC" }
```

Per-Field Boosts

On a multi-property index, give a field a default BM25 weight with `<field>_boost`, so matches in that field contribute more to the score:

```
CREATE INDEX ON Article (title, body) FULL_TEXT
  METADATA { "similarity": "BM25", "title_boost": 3.0 }
```

A field boost applies to **field-qualified** query terms on that field (e.g. `title:java`). Unqualified terms score against the field-agnostic token with boost `1.0`. Per-field boosts and query-time caret boosts compose: the effective weight is `caret * field_boost`.

The key names the property as the index declares it, so for a property indexed `BY ITEM` it is `"keywords by item_boost"`; the query still qualifies it with the plain name (`keywords:graph`).

From the Java API

The same configuration is available on the index builder:

```
database.getSchema().buildTypeIndex("Article", new String[] { "title", "body" })
    .withType(Schema.INDEX_TYPE.FULL_TEXT).withFullTextType()
    .withBM25(1.2f, 0.75f)          // similarity = BM25, k1, b
    .withFieldBoost("title", 3.0f)  // per-field boost
```

```
    .create();
```

Use `withSimilarity("CLASSIC")` to select the legacy model instead of `withBM25(⋯)`.

Relevance Score ($score)

Every match carries a floating-point relevance score, exposed as the query variable `$score`. Use it in projections or `ORDER BY` clauses:

```
SELECT title, $score
FROM Article
WHERE SEARCH_INDEX('Article[content]', 'java programming')
ORDER BY $score DESC
```

```
SELECT title, $score AS relevance
FROM Article
WHERE SEARCH_FIELDS(['content'], 'java programming')
ORDER BY relevance DESC
LIMIT 10
```

With BM25, documents matching rarer, more discriminative terms (and shorter documents matching the same term) rank higher. With CLASSIC, documents that match more distinct query terms rank higher.

`$score` is always a floating-point number, including for `CLASSIC` indexes (where it is the match count widened to a float, e.g. `3.0`). Application code that previously read `$score` as an integer must read it as a `Number`/float.

Result Order, LIMIT and Cancellation

When `SEARCH_INDEX(⋯)` drives the query, matches come back best first: by descending `$score`, and by ascending RID among equal scores, so the order is stable from one run to the next. Records are loaded only as rows are pulled, so a `LIMIT`, or a filter that stops early, also stops the record loading.

When nothing between the search and the `LIMIT` can drop, add or reorder a row (a single `SEARCH_INDEX` condition with no other filter, no `ORDER BY` or exactly `ORDER BY $score DESC` - also through an alias of the score, as in `SELECT $score AS s ⋯ ORDER BY s DESC` - and no `GROUP BY`, `DISTINCT`, aggregation, `UNWIND` or `expand()`, on the type the index belongs to or one of its supertypes), the search keeps only the best `SKIP + LIMIT` matches and the sort is skipped. `EXPLAIN` shows the rows the plan needs on the fetch step as `(best N needed)`:

```
EXPLAIN SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[content]', 'java database') = true
ORDER BY $score DESC LIMIT 20
```

(since v26.12.1): a filter ANDed with `SEARCH_INDEX` or `SEARCH_FIELDS` that no index decides (`AND`

`deleted_at IS NULL`, `AND n > 5`) no longer disables the `LIMIT` pushdown. The search starts from the best `SKIP + LIMIT` matches and, when the filter rejects some of them, goes on past them in the same order (descending `$score`, then RID) until the `LIMIT` is reached, so no sort runs after the filter and the query costs about the rows it returns rather than every match. When it has to go on, it reuses the postings it already read rather than reading the index again. The rows, their order and their scores are those of the complete search. `EXPLAIN` shows `(best N needed)` on the fetch step, followed by the filter. A filter that itself runs a full-text search (a second `SEARCH_INDEX` or `SEARCH_FIELDS`, also nested as in `ifnull(SEARCH_INDEX(…), false) = true`), calls a function the engine does not ship, or holds a subquery, keeps the previous plan: no pushdown, and an `ORDER BY $score DESC` sorts every row.

```
SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[content]', 'java database') = true AND deleted_at IS NULL
ORDER BY $score DESC LIMIT 20
```

(since v26.10.2): the `@rid` conditions ANDed with the search (`=`, `<>`, `<`, `<=`, `>`, `>=`, `IN`, `NOT IN` against a literal, a parameter or a variable computed by a global `LET`) are decided on the matches' RIDs before any record is loaded, and no longer count as a filter that disables the `LIMIT` pushdown. With `ORDER BY @rid` (ascending or descending) the matches come back in RID order without a sort, so a `LIMIT` loads only the rows of the page. Together they make keyset pagination over a full-text result cost one page per page, rather than every match past the cursor. `EXPLAIN` shows the conditions as `RID FILTER` and the order as `ORDER BY @rid ASC (first N needed)`:

```
SELECT @rid, title, $score FROM Article
WHERE @rid > :cursor AND SEARCH_INDEX('Article[content]', 'java database') = true
ORDER BY @rid LIMIT 500
```

The same `@rid` conditions are decided before loading for the `geo.*` predicates driven by a geospatial index, and for the root node of a `MATCH` (`{type: Article, where: (@rid > :cursor AND SEARCH_INDEX(…) = true)}`).

(since v26.11.1): a property equality ANDed with the search (`property = value`, against a literal, a parameter or a variable computed by a global `LET`) is decided on the matches before any record is loaded when a regular index lists the records it accepts: the index on `tenant` below lists the records of tenant `:t`, and the search drops every other match before it scores and loads them. This is what makes a tenant or access filter on a large shared corpus cheap when the filter rejects the best matches: the query loads the rows it returns, rather than every match until enough of them passed the filter. The listing is read only when it is no longer than the matches of the bucket, and is exact only for an index over exactly the equalities' properties that compares keys as `=` does (not a case-insensitive one); then the condition no longer counts as a filter that disables the `LIMIT` pushdown. Otherwise the listing still drops the matches it does not hold, and the rest are tested on their records once loaded. The BM25 scores are those of the unfiltered search. `EXPLAIN` shows the conditions handed to the search as `INDEX PRE-FILTER`:

```
CREATE INDEX ON Doc (tenant) NOTUNIQUE;
```

```
SELECT @rid, $score FROM Doc
WHERE SEARCH_INDEX('Doc[content]', :q) = true AND tenant = :t
ORDER BY $score DESC LIMIT 10
```

(since v26.10.2): an `OR` of searches (`SEARCH_INDEX(…) = true OR SEARCH_INDEX(…) = true`, `SEARCH_FIELDS` included) with no `ORDER BY` or exactly `ORDER BY $score DESC` is answered best first across all of them: the searches are merged by `$score` as their rows are read, instead of being run one after the other, deduplicated and sorted. A `LIMIT` therefore loads about the rows it returns rather than every match of every index, and without an `ORDER BY` the second index contributes to the first rows too, instead of only once the first one has none left. A record matched by several searches comes back once, with the best of its scores (the scores of different indexes are compared as they are, as `ORDER BY $score DESC` always compared them). A filter ANDed with the `OR` (`(… OR …) AND lang = 'en'`) keeps the merge; before v26.12.1 the searches were then not cut to the best `SKIP + LIMIT` matches, since then each one starts from them and goes on past them in order when the filter rejects rows. `EXPLAIN` shows `MERGE BY $score DESC` with one branch per search, each with `(best N needed)` when the `LIMIT` is pushed down:

```
SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[title]', 'java') = true OR SEARCH_INDEX(
'Article[content]', 'java') = true
ORDER BY $score DESC LIMIT 10
```

The merge does not apply when a branch of the `OR` is not a search, holds a second search or a call to a non-deterministic function, or reads a per-record `LET`: those branches still run one after the other, and an `ORDER BY $score DESC` sorts every row.

A cancelled full-text query, or one past `arcadedb.command.timeout` or its SQL `TIMEOUT`, stops inside the search: during the posting walk, the scoring and the record loading, rather than after the whole search is over.

Inspecting Scores with EXPLAIN / PROFILE

`EXPLAIN` (or `PROFILE`) of a query whose `WHERE` uses `SEARCH_INDEX(…)` annotates the full-text fetch step with a `SCORING` line: the similarity model, the `k1`/`b` parameters, the corpus statistics, and, per query term, its document frequency (`df`), `idf` and applied `boost`.

```
EXPLAIN SELECT title, $score FROM Article
WHERE SEARCH_INDEX('Article[content]', 'java database') = true
```

6.6.6. More Like This

"More Like This" finds documents similar to one or more source documents. It extracts representative terms from the sources, then searches for other documents sharing those terms.

The four full-text search functions (`searchIndex`, `searchFields`, `searchIndexMore`, `searchFieldsMore`) are registered under the `fulltext.*` namespace. The legacy

> snake_case names (`search_index`, `search_fields`, `search_index_more`, `search_fields_more`) remain available as aliases for backward compatibility. Both forms dispatch to the same implementation.

fulltext.searchIndexMore()

```
SELECT title, $score, $similarity
FROM Article
WHERE `fulltext.searchIndexMore`('Article[content]', [#10:3])
ORDER BY $similarity DESC
```

Signature: `fulltext.searchIndexMore(indexName, sourceRIDs [, options])` (alias: `search_index_more`)

Parameter	Type	Description
`indexName`	string	Full-text index name
`sourceRIDs`	array of RIDs	One or more source documents
`options`	map	Optional MLT tuning options (see More Like This Configuration). Unknown keys are rejected.

fulltext.searchFieldsMore()

Same as `fulltext.searchIndexMore` but resolves the full-text index automatically from field names:

```
SELECT title, $similarity
FROM Article
WHERE `fulltext.searchFieldsMore`(['content'], [#10:3])
ORDER BY $similarity DESC
```

Signature: `fulltext.searchFieldsMore(fieldNames, sourceRIDs [, options])` (alias: `search_fields_more`)

Parameter	Type	Description
`fieldNames`	array of strings	Fields to search; a full-text index covering these fields must exist
`sourceRIDs`	array of RIDs	One or more source documents
`options`	map	Optional MLT tuning options (see More Like This Configuration). Unknown keys are rejected.

Multiple Source Documents

Provide multiple RIDs to find documents similar to a combination of sources:

```
SELECT title, $similarity
FROM Article
```

```
WHERE SEARCH_INDEX_MORE('Article[content]', [#10:3, #10:4])
ORDER BY $similarity DESC
```

Similarity Score ($similarity)

`$similarity` is a normalized score from `0.0` to `1.0`. The most similar document in the result set always receives `1.0`.

```
SELECT title, $score, $similarity
FROM Article
WHERE SEARCH_INDEX_MORE('Article[content]', [#10:3])
ORDER BY $similarity DESC
LIMIT 5
```

More Like This Configuration

Pass an optional map of tuning options. Unknown keys are rejected with a descriptive error to catch typos; keys are case sensitive.

```
SELECT title, $similarity
FROM Article
WHERE `fulltext.searchFieldsMore`(['title', 'content'], [#10:3], {
  minTermFreq: 1,
  minDocFreq: 3,
  maxQueryTerms: 50,
  excludeSource: false
})
```

Option	Type	Default	Description
`minTermFreq`	int	`2`	Minimum times a term must appear in the source document(s) to be considered
`minDocFreq`	int	`5`	Minimum number of index documents that must contain a term for it to be used
`maxDocFreqPercent`	float	`null`	Exclude terms appearing in more than this fraction of all documents (e.g. `0.5` = 50%)
`maxQueryTerms`	int	`25`	Maximum number of terms to use in the similarity query
`minWordLen`	int	`0`	Ignore terms shorter than this length (0 = no minimum)
`maxWordLen`	int	`0`	Ignore terms longer than this length (0 = no maximum)
`boostByScore`	boolean	`true`	Weight terms by TF-IDF score rather than raw frequency

Option	Type	Default	Description
`excludeSource`	boolean	`true`	Exclude source documents from results
`maxSourceDocs`	int	`25`	Maximum number of source RIDs allowed

6.6.7. Practical Examples

Blog Search with Ranking

```
SELECT title, author, $score AS relevance
FROM BlogPost
WHERE SEARCH_INDEX('BlogPost[title,body]', '+java +spring -legacy')
ORDER BY relevance DESC
LIMIT 20
```

Autocomplete with Prefix Wildcard

```
SELECT title
FROM Product
WHERE SEARCH_INDEX('Product[name]', 'micro*')
```

Stemming with English Analyzer

With `EnglishAnalyzer`, searching for `"run"` also matches `"running"`, `"runs"`, and `"ran"`:

```
CREATE INDEX ON Article (content) FULL_TEXT
  METADATA { "analyzer": "org.apache.lucene.analysis.en.EnglishAnalyzer" }

SELECT * FROM Article WHERE SEARCH_INDEX('Article[content]', 'running')
-- also returns documents containing "run", "runs", "ran"
```

Related Articles Recommendation

```
SELECT title, $similarity AS score
FROM Article
WHERE SEARCH_INDEX_MORE('Article[title,body]', [#10:42], {
  "minTermFreq": 1,
  "minDocFreq": 2,
  "maxQueryTerms": 30
})
ORDER BY score DESC
LIMIT 5
```

Readers Who Liked This Also Liked

```
SELECT title, $similarity
FROM Article
WHERE SEARCH_INDEX_MORE('Article[content]', [#10:1, #10:7, #10:12])
ORDER BY $similarity DESC
LIMIT 10
```

Hybrid Retrieval: Fusing BM25 with Vector Search

With native BM25, the full-text index becomes a first-class leg of **hybrid retrieval**: exact keyword relevance (BM25), dense semantic similarity, and learned-sparse recall can be merged into a single ranked top-K, server-side, with `vector.fuse` (see concepts/vector-search.pdf).

This is the accuracy-first pattern behind enterprise document search - for example legal/government case files, where matching an exact statute or case number (keyword) matters as much as semantic similarity. Here a `Case` type carries the report `content` (full-text, BM25), a `dense` embedding, and learned-sparse `tokens`/`weights`. `RRF` fusion tolerates the very different score scales (BM25 in `[0, +inf)`, cosine in `[-1, 1]`):

```
SELECT title, caseNumber, score
FROM (
  SELECT expand(`vector.fuse`(
      `vector.neighbors`('Case[dense]', :denseVec, 100),
      `vector.sparseNeighbors`('Case[tokens,weights]', :qIdx, :qVal, 100),
      (SELECT @rid, $score FROM Case WHERE SEARCH_INDEX('Case[content]', :keywords) =
true LIMIT 100),
      { fusion: 'RRF', k: 60 }
  ))
)
ORDER BY score DESC
LIMIT 10
```

Because the fused candidates are records, the graph can be expanded in the same statement - rank the most relevant cases, then traverse to their related cases, circulars, and evidence:

```
SELECT caseNumber, title
FROM (
  SELECT expand(out('RELATED_TO'))
  FROM (
    SELECT expand(`vector.fuse`(
        `vector.neighbors`('Case[dense]', :denseVec, 100),
        (SELECT @rid, $score FROM Case WHERE SEARCH_INDEX('Case[content]', :keywords)
= true LIMIT 100),
        { fusion: 'RRF' }
    )) LIMIT 10
  )
```

```
)
```

6.6.8. Maintenance: BM25 Corpus Statistics

BM25 maintains per-type corpus counters (document count and total document length) that feed the average-document-length normalizer. An index on a type that has subtypes counts the documents of the whole hierarchy, as its searches do. They are updated incrementally as documents are indexed and removed, and only when the transaction commits: a rolled-back transaction, or one retried after a `ConcurrentModificationException`, never touches them. On an HA cluster the transaction carries the change it made to the counters, so every node moves them together and the same query scores the same on every replica. This holds once every node of the cluster advertises the capability to read that change; while one does not (a node on an older version, or one that has not answered the capability probe yet, as on a freshly started leader), the leader logs a warning and the other nodes do not count those transactions. The first search of a node repairs a difference in the number of documents; a rewrite that keeps the number of documents changes only the total length of the documents, which that check cannot see, so run `REBUILD INDEX ⋯ WITH statsOnly = true` on the leader once every node is upgraded. The first BM25 query of a session checks them with a cheap live count and, if they disagree (after a crash, for example), recomputes them with a scan of the type. The result of that repair is kept: it is written with the schema, at the next schema save or when the database closes, so it is paid once and not again after every clean restart. After a crash, or a process that is killed, the pages are restored from the log but the counters can lag them, because the counters are written at a clean close and at schema saves, not by the log. The recovery therefore marks every BM25 index for a full recompute, which its first search runs, so both the document count and the total length are repaired *(since v26.11.1: before, only a difference in the document count was noticed)*. The total length cannot be compared with the live data without analyzing the whole type, so the counters are saved with what that total depends on: the index analyzer that measured the lengths, and the rule that counts the tokens. When the analyzer the index uses now is not the one that measured them (the definition in `schema.json` was edited, say), the first search recomputes them even when the number of documents still agrees, and logs a warning that names both analyzers *(since v26.11.1: before, such a corpus kept the old lengths, and so the old average document length, for good)*. Counters saved by a version that did not record the analyzer are trusted as they are, and the next schema save records it. The analyzer is told by what it does, not only by its class name: the index measures a fixed set of sample texts with it, so a custom analyzer that changes behaviour under the same class name, or a Lucene upgrade that tokenizes differently, also triggers the recompute *(since v26.11.1: before, only a new class name was noticed)*. A total length or document count that a removal would drive below zero is a sign the counters were already wrong: it is counted, logged, and repaired by a recompute at the next search, rather than silently kept at zero. That pending recompute is saved with the counters, so a restart before the next search does not cancel it, and it is cleared only by a recompute that could be proven exact *(since v26.11.1: before, a restart in between kept the wrong total length for good)*. On an HA cluster that write is local to the node: the files of a follower and of the leader can hold different counters until the next `statsOnly` rebuild on the leader, which every node adopts. The warning that announces it names the logical index and the exact statement to run, for example `REBUILD INDEX `Article[content]` WITH statsOnly = true`.

To look at the counters without triggering the repair, read the index from the schema catalog:

```
SELECT bm25 FROM schema:index:`Article[content]`
```

`bm25` holds `totalDocs`, `sumDocLength`, `avgDocLength`, `countersValid`, `staleCheckDone` (whether the session's drift check has run), `scans` (full-type scans published into the counters), `epoch` (the number of explicit recomputes), `measuredWith` (the analyzer the total length was measured with, with a fingerprint of how it counts tokens), `measuredWithCurrentAnalyzer` (`false` when the index analyzes with another one, or the same one now counts differently, so the next search recomputes the counters) `clamps` (how many times a counter would have gone below zero since the database opened; anything above `0` means the counters were repaired) and `rescanRequired` (`true` when the counters are known to be wrong and the next search recomputes them; unlike `clamps` it survives a restart).

For an exact repair without a full reindex, run:

```
-- Re-derive the corpus statistics for one index...
REBUILD INDEX `Article[content]` WITH statsOnly = true

-- ...or for every full-text index
REBUILD INDEX * WITH statsOnly = true
```

This is also a good way to "pre-warm" the counters of a BM25 index migrated from an older version before serving traffic (otherwise the first query does a one-time full scan to build them).

On an HA cluster, run it on the leader: the recomputed counters are saved with the schema and shipped to the followers, which adopt them even though they keep their own copy of the index.

6.6.9. Behavior Changes When Upgrading

When BM25 support was introduced, existing full-text indexes keep their `CLASSIC` ranking (BM25 is opt-in for existing data, see Getting BM25 on Existing Indexes), but the following user-visible behaviors changed for **all** full-text indexes, including CLASSIC ones. Review them before upgrading.

Change	Impact	Migration
`$score` is now a **float**	It was an `Integer` match count for CLASSIC indexes and is now the same value widened to a `Float` (e.g. `3` → `3.0`). Code reading `$score` as a `java.lang.Integer` will fail (e.g. `ClassCastException`).	Read `$score` as a `Number`/`Float`, e.g. `Number) result.getProperty("$score".floatValue()`. SQL that only sorts/filters on `$score` is unaffected.

Change	Impact	Migration
Cursor order is now **most-relevant-first**	The CLASSIC full-text cursor previously returned matches in **ascending** score order (least relevant first) and ignored the result `limit`. It now returns descending score (with a stable tiebreaker) and honors `limit` - consistent with BM25. Applications that consumed the raw `Index.get(…)` order without an explicit `ORDER BY` will see it reversed, and a `limit` now truncates to the top-N.	Add an explicit `ORDER BY $score ASC` if you relied on the old order; SQL queries already using `ORDER BY $score DESC` are unaffected.
`IndexCursorEntry` identity excludes the score (extension API)	`equals()`/`hashCode()` now use only `(record, keys)`. Code that put `IndexCursorEntry` objects into a `Set` (or used them as map keys) expecting score-differentiated entries now deduplicates on `(record, keys)` only.	Treat `(record, keys)` as the identity. Affects only code working directly with the low-level index cursor API.

6.6.10. Getting BM25 on Existing Indexes

Indexes created before BM25 support open and keep scoring with `CLASSIC` (apart from the behavior changes in Behavior Changes When Upgrading). To switch existing data to BM25, **rebuild** the index: term frequencies were never stored before, so there is no in-place migration.

```
REBUILD INDEX `Article[content]`
```

6.6.11. BM25 Indexes on Lists and Maps Built Before v26.11.1

Since v26.11.1 a full-text index on a `BY ITEM`, `BY KEY` or `BY VALUE` property indexes the whole list (or map) of a record as one document. Before, each element counted as a document of its own, and an update that removed one of two elements analyzing to the same word (`beta` and `BETA`) could drop the record from the index. A BM25 index of that kind created with an older version still finds the right records, and its corpus counters repair themselves on the first search. The records not written since the upgrade, though, are ranked with their old per-element statistics until the index is rebuilt in full with `REBUILD INDEX <name>` (the name as `SELECT name FROM schema:indexes` lists it; `WITH statsOnly = true` is not enough). `CLASSIC` indexes need nothing.

6.6.12. Notes

- Full-text indexes require all indexed properties to be of type `STRING`.
- Full-text indexes cannot be marked `UNIQUE`.
- Without a `METADATA` block, newly created indexes use `StandardAnalyzer` with `OR` default operator,

leading wildcards disabled, and **BM25** similarity.

- `$score` and `$similarity` are always available as query variables for matching documents; non-matching documents receive `0`.
- `$score` is a floating-point value (for `CLASSIC` indexes too, where it is the match count as a float).
- BM25 is scored per bucket (like an Elasticsearch shard): on a multi-bucket type, `$score` is not globally calibrated across buckets, so prefer `ORDER BY $score` over comparing absolute scores across documents in different buckets.
- The direct `Index.get(text)` Java API does simple token matching only; use `SEARCH_INDEX(…)` for the full Lucene query syntax (boolean operators, phrases, wildcards, caret boosts).

6.7. Key/Value

The key/value model is the simplest data model: every value is reachable through a single identifier (the key). ArcadeDB supports the key/value pattern natively without sacrificing the richer features of the multi-model engine — values can be primitive types, full documents, or even graph elements.

6.7.1. How Keys Work in ArcadeDB

Every record stored in ArcadeDB is automatically addressable by its Record ID (RID), a compact identifier of the form `#bucket:offset`. The RID provides O(1) direct access to any record, which means the entire database functions as a high-performance key/value store — no separate engine, no extra indexes required.

In addition to RIDs, you can layer your own keys on top:

- **Indexed keys** — create a unique LSM-Tree index on any property to look up records by an arbitrary identifier such as `userId`, `sku`, or `email`.
- **Bucket-based keys** — group records into buckets and use the bucket as a key/value namespace where records are inserted and retrieved by RID without requiring a type schema.
- **Composite keys** — create indexes spanning multiple properties when the natural key is not a single value (for example, `(tenantId, accountId)`).

6.7.2. Key/Value with the Redis Wire Protocol

ArcadeDB speaks the Redis wire protocol, so any Redis client (here Jedis) can use it as a drop-in key/value backend. Simple string commands (`SET`, `GET`, `INCR`, …) operate on a fast in-memory namespace:

```
final Jedis jedis = new Jedis("localhost", 6379);

jedis.set("foo", "1");                 // store
String value = jedis.get("foo");       // read -> "1"
boolean present = jedis.exists("foo"); // check
jedis.incr("foo");                     // atomic increment -> 2
jedis.incrBy("foo", 3);                // -> 5
```

```
String removed = jedis.getDel("foo");  // read + delete atomically
```

To persist values as queryable documents, back the key with a unique index and use the hash commands. The `HGET`/`HEXISTS`/`HDEL` key uses the form `database.Type[indexName]`:

```
// One-time schema: a type plus a unique index acting as the key
database.command("sqlscript",
    "CREATE DOCUMENT TYPE Account;" +
    "CREATE PROPERTY Account.id LONG;" +
    "CREATE INDEX ON Account (id) UNIQUE;" +
    "CREATE PROPERTY Account.email STRING;" +
    "CREATE INDEX ON Account (email) UNIQUE;");

// Store a JSON value under the "Account" type
jedis.hset("mydb", "Account",
"{'id':1,'email':'jay@commodore.com','firstName':'Jay'}");

// Read it back by the indexed key (id == 1)
JSONObject doc = new JSONObject(jedis.hget("mydb.Account[id]", "1"));

// ...or by the email key
doc = new JSONObject(jedis.hget("mydb.Account[email]", "jay@commodore.com"));

boolean present = jedis.hexists("mydb.Account[id]", "1");
jedis.hdel("mydb.Account[id]", "1");   // delete by key
```

Records can also be addressed directly by their RID: `jedis.hget("mydb", "#13:432")`.

6.7.3. Key/Value with the Java API

Create a unique index on the property you want to use as the key, then look records up with `lookupByKey`:

```
database.command("sql",
    "CREATE DOCUMENT TYPE Account;" +
    "CREATE PROPERTY Account.id LONG;" +
    "CREATE INDEX ON Account (id) UNIQUE");

// Single-key lookup
final IndexCursor result = database.lookupByKey("Account", "id", 12345L);
if (result.hasNext())
  Document value = result.next().asDocument();

// Composite-key lookup (e.g. multi-tenant natural keys)
final IndexCursor composite = database.lookupByKey("Account",
    new String[] { "tenantId", "accountId" },
    new Object[] { "acme", 42L });
```

For ephemeral, in-memory state (session data, counters) that does not need its own type, use database global variables, which are also readable from SQL:

```
database.setGlobalVariable("myKey", "myValue");
Object value = database.getGlobalVariable("myKey");   // "myValue"
// SELECT $myKey returns the value from SQL
```

6.7.4. Key/Value over HTTP

Over the HTTP API, key/value access is expressed as SQL through the `/api/v1/command/<database>` endpoint. A lookup by indexed key is a simple parameterized query:

```
# Read the value whose key id = 1
curl -X POST "http://localhost:2480/api/v1/command/mydb" \
  -u root:password -H "Content-Type: application/json" \
  -d '{"language":"sql","command":"SELECT FROM Account WHERE id =
:id","params":{"id":1}}'

# Store / upsert a value by key
curl -X POST "http://localhost:2480/api/v1/command/mydb" \
  -u root:password -H "Content-Type: application/json" \
  -d '{"language":"sql","command":"UPDATE Account SET email = :e UPSERT WHERE id =
:id","params":{"id":1,"e":"jay@commodore.com"}}'
```

This makes ArcadeDB usable as a drop-in key/value backend for caches, session stores, configuration stores, or any pattern where direct addressing by identifier is the primary access mode.

6.7.5. Comparison with Other Models

Relational Model	Key/Value Model	ArcadeDB Key/Value Model
Table	Bucket	Bucket
Row	Key/Value pair	Document
Column	not available	Document field or Vertex/Edge property
Relationship	not available	Relationship

6.7.6. Why ArcadeDB for Key/Value

Most key/value stores treat the value as opaque — a blob of bytes the database doesn't understand. ArcadeDB treats every value as a structured record:

- **Rich values** — values can be documents, vertices, or edges with typed properties, not just byte arrays.
- **Queryable** — the same record retrievable by RID is also queryable with SQL, Cypher, Gremlin,

and other languages, without re-indexing or duplicating data.

- **Transactional** — key/value writes participate in full ACID transactions alongside graph and document operations.
- **Persistent** — LSM-Tree storage with automatic compaction ensures durability and efficient disk usage at scale.
- **Multi-protocol** — access the same data through HTTP, the Redis wire protocol, the Java API, or any other supported interface.

6.7.7. Further Reading

- Record ID (RID) — how RIDs work and how to use them as primary keys
- Buckets — physical storage units that double as key/value namespaces
- HTTP API — run SQL commands and queries (including key lookups) over REST
- Redis Wire Protocol — speak the Redis protocol against ArcadeDB

6.8. Record

A record is the smallest unit you can load from and store in the database. Records come in three types:

- Document
- Vertex
- Edge

6.8.1. Document

Documents are softly typed and are defined by schema types, but you can also use them in a schema-less mode too. Documents handle fields in a flexible manner. You can easily import and export them in JSON format. For example,

```
{
  "name":"Jay",
  "surname":"Miner",
  "job":"Developer",
  "creations":[{
    "name":"Amiga 1000",
    "company":"Commodore Inc."
  },{
    "name":"Amiga 500",
    "company":"Commodore Inc."
  }]
}
```

6.8.2. Vertex

In graph databases the vertices (also: vertices), or nodes represent the main entity that holds the information. It can be a patient, a company or a product. Vertices are themselves documents with some additional features. This means they can contain embedded records and arbitrary properties exactly like documents. Vertices are connected with other vertices through edges.

6.8.3. Edge

An edge, or arc, is the connection between two vertices. Edges can be unidirectional and bidirectional. One edge can only connect two vertices. Edges, like vertices, are also documents with additional features.

For more information on connecting vertices in general, see Relationships below.

6.8.4. Record ID

When ArcadeDB generates a record, it auto-assigns a unique identifier called a Record ID, RID for short. The syntax for the RID is the pound symbol (`#`) with the bucket identifier, colon (`:`), and the position like so:

`#<bucket-identifier>:<record-position>`.

- **bucket-identifier**: This number indicates the bucket id to which the record belongs. Positive numbers in the bucket identifier indicate persistent records. You can have up to 2,147,483,648 buckets in a database.
- **record-position**: This number defines the absolute position of the record in the bucket.

A special case is `#-1:-1` symbolizing the null RID.

The prefix character `#` is mandatory.

Each Record ID is immutable, universal, and only reused when configured to, see `bucketReuseSpaceMode`. Additionally, records can be accessed directly through their RIDs at O(1) complexity which means the query speed is constant, unaffected by database size. For this reason, you don't need to create a field to serve as the primary key as you do in relational databases.

Record Retrieval Complexity

Retrieving a record by RID is of complexity O(1). This is possible as the RID itself encodes both, the file a record is stored in, and the position inside it. In an RID, i.e. `#12:1000000`, the bucket identifier (here `#12`) specifies the record's associated file, while the record position (here `1000000`) describes the position inside the file. Bucket files are organized in pages (with default size `64KB`) with a maximum number records per page (by default `2048`). To determine the byte position of a record in a bucket file, the rounded down quotient of record position and maximum records per page yields the page (here `⌊1000000 / 2048⌋`), and the remainder gives the position on the page (here `⌊1000000 % 2048⌋`). In pseudo-code this computation is given by:

```
int pageId = floor(rid.getPosition() / maxRecordsInPage);
int positionInPage = floor(rid.getPosition() % maxRecordsInPage);
```

6.8.5. Types

The concept of type is taken from the Object Oriented Programming paradigm, sometimes known as "Class". In ArcadeDB, types define records. It is closest to the concept of a "Table" in relational databases and a "Class" in an object database.

Types can be schema-less, schema-full, or a mix. They can inherit from other types, creating a tree of types. Inheritance, in this context, means that a subtype extends a parent type, inheriting all of its properties and attributes. Practically, this is done by extending a type or setting a super-type.

Each type has its own buckets (data files). A type can support multiple buckets. When you execute a query against a type, it automatically fetches from all the buckets that are part of the type. When you create a new record, ArcadeDB selects the bucket to store it in using a configurable strategy — round-robin, per-thread, or hashed by a property.

As a default, ArcadeDB creates one bucket per type, but can be configured to for example to as many cores (processors) the host machine has, see `typeDefaultBuckets`. In this, CRUD operations can go full speed in parallel with zero contention between CPUs and/or cores. Having many buckets per type means having more files at file system level. Check if your operating system has any limitation with the number of files supported and opened at the same time (`ulimit` for Unix-like systems).

You can query the defined types by executing the following SQL query: `select from schema:types`.

6.8.6. Buckets

Where types provide you with a logical framework for organizing data, buckets provide physical or in-memory space in which ArcadeDB actually stores the data. Each bucket is one file at file system level. It is comparable to the "collection" in Document databases, the "table" in Relational databases and the "cluster" in OrientDB. You can have up to 2,147,483,648 buckets in a database.

A bucket can only be part of one type. This means two types can not share the same bucket. Also, sub-types have their separate buckets from their super-types.

When you create a new type, the `CREATE TYPE` statement automatically creates the physical buckets (files) that serve as the default location in which to store data for that type. ArcadeDB forms the bucket names by using the type name + underscore + a sequential number starting from 0. For example, the first bucket for the type `Beer` will be `Beer_0` and the correspondent file in the file system will be `Beer_0.31.65536.bucket`. By default ArcadeDB creates one bucket per type. For massive inserts, performance can be improved by creating additional buckets and hence taking advantage of parallelism, i.e. by creating one bucket for each CPU core on the server.

6.8.7. Types vs. Buckets in Queries

The combination of types and buckets is very powerful and has a number of use cases. In most case, you can work with types and you will be fine. But if you are able to split your database into multiple buckets, you could address a specific bucket based instead of the entire type. By wisely using the buckets to divide your database in a way that help you with the retrieval means zero or less use of indexes. Indexes slow down insertion and take space on disk and RAM. In most cases you need indexes to speed up your queries, but in some use cases you could totally or partially avoid using indexes and still having good performance on queries.

6.8.8. One bucket per period

Consider an example where you create a type `Invoice`, with one bucket per year. `Invoice_2015` and `Invoice_2016`. You can query all invoices using the type as a target with the `SELECT` statement.

```
SELECT FROM Invoice
```

In addition to this, you can filter the result set by the year. The type `Invoice` includes a `year` field, you can filter it through the `WHERE` clause.

```
SELECT FROM Invoice WHERE year = 2016
```

You can also query specific records from a single bucket. By splitting the type `Invoice` across multiple buckets, (that is, one per year in our example), you can optimize the query by narrowing the potential result set.

```
SELECT FROM BUCKET:Invoice_2016
```

By using the explicit bucket instead of the logical type, this query runs significantly faster, because ArcadeDB can narrow the search to the targeted bucket. No index is needed on the year, because all the invoices for year 2016 will be stored in the bucket `Invoice_2016` by the application.

6.8.9. One bucket per location

Like with the example above, we could split our records by location creating one bucket per location. Example:

```
CREATE BUCKET Customer_Europe
CREATE BUCKET Customer_Americas
CREATE BUCKET Customer_Asia
CREATE BUCKET Customer_Other

CREATE VERTEX TYPE Customer BUCKET Customer_Europe,Customer_Americas,Customer_Asia
,Customer_Other
```

Here we are using the graph model by creating a vertex type, but it's the same with documents. Use `CREATE DOCUMENT TYPE` instead.

Now in your application, store the vertices or documents in the right bucket, based on the location of such customer. You can use any API and set the bucket. If you're using SQL, this is the way you can insert a new customer into a specific bucket.

```
INSERT INTO BUCKET:Customer_Europe CONTENT { firstName: 'Enzo', lastName: 'Ferrari' }
```

Since a bucket can only be part of one type, when you use the bucket notation with SQL, the type is inferred from the bucket, "Customer" in this case.

With the Java API the record's type is the one it was created with, so the bucket named in `save(bucketName)` must belong to that type: `database.newVertex("Customer").save("Customer_Europe")` stores it, while saving it into a bucket of another type (a subtype included) is refused with an `IllegalArgumentException` since v26.11.1. Before, the record was written there and read back as the bucket's type, past the indexes and the listeners of the type it was saved with.

When you're looking for customers based in Europe, you could execute this query:

```
SELECT FROM BUCKET:Customer_Europe
```

You can go even more specific by creating a bucket per country, not just for continent, and query from that bucket. Example:

```
CREATE BUCKET Customer_Europe_Italy
CREATE BUCKET Customer_Europe_Spain
```

Now get all the customers that live in Italy:

```
SELECT FROM BUCKET:Customer_Europe_Italy
```

You can also specify a list of buckets in your query. This is the query to retrieve both Italian and Spanish customers.

```
SELECT FROM BUCKET:[Customer_Europe_Italy,Customer_Europe_Spain]
```

6.8.10. Bucket Selection Strategy

When a type is backed by more than one bucket, ArcadeDB has to decide which bucket should receive each new record. That decision is made by the type's **bucket selection strategy**. The strategy is set on the type (and inherited from the database default), so it is transparent to the application code that issues the `INSERT` or `CREATE`.

ArcadeDB ships with three built-in strategies, described in the sections below.

`round-robin`

The default strategy when a type has more than one bucket. Each new record is assigned to the next bucket in circular order; after the last bucket the counter wraps back to the first.

This is the right choice when you have no specific reason to choose otherwise. Writes are spread evenly across all buckets without any application-level coordination, and the algorithm has no per-record overhead.

The animation shows six successive inserts arriving at the producer and being routed to `Bucket 0`, `Bucket 1`, `Bucket 2`, `Bucket 0`, `Bucket 1`, `Bucket 2`.

`thread`

The bucket is chosen as `(threadId mod numBuckets)`, so each writer thread always lands in the same bucket. With one bucket per thread, two threads writing concurrently never touch the same bucket file or page, eliminating page-level lock contention between them.

This is the fastest strategy under heavy concurrent inserts. Best results come when the number of buckets is at least equal to the number of writer threads (e.g. one bucket per CPU core; see the `typeDefaultBuckets` setting).

The trade-off is that record distribution depends on how evenly threads insert: if one thread does most of the work, its bucket grows much faster than the others. Pick this when peak write throughput matters more than balanced bucket sizes.

The animation shows three threads firing inserts at the same instant; each thread's record always lands in its own dedicated bucket.

`partitioned(<property>)`

The bucket is chosen by hashing the value of a named property and mapping the hash to a bucket index. Two records that share the same key always land in the same bucket, deterministically.

The big win is on read. When you later query by that property, ArcadeDB knows exactly which bucket can possibly contain the row, so the index lookup hits a single sub-index instead of scanning every bucket's index in turn. For point lookups by key this can be much faster than `round-robin` or `thread`.

The trade-offs:

- Inserts are slightly slower than the other strategies because of the extra hashing step.
- The distribution of records across buckets only stays balanced if the chosen property has high cardinality and reasonably uniform values. A low-cardinality key concentrates writes into a few buckets.
- The `UPSERT` clause on `UPDATE` only guarantees atomicity when the `WHERE` condition matches a `UNIQUE` index, so this strategy is most often paired with one.

The animation shows four records with `id` values `42, 17, 99, 33` flowing through `hash(id) % 3` and

landing in their deterministic destination buckets (`Bucket 0`, `Bucket 2`, `Bucket 0`, `Bucket 0`).

Setting and changing the strategy

The strategy is inherited from a database-wide default (changeable with `ALTER DATABASE`) and can be overridden per type with `ALTER TYPE … BucketSelectionStrategy`. See the SQL reference under Bucket Selection Strategies for the exact syntax.

Schema design 101: choosing a bucket strategy

The default `round-robin` strategy is **safe but unconditionally generic**: it never produces hot buckets, but it also gives the planner zero query-time leverage. Most production schemas have an obvious partition axis (tenant, customer, region, organisation) that the default ignores. Picking a partitioned strategy at schema-creation time is one of the highest-leverage decisions you can make for a single-node deployment.

Use this 3-question decision tree before you create a new type:

1. **Is the data scoped to a tenant, customer, organisation, or other high-cardinality categorical key?** If yes, use `partitioned(<that-key>)`. Multi-tenant SaaS workloads, customer-scoped records, and per-organisation document stores all fit here. The planner will skip every bucket that doesn't match the scope predicate, so a `WHERE tenant_id = X` query touches one HNSW graph (or one LSM-Tree per-bucket sub-index) instead of all of them. This is the optimisation that makes filterable HNSW work for SaaS use cases (see concepts/vector-search.pdf).
2. **Is the data time-series with a hot recent / cold historical split?** If yes, prefer the `LSM_TIMESERIES` type's native partitioning (see Time series) - it understands time-window roll-off and downsampling out of the box. Generic types with `partitioned(timestamp)` can degenerate into a hot bucket for the current period.
3. **Is the workload single-tenant, mixed-cutting analytics, or a generic document store?** Stick with `round-robin` (the default). The other strategies' query-time benefit doesn't materialise without a stable partition axis.

Anti-patterns worth calling out:

- **Low-cardinality keys** (`status` with 3 values, `lang` with 5 values): every value collides into one bucket. With 16 buckets and a 3-value key, ~13 buckets stay empty and the other 3 carry everything.
- **Skewed distributions** (90% of traffic on one tenant): the hot tenant's bucket becomes the bottleneck. Either pre-shard manually or pick a finer-grained key.
- **Mutable partition keys**: the strategy assumes the key value never changes after insert. ArcadeDB refuses an update that would change a partition-key column with a `ValidationException` (HTTP `400`), and the record is left unchanged. *(Since v26.12.1: before, it was an `IndexException` answered with `500`.)* Removing the key is refused the same way, while writing its current value again is accepted. Every other property of the record can be updated, indexed or not. *(Since v26.12.1, issue #12961: before, an update that changed the value of*

any indexed property of a partitioned type was refused, not only one of the partition key.) If you need to "move" a record to a different tenant, drop and re-insert it.

- **Composite partition keys**: `partitioned(orgId, region)` works but only helps queries that bind **both** keys. A query that filters by `orgId` alone touches every bucket.

`needsRepartition` flag

A schema mutation can invalidate the partition mapping for existing records:

- `ALTER TYPE Doc BUCKET +newBucket` on a partitioned type changes the modulus: `hash(tenant_id) % oldCount` no longer matches `hash(tenant_id) % newCount` for the same value. Existing records stay in their old buckets; new records go to the right bucket. The mapping is now stale.
- Same for removing a bucket.
- `ALTER TYPE Doc BucketSelectionStrategy 'partitioned(…)'` on a populated type that previously used `round-robin` puts the records under a hash they were never routed by.
- Changing the strategy's property set (e.g. `partitioned(orgId)` → `partitioned(orgId, region)`) on a populated type has the same effect.

When this happens, ArcadeDB sets a `needsRepartition` flag on the type. The flag is exposed in `SELECT FROM schema:types` and shown as a yellow warning banner in Studio's type-detail panel. **While the flag is set, partition-aware bucket pruning is suppressed**: queries fan out across every bucket and stay correct, but lose the optimisation. The flag is cleared by a successful `REBUILD TYPE … WITH repartition = true`, which walks every record, moves the misplaced ones to their hash-target bucket, and then clears the flag.

You can also chain the rebuild atomically with the schema change:

```
ALTER TYPE Doc BUCKET +newBucket WITH repartition = true
```

This runs the bucket addition and the rebuild as one logical operation; the flag never goes true to subsequent queries. The trailing `WITH` settings are checked before the schema change is applied, so a statement refused for its settings (an unknown setting, a repartition of a vertex or edge type) leaves the type as it was. A rebuild that fails later on the stored records (for example past `rebuild.repartition.maxBufferedRids`) still leaves the bucket or strategy change applied, because schema changes are not transactional: fix the cause and run `REBUILD TYPE … WITH repartition = true` again.

Vertex and edge types: a repartition rebuild deletes each misplaced record from its old bucket and re-inserts it, which gives it a new RID. For graph types this breaks integrity in two ways:

- Inbound edges from other vertices keep pointing at the old RID and silently dangle.
- Outbound-edge segments are stored on the vertex itself, so the delete+re-insert

> wipes them - even on a type that has no inbound references.
>
> There is no safe in-place fix for either, so ArcadeDB refuses `REBUILD TYPE <V> WITH repartition = true` (and the chained `ALTER TYPE <V> … WITH repartition = true`) on vertex or edge types unconditionally. The chained `ALTER TYPE` is refused before its bucket or strategy change is applied, so the type keeps its buckets, its strategy and its `needsRepartition` flag. The supported workflow for repartitioning a populated graph type is to drop and re-import the data under the new partitioning.

Verifying it's working

To check whether a query is benefiting from partition pruning, prefix it with `EXPLAIN`:

```
EXPLAIN SELECT FROM Doc WHERE tenant_id = 'acme'
```

In the resulting plan, look for a `GetValueFromIndexEntryStep` line with a narrow `filtering buckets [N]` value (one bucket id rather than the full set). For Cypher, the partition-pruned path is logged in the `usedIndexName` field as `<label>[partition='<bucket>']`.

The partition key can be written as a literal or bound as a parameter (`WHERE tenant_id = ?`, `MATCH (d:Doc {tenant_id: $t})`): both prune to the bucket of the value. In SQL, the plan of a query pruned on a parameter is built for each execution rather than taken from the plan cache, because the bucket depends on the value bound. *(Since v26.10.2: before, a parameter did not prune, so `vector.neighbors()` and `vector.sparseNeighbors()` searched every bucket and answered differently with a parameter than with the same value written as a literal.)*

6.8.11. Relationships

ArcadeDB supports three kinds of relationships: **connections**, **referenced** and **embedded**. It can manage relationships in a schema-full or schema-less scenario.

6.8.12. Graph Connections

As a graph database, spanning edges between vertices is one way to express a connections between records. This is the graph model's natural way of relationsships and traversable by the SQL, Gremlin, and Cypher query languages. Internally, ArcadeDB deposes a direct (referenced) relationship for edge-wise connected vertices to ensure fast graph traversals.

Example

In ArcadeDB's SQL, edges are created via the `CREATE EDGE` command.

6.8.13. Referenced Relationships

In Relational databases, tables are linked through `JOIN` commands, which can prove costly on computing resources. ArcadeDB manages relationships natively without computing a `JOIN` but storing a direct `LINK` to the target object of the relationship: a property on the source record holds the target's RID, so following the relationship is a single random read rather than a join.

In the example below, the `Customer` record at `#5:23` has a property `invoice` whose value is the link `#10:2`. The `Invoice` record exists independently in its own bucket and keeps its own RID; deleting the customer does not delete the invoice.

Note that referenced relationships differ from edges: references are properties connecting any record while edges are types connecting vertices, and particularly, graph traversal is only applicable to edges.

6.8.14. Embedded Relationships

When using Embedded relationships, ArcadeDB stores the related record *inside* the container record, as the value of one of its properties. These relationships are stronger than referenced relationships. You can represent them as a UML Composition relationship.

Embedded records do not have their own RID, so they cannot be referenced from other records. They are only accessible by traversing the container record's property, and they are stored inside the container's record (not in a separate bucket for the embedded type). Consequently, deleting the container record also deletes its embedded records.

In the example below, the `Account` record at `#5:23` has a property `address` whose value is an embedded `Address` record. The embedded record carries its own properties (`city`, `street`) but has no RID of its own — it lives entirely within the account.

Because the embedded `Address` is reachable as `address` on `Account`, you query its inner properties through that path. The query below returns every account whose embedded address has `city = 'Rome'`:

```
SELECT FROM Account WHERE address.city = 'Rome'
```

6.8.15. 1:1 and n:1 Embedded Relationships

ArcadeDB expresses relationships of these kinds using the `EMBEDDED` type.

6.8.16. 1:n and n:n Embedded Relationships

ArcadeDB expresses relationships of these kinds using a list or a map of links, such as:

- `LIST` An ordered list of records.
- `MAP` An ordered map of records as the value and a string as the key, it doesn't accept duplicate keys.

6.8.17. Inverse Relationships

In ArcadeDB, all edges in the graph model are bidirectional. This differs from the document model, where relationships are always unidirectional, requiring the developer to maintain data integrity. In addition, ArcadeDB automatically maintains the consistency of all bidirectional relationships.

6.8.18. Edge Constraints

ArcadeDB supports edge constraints, which means limiting the admissible vertex types that can be connected by an edge type. To this end the implicit metadata properties `@in` and `@out` need to be made explicit by creating them. For example, for an edge type `HasParts` that is supposed to connect only from vertices of type `Product` to vertices of type `Component`, this can be schemed by:

```
CREATE EDGE TYPE HasParts;
CREATE PROPERTY HasParts.`@out` link OF Product;
CREATE PROPERTY HasParts.`@in` link OF Component;
```

6.8.19. Relationship Traversal Complexity

As a native graph database, ArcadeDB supports index free adjacency. This means constant graph traversal complexity of O(1), independent of the graph expanse (database size).

To traverse a graph structure, one needs to follow references stored by the current record. These references are always stored as RIDs, and are not only pointers to incoming and outgoing edges, but also to connected vertices. Internally, references are managed by a stack (also known as LIFO), which allows to get the latest insertion first. As not only edges, but also connected vertices are stored, neighboring nodes can be reached directly, particularly without going via the connecting edge. This is useful if edges are used purely to connect vertices and do not carry i.e. properties themselves.

See Database for the database lifecycle, URL format, and how to create one.

6.9. Database

A **database** is the top-level container for everything ArcadeDB stores: types, buckets, indexes, records, and the write-ahead log that makes them durable. A single ArcadeDB process — whether an embedded JVM or a standalone server — can host many databases at once; they share the JVM, the network listeners, and the global configuration, but each database is otherwise independent. There is no shared catalog and no concept of joins or transactions across databases: every query, every transaction, and every backup operates on exactly one database.

Each database is one directory on disk. Inside that directory live the bucket files (one file per bucket), the WAL, the schema definition (a JSON file), and any persisted indexes. The database name **is** the directory name, so the easiest way to think about a database is as a self-contained on-disk dataset that you can copy, archive, or move between servers by moving the folder. A server discovers its databases by scanning its data root directory at startup, which is also why dropping a directory in or out of that root is enough to register or unregister a database.

The same on-disk database is reachable from any of the supported entry points concurrently. In an embedded scenario you open it directly from Java with `DatabaseFactory.open()` or `DatabaseFactory.create()`. When run as a server, the same database is reachable through the HTTP/JSON API and through the wire-protocol plugins (PostgreSQL, MongoDB, Redis, Gremlin Server, gRPC) at the same time, all backed by the same storage, transactions, and schema.

The database name must be unique within the hosting process.

6.9.1. Database URL

ArcadeDB uses its own URL format of `<engine>:<db-name>`. The embedded engine is the default and can be omitted, so to open a database on the local file system you can pass the path directly as the URL.

6.9.2. Database lifecycle

A database must be opened before use and closed when you are done with it. The Java `Database` interface implements `AutoCloseable`, so the canonical pattern is a try-with-resources block, which guarantees the WAL is flushed and the files are released even on exception:

```
try (Database db = new DatabaseFactory("/data/mydb").open()) {
  // ... use the database ...
}
```

ArcadeDB automatically closes all opened databases when the process dies **gracefully** — i.e. not killed by force. On Unix/Linux this is the case for `SIGTERM` (Docker exit code `143`) but not `SIGKILL` (Docker exit code `137`). On a forced kill the next start replays the WAL to restore consistency, but you will pay that recovery cost; clean shutdown is always preferable.

The log tells you how the replay ended:

- `Recovery of database '<name>' completed` is logged only when every transaction in the WAL was replayed.
- `Recovery of database '<name>' completed, its WAL files are kept until an fsync of the data files succeeds` means every transaction was replayed but the replayed pages could not be forced to disk. The WAL stays the durable copy of them and is deleted automatically once a later fsync succeeds; a close whose fsync fails too keeps it, and the next open replays it again.
- `Recovery of database '<name>' did not complete` means the replay stopped early. If an error interrupted it (for example the JVM ran out of heap while replaying a large WAL), the open fails, the WAL files are left exactly as they were and the database lock is released: fix the cause and open the database again, even from the same JVM, and the replay starts over. If the WAL itself is damaged (a corrupt record followed by valid ones, or a version gap), the files are moved aside with a `.corrupt` extension for manual inspection and the database opens without the transactions that follow the damage.

6.9.3. Creating Databases

A new database starts empty: no types, no buckets, no indexes. You can create one through any of the supported entry points — they all hit the same underlying storage layer, so the result is identical.

Console

The console is the most direct way: a one-liner you can run from a terminal, a `Dockerfile`, or a CI step.

```
$ bin/console.sh "create database mydb"
```

If the server is already running, point the console at it and use the same command:

```
$ bin/console.sh "connect remote:localhost/mydb root <password>; create database mydb"
```

Studio

Studio exposes a **Create Database** button in the GUI. Useful when you are exploring interactively or onboarding a new team member. Studio calls the HTTP API under the hood, so the result is identical to the curl example below.

HTTP API

The HTTP `server` command creates a database remotely on a running server. Useful for managing databases programmatically across containers, VMs, or Kubernetes pods.

```
$ curl -u root:<password> \
       -H 'Content-Type: application/json' \
       -d '{"command": "create database mydb"}' \
       http://localhost:2480/api/v1/server
```

Java API

When ArcadeDB is embedded in a JVM application, use `DatabaseFactory.create()`:

```
try (Database db = new DatabaseFactory("/data/mydb").create()) {
  // schema setup, initial data, ...
}
```

`create()` fails if the directory already exists; pair it with `exists()` (or use `open()` after the first run) for idempotent setup.

Server start-up (declarative)

The `arcadedb.server.defaultDatabases` setting declares a comma-separated list of databases that **must** exist when the server boots. Any name in the list that is missing on disk is created automatically, optionally with a root user and seed data. This is the cleanest pattern for declarative provisioning — the server's configuration **is** the source of truth for which databases exist.

```
arcadedb.server.defaultDatabases=mydb[root|<password>]
```

Once a database exists, define its schema with `CREATE TYPE` (and the related `CREATE BUCKET` / `CREATE INDEX` commands), or import an existing dataset with `IMPORT DATABASE`.

6.10. Schema

ArcadeDB supports schema-less, schema-full and hybrid operation. A type (document, vertex or edge) must be declared in a database before records can be inserted into it; everything else — properties, indexes, additional buckets, triggers, materialized views — is layered on top of that type. For declared properties, all writes are validated against the property's type and constraints, while undeclared properties are accepted as long as no declared constraint is violated.

This section introduces the building blocks of an ArcadeDB schema: Types, Properties, Buckets, Indexes, Triggers, and Materialized Views. For the low-level on-disk layout (pages, record IDs, bucket files, page versioning), see Storage Internals.

6.10.1. Types

A type is the named "shape" of a record. ArcadeDB has three flavours, all backed by the same storage engine:

- **Document type** — schemaless or schema-full documents (JSON-like records), the analog of a table.
- **Vertex type** — documents that participate in the graph as nodes; carry edges-in / edges-out adjacency lists.
- **Edge type** — documents that connect two vertices, bidirectional by default and traversable in both directions.

Types support single inheritance via the `EXTENDS` keyword and are queryable polymorphically (a query on a parent type sees all subtypes' records). See Inheritance for the details. The SQL command surface is `CREATE TYPE`, `ALTER TYPE`, and `DROP TYPE`.

6.10.2. Properties

A property is a declared field on a type. Declaring a property unlocks per-field validation (data type, NOT NULL, regex, min/max, default), polymorphic queries against the field, and the `EXTERNAL` storage option for heavy values (see External Property Storage below).

Properties are managed via `CREATE PROPERTY`, `ALTER PROPERTY`, and `DROP PROPERTY`.

External Property Storage

By default, every property of a record is serialized inline in the same page as the topology (vertex/edge identity and edges-out/edges-in lists). That keeps reads simple, but it means traversal-only queries pay for every heavy property in cache misses and I/O — even when the projection never touches them. A 4 KB embedding on every vertex pushes useful topology out of the page cache long before the working set is exhausted.

Properties marked `EXTERNAL true` are stored in a paired **external bucket** instead. The primary

record carries only a small pointer (`[bucketId][position]`); the value lives in a separate file that traversal-only queries never read.

Use it when:

- the property is large compared to the rest of the record (vector embeddings, long text, embedded JSON, full-text payloads), and
- most queries do not project the property.

Skip it for small, hot properties — the extra pointer indirection adds a page lookup on every read that does need the value.

Per-bucket pairing

Each primary bucket of a type that has at least one EXTERNAL property gets a paired bucket suffixed with `_ext`. The mapping is persisted in the schema and visible from `SELECT FROM schema:types` under the `externalBuckets` field, e.g. `{ "Person_0": "Person_0_ext" }`. External buckets are internal: they are filtered out of `schema:buckets` by default and rejected by user-facing DML (`INSERT INTO BUCKET:` against an external bucket fails). To inspect them explicitly:

```
SELECT name, purpose FROM schema:buckets WHERE purpose <> 'PRIMARY'
```

External buckets show `purpose: 'EXTERNAL_PROPERTY'`.

`DROP TYPE` drops the paired buckets together with the primary ones, so the EXTERNAL values of a dropped type do not stay on disk. *(Since v26.12.1, issue #12995.)*

(Since v26.10.2) a paired bucket belongs to its type for security and locking:

- It carries the type's permissions (see Groups). A user who may read the type reads its EXTERNAL values, and one refused read, update or delete on the type is refused on the paired bucket too. A user allowed to update the type may store and clear EXTERNAL values through an `UPDATE`, which creates or deletes the external record behind them, without holding `createRecord` or `deleteRecord`. A direct insert into an `_ext` bucket is refused for every user, `root` included.
- `LOCK TYPE` and `LOCK BUCKET` on a primary bucket also lock its paired bucket, and so does a lock on a parent type for its subtypes' paired buckets.

Subtypes and embedded documents

A subtype that inherits an EXTERNAL property has paired buckets of its own, and *(since v26.10.2)* its records store the value there, as the parent's records do in the parent's paired buckets. Subtype records written by an earlier version hold an inherited EXTERNAL value inline. They stay inline after the upgrade, and are read correctly, until they are next written or until you run `REBUILD TYPE <Subtype>`, which moves their values to the paired buckets.

An embedded document has no bucket of its own, so the EXTERNAL values of an embedded document's type are stored inline in the owner record.

Compression

The payload stored in the paired bucket can be LZ4-compressed via the per-property `COMPRESSION` attribute:

- `none` (default) — raw bytes; no compression overhead, no compression savings.
- `lz4` — always LZ4-compressed.
- `auto` — try LZ4 and keep the compressed bytes only when they save more than 10% of the raw size; otherwise fall back to raw. The decision is made per record, so a single property can mix compressed and uncompressed records (text gets compressed, vector embeddings typically fall back to raw).

The compression mode is recorded in the main record's pointer byte, not inside the external blob, so reads dispatch to the right decoder without an extra header lookup.

Migrating existing records

Toggling `EXTERNAL` does not rewrite existing records eagerly. Each record is converted on its next write (insert, update, or upsert), and orphaned external values are reclaimed in the same transaction. To migrate a populated type at once, use `REBUILD TYPE`:

```
REBUILD TYPE Person
REBUILD TYPE Parent POLYMORPHIC
REBUILD TYPE Person WITH batchSize = 1000
```

Tiered storage

External buckets can be placed on a different volume from the primary buckets — for example, topology on fast NVMe and heavy payloads on cheaper bulk storage. Set the database-scope configuration `arcadedb.externalPropertyBucketPath` to the target directory before creating the EXTERNAL property; new paired buckets are created there, in a sub-directory named after the database, instead of in the database directory. Existing external buckets are not relocated when this setting changes; move the files manually if you need to migrate after the fact.

The server-level value (`-Darcadedb.externalPropertyBucketPath=/mnt/bulk`, or the environment) applies to every database and is the directory a single database may choose from. A value set for one database, with `ALTER DATABASE` or `set database setting`, must lie inside that directory or inside the database's own directory, and `ALTER DATABASE` of it needs the `updateSecurity` permission:

```
-- server started with -Darcadedb.externalPropertyBucketPath=/mnt/bulk
ALTER DATABASE `arcadedb.externalPropertyBucketPath` '/mnt/bulk/embeddings'
```

A directory outside both is refused, and so is one reached through `..` or a symbolic link that leads out.

Transactional semantics

- The primary record and its external value are written in the same transaction and WAL group, so commit, rollback, and crash recovery are atomic across both.

- Updating an EXTERNAL property updates the external record in place and does not rewrite the main record bytes.
- Deleting the primary record cascades the delete to every linked external record in the same transaction.
- If a property is toggled `EXTERNAL false`, renamed, or dropped, the orphaned external value is reclaimed on the next write of that record (or by `REBUILD TYPE`).

6.10.3. Buckets

A bucket is the physical storage unit for records of a type. Every type maps to one or more buckets, and each record lives in exactly one bucket. Buckets are append-friendly LSM files on disk; partitioning a type across multiple buckets unlocks per-bucket page locality and lets concurrent writers avoid contending on the same file.

The mapping between records and buckets is controlled by the type's Bucket Selection Strategy. Buckets are managed via `CREATE BUCKET`, `DROP BUCKET`, and `ALTER TYPE ⋯ BUCKET` (see `ALTER TYPE`). For the on-disk page layout and how records and RIDs are stored inside a bucket, see Storage Internals in the reference.

Bucket Selection Strategies

See Bucket Selection Strategy for an animated explanation of how each strategy dispatches new records.

When you create a type, it inherits the bucket selection strategy defined at the database level. By default this is `round-robin`. Change the database default with the `ALTER DATABASE` command, or override it per type with `ALTER TYPE`.

Supported strategies:

Strategy	Description
`round-robin`	Selects the next bucket in a circular order, restarting once complete.
`thread`	Selects the next bucket by using the partition (mod) from the current thread id. This strategy gives the best results in terms of performance if you are using multiple threads and multiple buckets.
`partitioned(<primary-key>)`	Uses the primary key to assign a partition to the record. This allows to speedup the lookup into the index avoiding to search for a key in all the sub-indexes. Use this if you have multiple buckets and you want fast lookup but slower insertions.

Picking the right strategy at type-creation time is one of the highest-leverage decisions you can make for query performance. See Schema design 101: choosing a

bucket strategy for a 3-question decision tree, the anti-patterns to avoid, and the `WITH repartition = true` workflow for changing strategy on a populated type.

6.10.4. Indexes

An index is a secondary data structure that accelerates lookups, range scans, or similarity queries against a property (or composite of properties) on a type. ArcadeDB indexes are first-class schema objects: they live in their own files, are kept in sync transactionally, and survive restart and replication.

Several index implementations are available, chosen by the use case:

- `LSM_TREE` (default) — ordered key index for equality and range queries.
- `HASH_INDEX` — key-only equality lookups; smaller and faster than LSM when range scans are not needed.
- `LSM_VECTOR` — HNSW-based dense vector similarity (cosine, dot-product, Euclidean).
- `LSM_SPARSE_VECTOR` — learned-sparse retrieval (SPLADE, BGE-M3).
- `FULL_TEXT` — Lucene-backed text search with native BM25 ranking.

See Indexes in the concepts section for the algorithms, trade-offs, and the case-insensitive (COLLATE CI) variant. The SQL command surface is `CREATE INDEX`, `REBUILD INDEX`, and `DROP INDEX`.

6.10.5. Triggers

A trigger is a Java callback invoked by the engine on record-level events (before/after create, read, update, delete) for a given type. Triggers run inside the originating transaction, so they can validate, enrich, or reject writes, and propagate changes to external systems (web-socket fan-out, audit logs, derived caches).

The SQL command surface is `CREATE TRIGGER` / `DROP TRIGGER`; see Triggers for the registered-class contract and event-type table.

6.10.6. Materialized Views

A materialized view is a query result persisted as its own type, refreshed either manually, on a schedule, or incrementally as the underlying data changes. Once defined, a view is queried like any other type — the optimizer can rewrite queries to hit the view instead of the base type when the projection and filter match.

See Materialized Views for the refresh modes (manual, incremental, periodic) and the design trade-offs. SQL: `CREATE MATERIALIZED VIEW`, `ALTER MATERIALIZED VIEW`, `REFRESH MATERIALIZED VIEW`, `DROP MATERIALIZED VIEW`.

6.11. Inheritance

Unlike many object-relational mapping tools, ArcadeDB does not split documents between different types. Each document resides in one or a number of buckets associated with its specific type. When

you execute a query against a type that has subtypes, ArcadeDB searches the buckets of the target type and all subtypes.

Declaring Inheritance in Schema

In developing your application, bear in mind that ArcadeDB needs to know the type inheritance relationship.

For example,

```
DocumentType account = database.getSchema().createDocumentType("Account");
DocumentType company = database.getSchema().createDocumentType("Company").
addSuperType(account);
```

A record of a sub type is also a record of its super types, so two rules apply to every link. A type hierarchy cannot have a cycle: a type cannot extend itself or one of its own sub types. And a type can only extend a document type or a type of its own kind: a vertex type can extend a vertex or a document type, an edge type an edge or a document type, and a document type only a document type. A link that breaks either rule is refused with a `SchemaException` and the schema is left as it was. *(Since v26.12.1, issues #12028 and #12029.)* See `ALTER TYPE` for the details.

Using Polymorphic Queries

By default, ArcadeDB treats all queries as polymorphic. Using the example above, you can run the following query from the console:

```
SELECT FROM Account WHERE name.toUpperCase() = 'GOOGLE'
```

This query returns all instances of the types `Account` and `Company` that have a property name that matches `Google`.

How Inheritance Works

Consider an example, where you have three types, listed here with the bucket identifier in the parentheses.

By default, ArcadeDB creates a separate bucket for each type. It indicates this bucket by the `defaultBucketId` property in a type and indicates the bucket used by default when not specified. However, a type has a property `bucketIds`, (as `int[]`), that contains all the buckets able to contain the records of that type. The properties `bucketIds` and `defaultBucketId` are the same by default.

When you execute a query against a type, ArcadeDB limits the result-sets to only the records of the buckets contained in the `bucketIds` property. For example,

```
SELECT FROM Account WHERE name.toUpperCase() = 'GOOGLE'
```

This query returns all the records with the name property set to `GOOGLE` from all three types, given

that the base type `Account` was specified. For the type `Account`, ArcadeDB searches inside the buckets `10`, `13` and `27`, following the inheritance specified in the schema.

6.12. Indexes

An **index** is a secondary data structure that speeds up a specific lookup pattern (equality, range, similarity, text search, spatial predicate) on a property — or on a composite of properties — of a type. Without an index, the query engine has to scan every record of the target type and evaluate the filter on each one. With the right index, it jumps straight to the matching records.

Indexes are first-class schema objects in ArcadeDB: they live in their own files, are kept in sync transactionally with the data, and survive restart and replication.

6.12.1. Why (and when) to add an index

Indexes are not free. Each insert, update, and delete also touches every index defined on the affected property, so a type with five indexes pays roughly five times the write cost of a type with none. The break-even point is the read/write ratio of the workload: if you read a property far more than you write it, an index is almost always worth it; if you mostly append and rarely filter, you can skip it.

Add an index when:

- You repeatedly filter by a property in `WHERE`, `JOIN`, or graph traversal (e.g. `WHERE email = ?`, `MATCH (u {username: $u})`).
- You sort or paginate by a property and need natural ordering (`ORDER BY created_at`).
- You enforce a uniqueness constraint (e.g. no two users with the same email).
- You need a **different** access pattern than ordered scans — vector similarity, full-text search, or spatial containment.

Skip an index when:

- The type is small enough that a full scan is already fast (a few thousand records).
- The property is write-hot and almost never queried.
- The selectivity is poor — indexing a boolean column with two values rarely beats a scan.

6.12.2. Index Types at a Glance

Index	What it does	Use it for	SQL keyword
LSM Tree **(default)**	Ordered key index on disk. O(log N) lookups, native range scans, natural `ORDER BY`.	Equality, range, `ORDER BY`, joins, uniqueness constraints.	`LSM_TREE`
Hash Index	Unordered hash index with O(1) point lookups, no range support.	Primary-key access, JOINs, edge traversal where ordering is irrelevant.	`HASH`

Index	What it does	Use it for	SQL keyword
Dense Vector (HNSW)	HNSW graph over `float32` (or `int8`) embeddings; configurable similarity metric.	Semantic similarity search over text/image/audio embeddings (RAG, recommendation).	`LSM_VECTOR`
Sparse Vector	Inverted posting lists on `(dim_id, weight)` pairs with WAND retrieval.	Learned-sparse retrieval (SPLADE, BGE-M3) with embeddings produced outside the database. For BM25 keyword ranking use the Full-Text index.	`LSM_SPARSE_VECTOR`
Full-Text	Lucene-backed tokeniser + analyser pipeline with native BM25 ranking (the default).	Natural-language keyword search ranked by BM25 relevance, fuzzy matching, "more like this", and the keyword leg of hybrid BM25 + vector search.	`FULL_TEXT`
Geospatial	Quad-key index over WKT geometries.	Spatial predicates (`geo.within`, `geo.intersects`, distance, bounding-box queries).	(geometry property)

Each index can additionally be made **unique** (rejects duplicates), **case-insensitive** (see COLLATE CI), and configured with a **null strategy** (skip nulls, error on null, or index nulls).

6.12.3. LSM Tree (default)

ArcadeDB's default index uses a Log-Structured Merge Tree. New keys land in an in-memory write buffer; when the buffer fills, it is flushed to disk as an immutable sorted page. Background compaction periodically merges these pages so that lookups visit only a handful of files even as the index grows. You can also trigger a compaction on demand with `COMPACT INDEX`, for example to settle an index right after a bulk load. When you need to build a large LSM index over data that is **already loaded**, the sorted bulk build constructs it directly in compacted form and is dramatically faster than the record-at-a-time build.

Why ArcadeDB picks LSM by default:

- **Write-friendly**: inserts hit memory first, no in-place page updates, no random-write amplification.
- **Compaction is cheap**: data is already sorted, so merging two pages is a linear scan rather than a B+Tree rebalance.
- **Range and ordered access are native**: keys are stored in order, so `ORDER BY` and `WHERE x BETWEEN ?` walk the index directly without an in-memory sort.
- **Space-efficient**: keys are packed and compressed, with no half-empty B+Tree pages.

The classic B+Tree (the default in most relational databases) trades these write-side wins for slightly

faster random-access reads on a steady-state index. If your workload is read-only and fits in memory, B+Tree can win small constants; under ingest pressure, mixed read/write, or large datasets on slow storage, LSM almost always wins. For the on-disk format and compaction details, see LSM-Tree Internals in the reference.

Large ranges: physical order and the scan fallback

Fetching records through an index costs one page access per record, in key order, which means in random order on disk. That is cheap for a selective condition and while the type fits the page cache, but a condition that matches a large share of the type - `WHERE shipdate <= '1998-09-02'` over most of a table - is then far slower than simply scanning the type. So, when the statement aggregates the rows (`count()`, `sum()`, `GROUP BY`, ...) or sorts them with an `ORDER BY` the index does not serve, the SQL engine reads the matching index entries alone, before loading any record, and decides:

- when more of them match than a share of the records the type holds, the index is set aside and the rows come from a scan of the type, filtered by the same condition;
- otherwise the matching records are loaded in **physical order** (bucket by bucket, in the order they are stored), so every page is read once, in a forward sweep, however many of its records match.

In SQL both run in parallel wherever a scan of the type would: the scan is the parallel scan, and the records loaded in physical order are shared among its workers, a slice of the sorted record addresses each. An aggregation over them runs in those workers too, together with the conditions the index does not answer. The index entries themselves are read by one thread, so the more workers the scan has, the sooner it wins: the share is `arcadedb.queryIndexMaxSelectivity` (0.6 by default) for a sequential scan, divided by (1 + W) / 2 for a scan on W workers - 24% of the type on 4 workers, 6% on 18. The OpenCypher label scan runs on one thread, so it gives way at the sequential share.

The decision is taken on every execution with the actual parameter values. `PROFILE` shows the choice under `EXTRACT VALUE FROM INDEX ENTRY` (`served by physical order`, with `loaded in parallel` when the workers loaded the records, or `served by full scan`). OpenCypher does the same for a `MATCH` anchored on an index range whose `RETURN` aggregates or has an `ORDER BY`: `PROFILE` then shows `NodeIndexRangeScan` as `served in physical order` or `served by label scan`.

An SQL lookup of a key with many entries (an `=` or an `IN` on a non-unique index) that runs while other transactions commit stays consistent without slowing them down: every row loaded after a commit is checked against the key it was found under, and the key is read again only when one of its records was deleted or moved to another key, to pick up the record that replaced it. No row is returned twice or for a key it no longer holds, and no lock is held, so commits are never stalled by the query. `PROFILE` shows the work under `EXTRACT VALUE FROM INDEX ENTRY` (`rows checked against their key after a commit`, `key read again N times`). An OpenCypher `ORDER BY` on the indexed property itself, under a `LIMIT`, is answered from the index order instead, so the scan stops at the first rows rather than reading the whole range (see keyset pagination). Over a whole label this holds in both directions when the property is declared `MANDATORY` and `NOTNULL` (the constraint is trusted: `ALTER PROPERTY` does not validate vertices written before it), when the only `WHERE` is `x IS NOT NULL`, and for an index created with `NULL_STRATEGY INDEX`, whose null keys are read from the end openCypher sorts them at (last ascending, first descending). Over a nullable property with the

default null strategy the index holds no null key: ascending, the vertices without one follow the index entries, and descending, where nulls sort first, they are found by a scan of the label that keeps only them and precede the index entries read from the top (`PROFILE` shows `index order, descending, after null keys`). That scan stops at the `LIMIT` when enough vertices lack the key, and otherwise reads the label without sorting it; declaring the property `MANDATORY` and `NOTNULL`, or creating the index with `NULL_STRATEGY INDEX`, leaves nothing to scan.

An OpenCypher `WHERE` that is an `OR` of equalities or `IN` lists on indexed properties is answered from the indexes, one seek per property with the vertices found by both returned once; with a disjunct on a property that has no index it scans. A `MATCH` whose filter uses only comparisons, arithmetic, string matching and `IN` on the node's properties (no function calls) scans its label in parallel, like the SQL scan, when the type is large enough and no transaction is open.

A statement whose result can show the order the rows arrive in keeps reading them in key order exactly as before: one that returns its rows as they come, with no aggregation and no `ORDER BY`; one using an order-sensitive aggregate such as `list()` (SQL) or `collect()` (OpenCypher); and a `GROUP BY` with `LIMIT` or `SKIP` but no `ORDER BY`. The adaptive fetch applies to `LSM_TREE` indexes with the default collation and no `BY ITEM`/`BY KEY`/`BY VALUE` property, searched with `=`, `<`, `<=`, `>`, `>=` or `BETWEEN`, and never to a point lookup on a `UNIQUE` index. Setting `arcadedb.queryIndexMaxSelectivity` to 0 turns it off.

LSM indexes come in four flavours, picked at `CREATE INDEX` time:

- `UNIQUE` — rejects duplicate keys.
- `NOTUNIQUE` — allows duplicates; the index returns every record with that key.
- `FULL_TEXT` — see Full-Text Index.
- Special variants for vector and spatial workloads — see the rest of this page.

6.12.4. Hash Index

When you only ever do point lookups (`WHERE id = ?`) and never sort or range-scan, a hash index is the right tool: each lookup is one or two page reads regardless of index size, with none of LSM's compaction overhead.

ArcadeDB uses extendable hashing, a disk-oriented variant:

1. Each key is hashed to a binary code; the index maintains a **global depth** — the number of leading bits used as a prefix.
2. A **directory** maps every $2^{\text{globalDepth}}$ prefix to a **bucket page**. Multiple directory entries can point at the same bucket while the bucket's **local depth** is shallower than the global depth.
3. A lookup hashes the key, reads the directory entry for the matching prefix, and scans the target bucket page: every entry has a 1-byte tag (the low byte of its key hash), so only the entries whose tag matches are compared in full.
4. When a bucket overflows, **only that bucket splits** — the local depth bumps by one, entries are redistributed, and the directory doubles only when the local depth exceeds the global depth.

Use case	Hash	LSM Tree
Point lookup (`WHERE id = ?`)	**Best** — O(1), 1–2 page reads	O(log N), several page reads
JOIN / edge traversal	**Best** — constant-time resolution	Good
Range scan (`WHERE age > 30`)	Not supported	**Best** — ordered iteration
`ORDER BY` on indexed field	Not supported	**Best** — natural ordering
Steady-state insert throughput	Consistent — no compaction	May dip during compaction

Two modes are available: `UNIQUE_HASH` (rejects duplicates) and `NOTUNIQUE_HASH` (allows duplicates). The same null strategies as LSM are supported (`SKIP`, `ERROR`, `INDEX`): with `NULL_STRATEGY ERROR` a record whose indexed property is null or missing fails the commit, as it does on an LSM index.

A hash index answers an equality on its whole key only. A composite hash index on `(a, b)` is used when a query holds both `a` and `b` equal to a value, never for a condition on `a` alone: such a query is answered by a scan or by another index, without an error. With `NULL_STRATEGY INDEX`, a unique index (`UNIQUE` or `UNIQUE_HASH`) accepts any number of records whose key is entirely null, and `WHERE p IS NULL` returns all of them.

The entries of a bucket page are not kept in key order, so an insert appends the entry and one slot instead of shifting the slot directory. The page size defaults to 4 KB for fixed-width keys (numbers, dates, links) and 16 KB when a key column is a `STRING`, `BINARY` or `DECIMAL`; on the workloads measured this made hash inserts as fast as LSM or faster, and point lookups about three times faster. Since 26.10.2 a `NOTUNIQUE_HASH` key keeps its record ids inline in its entry while they take up to a quarter of a page; past that they move to pages of their own, and adding a record to the key writes the last of those pages only. Inserting into a key costs the same whether it holds ten record ids or millions, so `NOTUNIQUE_HASH` also suits low-cardinality columns, and the pages freed by deletions are reused. Hash indexes created before 26.10.2 keep every record id inline, where a key with thousands of records is slow to insert into, until `REBUILD INDEX` moves them to the current layout. An older server refuses to open a database holding a hash index created or rebuilt by 26.10.2 or later, naming the unknown layout version: before such a downgrade, drop the hash indexes and recreate them after it. Hash indexes created by an earlier version with sorted pages keep those pages (and their page size) and keep working; `REBUILD INDEX` moves them to the current layout. The layout version is part of the index file name, and a server that predates it does not check it: it would read the new pages as sorted ones and corrupt the index. Never open a database holding a hash index in the current layout with an older server; after a downgrade, drop and recreate the hash indexes first (the bucket pages also carry a bit in the header that older servers misread). Smaller pages mean more of them (about 16 times more than at 64 KB for the same data), so expect more, smaller files, and a lookup scans the tags of the page, so an explicit page size above 16 KB makes hash lookups slower than the default, not faster; the directory is copied on every doubling and the previous copy stays in the file until the next rebuild.

6.12.5. Dense Vector Index (LSM_VECTOR)

`LSM_VECTOR` indexes dense `float32` (or quantized `int8`) embeddings using an HNSW graph layered on top of the LSM-Tree storage backbone. Use it for semantic similarity over text, image, or audio embeddings produced by a model — e.g. retrieval-augmented generation, "find similar products", or content-based recommendation.

What you get out of the box:

- **Similarity metrics**: `COSINE` (default), `DOT_PRODUCT`, `EUCLIDEAN`.
- **Quantization**: `NONE`, `INT8`, `BINARY`, or `PRODUCT` (index-internal compression that trades a bit of recall for a 4–8× memory reduction and a 2–3× search speed-up).
- **Wire encoding**: store the property as `ARRAY_OF_FLOATS` or as a `BINARY` byte-per-dim payload (`encoding: INT8`) to cut HTTP and bucket size 4× when your model emits int8 natively.
- **Persistent, transactional, replicated**: like every other ArcadeDB index, it survives restart, joins HA replication, and is updated inside the originating transaction.

See Vector Search for the full parameter table (dimensions, `efSearch`, `maxConnections`, beam width, multi-layer HNSW, on-graph storage) and worked examples in SQL, Cypher, and Java.

For wide vertices, declare the embedding property as `EXTERNAL true` so the bytes move out of the main bucket. Traversals that don't project the vector stop paying for it in cache misses. See Store Embeddings in an EXTERNAL Property.

6.12.6. Sparse Vector Index (LSM_SPARSE_VECTOR)

Where dense vectors collapse meaning into a few hundred floats, **sparse** vectors keep only the non-zero positions of a high-dimensional vocabulary — typically tens of thousands of token slots with a handful of non-zero weights per document. They are the natural output of learned-sparse retrieval models (SPLADE, BGE-M3, OpenSearch sparse encoders) and of BM25-as-sparse-vector pipelines, and they excel at exact-term recall where dense embeddings get fuzzy.

`LSM_SPARSE_VECTOR` stores the data as an inverted index — a posting list per dimension keyed by `(dim_id, rid, weight)` — and retrieves with document-at-a-time WAND, pruning postings whose cumulative upper bound cannot beat the current top-K score.

Per-document storage is two parallel array properties on the same record:

- `ARRAY_OF_INTEGERS` of non-zero dimension ids, and
- `ARRAY_OF_FLOATS` of the matching weights (non-negative).

Pair a dense and a sparse index on the same record for **hybrid retrieval**: dense vectors recover paraphrased / semantically close matches, sparse vectors anchor on exact terms, and Reciprocal Rank Fusion combines both rankings without score-scale calibration. See Sparse Vector Search and the surrounding hybrid-retrieval examples for the full reference and worked queries.

6.12.7. Full-Text Index

`FULL_TEXT` is built on top of ArcadeDB's LSM Tree — the index storage, ACID transactions, WAL, background compaction, replication, and HA all come for free from the underlying LSM implementation. Lucene is layered on top only for what it does best: the **analyzer** / **tokenizer** / **stemmer** pipeline that turns a free-text field into the searchable terms the LSM stores.

The result is a Lucene-quality search experience over text — tokenization, stemming, fuzzy match, "more like this" recommendations, relevance ranking — without the operational cost of running a separate Lucene index file format alongside the database.

Highlights:

- **LSM-Tree storage under the hood**: same crash-safe writes, compaction, and replication path as every other ArcadeDB index — no separate Lucene segments to manage.
- **Pluggable analyzer per field** (standard, stop-word, language-specific, custom) for tokenization and stemming.
- **Full Lucene query syntax** at query time: phrase, proximity, fuzzy (`~`), wildcards, boolean operators, and caret term boosts (`term^N`).
- **BM25 relevance scoring** (TF/IDF + document-length normalization, with tunable parameters and per-field boosts; legacy match-count `CLASSIC` scoring is still available) exposed as the `$score` query variable, so you can `ORDER BY $score DESC` and rank results.
- **"More like this"** support to find documents similar to a given record by their token distribution.

For the full create-index syntax, analyzer configuration, query language, scoring, and the MLT (more-like-this) configuration, see Full-Text Index in the data-modeling guide. To tune BM25 see Similarity Models, and to combine BM25 with vector search see Hybrid Retrieval.

6.12.8. Geospatial Index

A geospatial index sits on a property holding a WKT geometry string (Point, LineString, Polygon, MultiPolygon). The index encodes each shape's bounding region with a configurable quad-key precision so spatial predicates touch only the candidate cells instead of every row.

What it accelerates:

- `geo.within`, `geo.contains`, `geo.intersects` and the other `geo.*` spatial predicates.
- Bounding-box queries and "find everything within N km of point P".
- Joins between two spatial types (which features lie inside which polygons).

For the full SQL surface, precision tuning, and worked queries, see Geospatial Index in the data-modeling guide.

6.12.9. Case-Insensitive Indexes (COLLATE CI)

The text indexes above are case-sensitive by default: `"Hello"` and `"hello"` are different keys. Add

`COLLATE CI` to a property in the `CREATE INDEX` statement to fold case at index time. This affects LSM and Hash indexes; it does not apply to vector, sparse-vector, or full-text indexes (which have their own normalization story).

Effect:

- **Plain `=` and `IN` are case sensitive**: `WHERE Name = 'Hello World'` matches only `"Hello World"`, with or without the index. The index narrows the candidates and the condition is checked on the rows it returns, so the answer never depends on whether the optimizer used the index (since 26.10.2, issue #9403). For a case-insensitive lookup write it on `Name.toLowerCase()` (see below).
- **Range queries on the plain property** (`Name >= 'C'`, `Name BETWEEN 'a' AND 'c'`) never use the index. The index holds lower-cased keys and probes with a lower-cased bound, which is not the case-sensitive comparison the predicate asks for, so these conditions are evaluated by a scan and answer exactly as they do without the index (issue #8932). To get a case-insensitive range, write it on `Name.toLowerCase()`, which the index serves (see below).
- **Unique constraints** fold case: a `UNIQUE` index with `COLLATE CI` prevents inserting both `"Admin"` and `"admin"`.
- **Original values are preserved** in the document; only the index key is lowercased (using the root locale).
- **Sorting and aggregates ignore the index**: because the index order is the lowercased order, `ORDER BY`, SQL `min()`/`max()`, Cypher range predicates (`<`, `⇐`, `>`, `>=`), the native `select()` API range operators (including `between`) and Gremlin range `has()` on the property never use a `COLLATE CI` index for ordering, so they return the same result as without the index.

```
CREATE INDEX ON <type> (<property> COLLATE CI) <index-type>
```

Composite indexes can mix policies per property:

```
CREATE INDEX ON Product (Name COLLATE CI, Code) UNIQUE
```

Here `Name` is case-insensitive while `Code` remains case-sensitive.

The query optimizer recognizes `.toLowerCase()` on a property with a `COLLATE CI` index and rewrites the query to use the index directly:

```
SELECT FROM Product WHERE Name.toLowerCase() = 'hello world'
```

So existing applications that fold case in queries get the speedup for free.

The index is used for `=` and `IN` with any value, and the condition is still checked on the rows it returns, so `Name.toLowerCase() = 'HELLO'` (an upper-case value can never match a lower-cased name) correctly returns nothing. Ranges and `BETWEEN` on `Name.toLowerCase()` use the index only when their bounds are literals that are already lower case; with a mixed-case literal or a bound parameter the query is evaluated without the index. The property name in `toLowerCase()` must

match the indexed property exactly, including its case (issue #8560).

When to reach for it:

- **User-facing lookups**: usernames, emails, product names — the end-user expects case-insensitive matching.
- **Deduplication**: a uniqueness constraint that ignores case.
- **Replacing `toLowerCase()` patterns**: the conversion happens once at insert time instead of on every query.

6.12.10. Building a large index over existing data (sorted bulk build)

Currently exposed through the Java API only (no SQL keyword yet).

Creating an index on a type that already holds data has to read every existing record and add its key to the index. The default path inserts those keys one at a time through the normal write buffer, exactly as if the records were being indexed live. That is fine for small and medium types, but it does not scale linearly: as the index grows, each insert does more work and the buffer flushes and compacts repeatedly mid-build, so the effective build rate **degrades as it runs**. On tens of millions of records the index step, not the data load, becomes the bottleneck.

The **sorted bulk build** is an opt-in, offline build strategy for exactly this situation. Instead of inserting keys one at a time, it:

1. scans the records and extracts the same index keys the normal path would (scalars, composites, list/map expansions, nulls, and collation are all preserved);
2. builds bounded, sorted runs in memory and spills them to disk when a configurable memory budget is reached, so the build stays within RAM regardless of dataset size;
3. k-way merges the runs into one globally sorted key stream; and
4. writes that stream directly as compacted on-disk pages.

Because the keys arrive already sorted, this replaces a growing sequence of against-a-growing-tree inserts with a bounded external sort followed by a sequential page write. The output is already in compacted form, so you also skip the post-build `COMPACT INDEX` step. On a 10-million-key index this reaches equivalent compacted output several times faster than the default build followed by a compaction.

```
database.getSchema().buildTypeIndex("Document", new String[] { "uri" })
    .withType(Schema.INDEX_TYPE.LSM_TREE)
    .withUnique(true)                // both unique and non-unique are supported
    .withBuildMode(IndexBuildMode.SORTED)
    .withBuildMemoryBudget(1L << 30) // optional: RAM budget in bytes; 0 (default)
auto-sizes from the heap
    .withBuildMergeFanIn(8)          // optional: number of runs merged per pass
(default 8, min 2)
    .create();
```

All three build options are optional: `withBuildMemoryBudget` bounds the RAM used before spilling (defaults to a fraction of the heap), `withBuildMergeFanIn` sets how many sorted runs are merged per pass, and `withBuildSpillDirectory(Path)` chooses where the temporary spill files are written (defaults to the database directory). They are ignored under `IndexBuildMode.DEFAULT`.

For a **unique** index, uniqueness is enforced directly on the globally sorted stream: two records that produce the same key are adjacent after the sort, so a duplicate is detected in a single pass (across all buckets, with no extra sort or in-memory hash set) and the build throws `DuplicatedKeyException` and publishes nothing. As with online unique indexes, an all-null key is exempt (SQL `NULL != NULL`), while a partial-null composite key is still constrained; the null strategy (`SKIP`, `ERROR`, `INDEX`) behaves exactly as it does for a normal build.

The sorted mode is an **explicit operator choice with strict preconditions**. It is restricted to:

- the `LSM_TREE` index type (unique or non-unique),
- a **single-server, non-replicated** database (not part of an HA cluster), and
- **no active transaction** on the calling thread (the build runs in its own exclusive window).

If any precondition is not met, the build **fails immediately, before creating any file** — it does not silently fall back to the slower default path, so you always get the strategy you asked for. Everything else (online reads and writes, HA replication) is unaffected because unsupported requests are rejected up front. Leave `withBuildMode` unset (or set it to `IndexBuildMode.DEFAULT`) to keep the standard record-at-a-time build.

Parallel construction, an automatic warn-and-fall-back mode, HA/replicated operation, and a SQL syntax are planned as follow-ups.

6.12.11. What can be indexed

ArcadeDB indexes can be defined on properties of nearly any type, with the right index choice per data shape:

Property kind	Indexable	Typical index
Scalar (`STRING`, `INTEGER`, `LONG`, `FLOAT`, `DOUBLE`, `DATE`, `DATETIME`, `BOOLEAN`, and since v26.10.2 the native temporal types `LOCAL_TIME`, `OFFSET_TIME`, `ZONED_DATETIME` and `DURATION`)	Yes	`LSM_TREE` for equality + range, `HASH` for point-only
Composite of N properties	Yes	`LSM_TREE` (ordered prefix scans) or `HASH`
`STRING` for natural language	Yes	`FULL_TEXT`

Property kind	Indexable	Typical index
`ARRAY_OF_FLOATS` / `BINARY` embedding	Yes	`LSM_VECTOR`
`ARRAY_OF_INTEGERS` + `ARRAY_OF_FLOATS` (paired)	Yes	`LSM_SPARSE_VECTOR`
`STRING` containing WKT geometry	Yes	Geospatial index
`LIST` / `ARRAY_OF_*` (per element)	Yes	`LSM_TREE` indexed by values
`MAP`	Yes	`LSM_TREE` indexed by keys or by values

A `MAP` value indexed as a whole (without `BY KEY`/`BY VALUE`) is ordered by its sorted (key, value) pairs, independently of the insertion order of the entries. An index of this kind created with an earlier version must be rebuilt after upgrading (`REBUILD INDEX`).

Unique and non-unique modes are available for the scalar and composite cases. For the exact SQL command syntax, see `CREATE INDEX`; to drop or rebuild after a schema change, see `DROP INDEX` and `REBUILD INDEX`; to force a compaction on demand (for example after a bulk load), see `COMPACT INDEX`.

6.12.12. Looking up a value the property cannot hold

A lookup with a value of a kind the indexed property cannot hold (a date for a `BOOLEAN`, a list or a map for a number, a `BOOLEAN` for a `DATE`) matches nothing, with or without the index:

- an equality or an `IN` answers no rows;
- a range (`>`, `>=`, `<`, `⇐`) through the index answers no rows. The plain scan orders a few of those kinds across kinds by what they convert to (a date against a `DECIMAL` by its epoch number), so it may answer rows where the index declines; the index never answers a row the scan does not.

An openCypher `IN` whose right side is one value, `n.p IN $ids`, has to be a list: a string, a boolean or a map is refused with `IN requires a list on the right side`, whether or not the property is indexed. A list literal (`n.p IN [$a, $b]`) takes each item as one value.

Before v26.11.1 (issue #10541) the index answered these lookups differently from the scan: an equality or an `IN` on an indexed `BOOLEAN`, `DECIMAL` or `DATE` property raised a `ClassCastException` or "Comparison between type N and N not supported", a range with such a value on an indexed `INTEGER`, `LONG` or `DOUBLE` property answered every row of the type, and an openCypher `IN` over a parameter that was not a list answered no rows instead of being refused. A query that relied on any of them now gets the answer it would get without the index.

6.12.13. Which index a SQL query uses when several apply

When more than one index can answer the `WHERE` of a SQL `SELECT`, the planner keeps the indexes that cover the most conditions, so a composite index that covers two conditions wins over a single-property index that covers one. Among indexes that cover the same number of conditions, a lookup of whole keys of a `UNIQUE` index wins: an equality, or an `IN` list, on every property of the index. It answers at most one record per key, so `WHERE key = ? AND tenant = ?` reads the `UNIQUE` index on `key` and filters on `tenant`, even when `tenant` has a `NOTUNIQUE` index of its own, whichever of the two indexes was created first. `EXPLAIN` shows the index chosen.

A `UNIQUE` index that the `WHERE` reaches with a range condition, or with only some of the properties of a composite key, is not a lookup of whole keys and gets no such preference.

Among the indexes still left, a lookup of keys wins over a range, whether the index is `UNIQUE` or not. A lookup names the values to find: an equality, an `IN` list, `CONTAINS`, `CONTAINSANY`, `CONTAINSALL`, `CONTAINSVALUE`, `CONTAINSKEY` or `IS NULL`. A range (`>`, `<`, `BETWEEN`, ...) reads every entry between its bounds, and an open one such as `score > 1` can read the whole index. So `WHERE cat = 'rare' AND score > 1` reads the index on `cat` and filters on `score`, whichever of the two indexes was created first. Two lookups, or two ranges, are still chosen as before. The result is the same whichever index is read, since the whole `WHERE` is applied to the records it returns.

(Since v26.12.1.) Before, the choice between such indexes fell to the first one the type listed when no query statistics were recorded, so the statement could read through the `NOTUNIQUE` index on a property shared by most records and take longer the more records shared the value (51 ms instead of 1.4 ms on 50,000 records, issue #11963), or read the whole range of an open range condition to return a few records (82 ms instead of 1.7 ms on 200,000 records, issue #12282).

6.12.14. See also

- Indexes in the Schema concepts page — the one-paragraph entry point with cross-refs.
- Sorted bulk build — fast offline construction of a large LSM index over existing data.
- LSM-Tree Internals — on-disk format, compaction, page layout.
- Vector Search — full reference for dense and sparse vectors and hybrid retrieval.
- Full-Text Index — analyzers, query syntax, scoring, MLT.
- Geospatial Index — precision tuning and spatial predicates.

6.13. Transactions

A transaction comprises a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions. Transactions in a database environment have two main purposes:

- to provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status

- to provide isolation between programs accessing a database concurrently. If this isolation is not provided, the program's outcome are possibly erroneous.

A database transaction, by definition, must be atomic, consistent, isolated and durable. Database practitioners often refer to these properties of database transactions using the acronym ACID). - Wikipedia

ArcadeDB is an ACID compliant DBMS.

ArcadeDB keeps the transaction in the host's RAM, so the transaction size is affected by the available RAM (Heap memory) on JVM. For transactions involving many records, consider to split it in multiple transactions.

6.13.1. ACID Properties

6.13.2. Atomicity

"Atomicity requires that each transaction is 'all or nothing': if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen." - Wikipedia

6.13.3. Consistency

"The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors do not violate any defined rules." - Wikipedia

ArcadeDB uses the MVCC to assure consistency by versioning the page where the record are stored.

Look at this example:

Sequence	Client/Thread 1	Client/Thread 2	Version of page containing record X
1	Begin of Transaction		
2	read(x)		10
3		Begin of Transaction	
4		read(x)	10
5		write(x)	10
6		commit	10 → 11
7	write(x)		10

Sequence	Client/Thread 1	Client/Thread 2	Version of page containing record X
8	commit		10 → 11 = Error, in database x already is at 11

6.13.4. Isolation

"The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction." - Wikipedia

The SQL standard defines the following phenomena which are prohibited at various levels are:

- **Dirty Read**: a transaction reads data written by a concurrent uncommitted transaction. This is never possible with ArcadeDB.
- **Non Repeatable Read**: a transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).
- **Phantom Read**: a transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction. This happens also when records are deleted or inserted during the transaction and they could become visible during the transaction.

The SQL standard transaction isolation levels are described in the table below:

Isolation Level	Dirty Read	Non repeatable Read	Phantom Read
`READ_COMMITTED` (default)	Not possible	Possible	Possible
`REPEATABLE_READ`	Not possible	Not possible	Possible

The SQL `SERIALIZABLE` level is not supported by ArcadeDB. Asking for it, or for any other level the engine does not provide, is refused on every interface (HTTP, SQL, openCypher, PostgreSQL and gRPC *(since v26.11.1)*) with an error naming `READ_COMMITTED` and `REPEATABLE_READ`, rather than starting the transaction at a weaker level than the one asked for.

Under `REPEATABLE_READ` the pages a transaction reads are kept and read again from there, so a record read twice returns the same content. A record larger than a page spans several pages, and the transaction may already hold its first page from reading another record stored there. If another transaction rewrites the large record before it is read, the version that page belongs to can no longer be read whole, so the transaction reads the complete new version instead, never parts of two versions, and every later read of that record in the same transaction returns that same version. The other record keeps reading as it did. Updating the large record in that transaction fails with a retryable `ConcurrentModificationException`, because the page the transaction holds predates the rewrite: retrying the transaction applies the update on the new version *(since v26.10.2)*.

Counts. Under `REPEATABLE_READ`, `count(*)` on a type and `countType()` repeat inside the transaction

too: the committed record counter of each bucket is read at the first count of the transaction and kept, plus the transaction's own inserts and deletes. Under `READ_COMMITTED` they still follow every commit. *(Since v26.11.1; before, a count followed the commits of other transactions at both levels and could disagree with a scan in the same transaction.)*

Indexes. Under `REPEATABLE_READ`, an index lookup or scan finds every key that was committed before the transaction read the index and has not been removed since, also when the index is compacted while the transaction is open (the automatic compaction after writes, or `COMPACT INDEX`). The uniqueness check of a `UNIQUE` index reads the same view, so a duplicate of such a key is refused. *(Since v26.12.1; before, keys moved by an incremental compaction could be missing from the transaction's lookups, and a duplicate of one of them could be committed.)*

Neither level is a snapshot. A page is pinned when the transaction first reads it, not when the transaction begins. A type scan reads its pages one after another, so at both levels it can observe part of a concurrent commit that spans several pages: the pages it reached before that commit show the old records, the ones it reached after show the new ones. For the same reason a count and a scan of the same type pin at different moments, and can disagree inside one `REPEATABLE_READ` transaction. A reader that needs one committed state of a whole type can take a `LOCK TYPE` on it in its transaction, which keeps the writers of that type out until the transaction ends.

Lost updates. Under `READ_COMMITTED`, reading a record and then writing a value computed from it is protected against a concurrent change: if another transaction committed a change to the record after you read it, saving a property on it fails with a retryable `ConcurrentModificationException`, instead of silently overwriting that change. Retrying the transaction, for example with `database.transaction(…)`, reads the record again and applies the write on the new value.

This holds for records of any size, including documents larger than a page. Likewise, a `DELETE` that selected a record which another transaction then deleted and committed fails with a retryable `ConcurrentModificationException`, so the retry simply no longer finds the record.

A delete is protected the same way against a record that changed after it was read. If another transaction committed a change to the record, or deleted it and an insert then reused its RID, deleting the record read before fails with a retryable `ConcurrentModificationException`, instead of removing whatever now sits at that RID. A SQL `DELETE … WHERE` racing an `UPDATE` that makes the record stop matching is retried and no longer deletes it. Under contention more deletes are retried: `database.transaction(…)` and a single-request command over HTTP retry them, while a transaction spanning several HTTP requests receives the exception at the delete. A record deleted by its RID without being loaded first is deleted as it is found *(since v26.12.1; before, the delete removed whatever the RID held when it reached the record)*.

```
database.transaction(() -> {
  Vertex v = rid.asVertex();
  v.modify().set("n", v.getInteger("n") + 1).save(); // retried, never lost
});
```

Adding or removing edges on a vertex is never refused, and a record kept from an earlier transaction is refreshed as before. The check can be disabled with the `txStaleReadCheck` setting.

This protection comes from the record object: the embedded record keeps the image it was read with, and its `save()` compares that image with the stored one. It does not cover the remote clients (`RemoteDatabase` over HTTP, `RemoteGrpcDatabase` over gRPC, and the drivers of other languages): there the read and the write are two requests, the record that comes back carries no image of what was read, and the `UPDATE` that `save()` sends reads the record again on the server, where under `READ_COMMITTED` it finds the latest committed version. A remote read-modify-write under `READ_COMMITTED` therefore overwrites a change another transaction committed between the read and the write, without an error. To make it safe, either:

- run the transaction under `REPEATABLE_READ` (`database.setTransactionIsolationLevel(Database.TRANSACTION_ISOLATION_LEVEL.REPEATABLE_READ)` before `begin()`, or the isolation level of the `transaction()` call of the drivers): the server keeps what the transaction read, so the write of a record changed meanwhile fails with a retryable `ConcurrentModificationException`, and `transaction(…)` retries it;
- or let the server compute the new value in one statement, so there is no gap between the read and the write: `UPDATE Account SET balance = balance + 50 WHERE @rid = #1:0`.

```
RemoteDatabase database = new RemoteDatabase("localhost", 2480, "bank", "root",
"password");
database.setTransactionIsolationLevel(Database.TRANSACTION_ISOLATION_LEVEL.REPEATABLE_
READ);
database.transaction(() -> {
  MutableDocument account = database.lookupByRID(rid, true).asDocument().modify();
  account.set("balance", account.getInteger("balance") + 50).save(); // retried, never
lost
});
```

Reads of a page the transaction has written. Under `READ_COMMITTED`, once a transaction has written a bucket page, it reads that page from its own copy, taken at its first write, so changes other transactions committed to the records of that page afterwards are not seen by it. If its commit finds that a record it read that way changed meanwhile (or appeared, for a scan of the page), the commit fails with a retryable `ConcurrentModificationException` rather than being merged on the newer page, so nothing decided on a stale read is committed. This covers the edge list of a vertex the transaction added a relationship to: traversing or counting that list afterwards, or checking a unique edge type against it, reads the list as it was before what other transactions appended to it meanwhile, so the commit is refused if they did. A transaction whose reads of the page are still current commits through the merge (see `txPageSlotMerge` in Settings), and so do relationships added without reading the list. A scan that is still running when the transaction writes a page it already read, after another transaction committed to that page, is refused the same way. Reads made by record listeners (triggers) count as the application's reads *(since v26.12.1)*.

A record other transactions **created** on that page after the first write, such as a new chunk of an edge list, is read from the committed page instead of being reported as missing *(since v26.12.1)*. Adding a relationship to a vertex whose edge list was created meanwhile therefore no longer fails with `Edge list … head chunk … not visible yet`. Updating or deleting such a record in the same transaction is refused with a retryable `ConcurrentModificationException`. `REPEATABLE_READ` keeps its snapshot and does not see the new record.

Edges to a deleted vertex. Creating an edge of a `UNIDIRECTIONAL` type to a vertex that another transaction deleted and committed meanwhile fails with a retryable `ConcurrentModificationException`, as it already did for bidirectional edges, so no edge is left pointing at a deleted vertex. A destination that no longer exists when the edge is created, deleted by a transaction committed before, is not a conflict: the edge is refused at once with a `RecordNotFoundException` (`VertexNotFoundException`), as SQL `CREATE EDGE` and every bidirectional type refuse it, since no retry could succeed *(since v26.11.1)*. Likewise, a transaction deleting a vertex fails and can be retried if another transaction committed an edge of a `UNIDIRECTIONAL` type ending in that bucket after the delete looked for its incoming edges.

Traversals during concurrent deletes. A traversal (a Cypher `MATCH`, a SQL `out()`/`in()`, or `getVertices()`/`getEdges()` in Java) that reaches a vertex or an edge another transaction deletes and commits before the traversal reads it fails with a retryable `ConcurrentModificationException`, so `database.transaction(…)` and the HTTP API retry it. A record looked up by its RID, one deleted by the transaction itself, or an edge entry that was already dangling before the traversal started still answers `RecordNotFoundException` *(since v26.12.1; before, every such read failed with `RecordNotFoundException`)*.

Writes during an index change. A SQL write (`INSERT`, `UPDATE`, `DELETE`, `CREATE VERTEX`, `CREATE EDGE`, and the writes of a SQL script) that reads through an index another request drops or rebuilds meanwhile fails with a retryable `ConcurrentModificationException`, HTTP `503`, so `database.transaction(…)`, `COMMIT RETRY` and the HTTP API retry it. A write cannot be planned again halfway, as a `SELECT` is, because it may already have written: the retry rolls the transaction back and runs the statement again from the start, against the new schema, so nothing is applied twice. In Java the cause is a `StaleIndexException`, a subclass of `IndexException`, so code that catches `IndexException` keeps catching it. Other index errors, such as a key too large for a page, are not retryable and are reported as before. A write into a bucket that `ALTER TYPE … BUCKET +` has just added to an indexed type commits normally, even while the request that added it is still open *(since v26.12.1, issue #11835; before, such writes could fail with `400` "Cannot update the index … because the file is invalid" or `500 IndexException`, although a retry succeeded)*.

Using `remote` access all the commands are executed on the server, so out of transaction scope. Look below for more information.

Look at these examples:

Sequence	Client/Thread 1	Client/Thread 2
1	Begin of Transaction	
2	read(x)	
3		Begin of Transaction
4		read(x)
5		write(x)
6		commit
7	read(x)	
8	commit	

At operation 7 the client 1 continues to read the same version of x read in operation 2.

Sequence	Client/Thread 1	Client/Thread 2
1	Begin of Transaction	
2	read(x)	
3		Begin of Transaction
4		read(y)
5		write(y)
6		commit
7	read(y)	
8	commit	

At operation 7 the client 1 reads the version of y which was written at operation 6 by client 2. This is because it never reads y before.

6.13.5. Durability

"Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory." - Wikipedia

Fail-over

An ArcadeDB instance can fail for several reasons:

- Hardware problems, such as loss of power or disk error
- Software problems, such as a operating system crash
- Application problems, such as a bug that crashes your application that is connected to the ArcadeDB engine.

You can use the ArcadeDB engine directly in the same process of your application. This gives superior performance due to the lack of inter-process communication. In this case, should your application crash (for any reason), the ArcadeDB engine also crashes.

If you're connected to an ArcadeDB server remotely, and if your application crashes but the engine continues to work, any pending transaction owned by the client will be rolled back.

Auto-recovery

At start-up the ArcadeDB engine checks to if it is restarting from a crash. In this case, the auto-recovery phase starts which rolls back all pending transactions.

ArcadeDB has different levels of durability based on storage type, configuration and settings.

WAL Flush and Durability

ArcadeDB uses a Write-Ahead Log (WAL) to guarantee transaction durability. The `arcadedb.txWalFlush` setting controls whether the WAL is flushed (fsynced) to disk at commit time:

Value	Behavior	Durability guarantee
`0` (default)	No flush. WAL is written to the OS page cache but not fsynced.	Safe against process crashes (OS page cache survives). **Not safe against power loss or OS crash** - committed transactions may be lost.
`1`	Flush without metadata (`fdatasync`). WAL content is fsynced to the storage device.	Safe against power loss. Recommended for production.
`2`	Full flush (`fsync`). WAL content and all filesystem metadata (including timestamps) are fsynced.	No additional recovery value over `1`: `fdatasync` already flushes the metadata needed to read back the appended records (POSIX). Benchmarks show no measurable performance difference either.

Why write order is not persistence order

At commit time ArcadeDB always issues the WAL `write()` **before** the data pages are published, so the write-ahead ordering holds in program order. With `txWalFlush=0`, however, both writes only land in the OS page cache as dirty pages: the kernel's writeback persists dirty pages based on memory pressure, LRU and expiry timers, **with no ordering guarantee between them**, and the block layer and the drive cache can reorder further. Ordering on the physical medium is only ever enforced by an explicit barrier (`fsync`/`fdatasync`). On a power loss or kernel panic it is therefore possible for a data page to have reached the disk while its WAL record did not - despite the WAL having been written first.

The consequence is worse than losing the most recent transactions: a multi-page transaction can persist some pages and not others, with **no WAL record to replay or repair from** - torn records across pages and indexes pointing at data that never persisted. This is structural corruption, which is why `txWalFlush=1` is the production recommendation: it makes the WAL record durable at commit, after which data pages may lag arbitrarily because recovery replays them from the WAL.

Two technical notes:

- `txWalFlush=1` (`fdatasync`-class) is sufficient for the WAL's append-only pattern: POSIX `fdatasync` also flushes the metadata needed to **read the data back**, which includes the file-size extension of an append. Level `2` adds only timestamp-class metadata, with no recovery value - benchmarks show no measurable difference between the two.
- `txWalFlush=0` **is** safe against process crashes (the OS page cache survives the process), only power loss and kernel panics are at risk.

Data files and WAL deletion

A WAL file is deleted in three places: on a clean close of the database, when the background WAL rotation retires a full WAL file, and at the end of the recovery that replayed it after a crash. Before each deletion ArcadeDB forces to disk the data files that the WAL still protects, because once the WAL is gone the data files are the only copy of those pages.

Only the files that need it are forced: a data file is forced when a page was written to it, or when it was created or renamed, since its last successful fsync. A file that was only read is skipped, since it holds nothing that is not already on disk. A file with page writes only is forced with `fdatasync`, which also persists the file-size change of an append. A created or renamed file is forced with a full `fsync`, followed by an fsync of its directory, so its directory entry survives a power loss too. If an fsync fails, the WAL is kept, and the next pass retries the fsync. If a clean close cannot complete it, the WAL and the lock file stay on disk, and the next open runs recovery. Since v26.11.1 the same happens on the close of a database fenced for recovery (after a commit failed past its WAL write): its WAL is always kept, so the reopen replays it. The pages no WAL describes, those of a transaction run without the WAL (for example a bulk load), are still written to disk while the database is fenced, since the reopen could not replay them.

Group commit

Since v26.11.1, with `arcadedb.txWalFlush` set to `1` or `2`, concurrent commits share one WAL flush instead of each flushing on its own (group commit, setting `arcadedb.txWalGroupCommit`, on by default). A commit still returns only once its WAL is on disk, but it no longer holds the locks of the files it wrote while its WAL is written and flushed, so the commits that arrive meanwhile are written and flushed together. On an Apple silicon laptop, 50 clients writing into the same type went from about 230 to about 4,000 commits per second with `txWalFlush=2`. A single writer sees no difference. The engine metrics `wal.group.*` and `commit.lock.*` show how many commits each flush served and how long a commit holds its locks.

What changes compared to the previous versions:

- A committed transaction becomes visible to other transactions a few milliseconds before its WAL is flushed, as with `txWalFlush=0`. After a crash in that window (power loss, operating system or JVM crash), the transaction is lost: its writer got an error or no answer, but another reader may have seen it. If your application cannot accept that, set `arcadedb.txWalGroupCommit=false`: a transaction then becomes visible only once its WAL is on disk, and the commits into the same type wait for each other's flush.
- A data page is never written to disk before the WAL that describes it is flushed.
- If the shared flush fails, the commits waiting for it fail and the database is fenced: close and reopen it to run recovery.
- It applies to databases that are not replicated (embedded, or served by a single server) with the default asynchronous page flush. The commits of an HA cluster are not affected, see below.

WAL flush in an HA cluster

A replicated commit is made durable by the Raft log: a majority of the nodes flushes the entry before the commit completes, and a restarted node replays every entry past its last snapshot from

that log. Since v26.12.1 the leader therefore writes the WAL record of a replicated commit without flushing it, whatever `arcadedb.txWalFlush` says, as the other nodes already applied it with no WAL record at all. Before, with `txWalFlush=1` (what production mode sets), the leader flushed the record of every commit, one at a time, on the single thread that applies the entries of every database of the node, which halved the commit rate of a cluster. The `forcedWALWrites` entry of the transaction manager's statistics (`DatabaseInternal.getTransactionManager().getStats()` in Java) counts the commits that flushed their own WAL record.

WAL files opened on demand

Since v26.11.1 a database no longer opens its whole pool of WAL files when it opens (one per core by default). A WAL file is opened on the first commit that needs it, so `arcadedb.txWalFiles` is now the maximum size of the pool, not the number of files always open. The WAL housekeeping also closes and removes a WAL file that received no writes for `arcadedb.txWalIdleTimeout` milliseconds (60 seconds by default), once all its pages reached the data files. A database that is only read, or not written for that long, holds no WAL file open. Set `arcadedb.txWalIdleTimeout` to 0 or a negative value to keep every WAL file open until the database is closed.

The WAL housekeeping of all the databases of the JVM runs on one shared scheduler of two daemon threads, named `ArcadeDB WAL Housekeeping-N`, instead of one `java.util.Timer` thread per database (named `ArcadeDB TransactionManager <db>`) as before. On a 16-core machine a server that hosted thousands of databases held a thread and about 18 file descriptors per database just for the WAL (issue #10234).

Up to version 26.9.x, a clean close forced every data file of the database, including the files that nothing had written. Each of those fsyncs costs a flush of the device write cache, about half a millisecond per file on a laptop NVMe drive and more on network or cloud block storage. So the close of a database with 100 indexed types took about 100 ms even after a read-only session. Now a close after a read-only session forces no file. After a crash, recovery forces every data file once, because it is not known which writes of the crashed process reached the disk.

Measured cost

Commit-heavy benchmark (one small insert per transaction, so the WAL flush dominates; Apple silicon laptop, macOS):

Level	TPS (1 thread)	Latency (1 thread)	TPS (8 threads)	Latency (8 threads)
`0` (no flush)	24,813	40 us	68,772	116 us
`1` (fdatasync)	282	3.5 ms	587	13.6 ms
`2` (full fsync)	305	3.3 ms	620	12.9 ms

The dramatic gap on macOS is platform-specific: Java's `force()` issues `F_FULLFSYNC` there, which drains the entire drive write cache (~3 ms on Apple SSDs). On Linux server hardware with NVMe, `fdatasync` typically costs 30-100 us, so the production penalty is far smaller than this laptop measurement suggests. Reproduce on your

own hardware with `mvn -pl engine test -Dtest=WALFlushBenchmark` from the ArcadeDB source tree.

In `production` server mode (`arcadedb.server.mode=production`), ArcadeDB automatically sets `txWalFlush=1` if you have not explicitly configured it. This ensures that production deployments are durable by default. If you explicitly set `txWalFlush=0` in production mode, a warning is logged at startup. Note the server mode **defaults to** `development`, so a server that never sets the mode - and any embedded application - runs with `txWalFlush=0`: set the mode (or the setting) explicitly for durability.

The WAL flush setting can also be changed per-database or per-thread via the Java API:

```
// Per-database: every thread's transactions, from now on
database.setWALFlush(WALFile.FlushType.YES_NOMETADATA);

// Per-thread: this thread's transactions, from now on, whatever the database setting
says
database.getTransaction().setWALFlush(WALFile.FlushType.YES_FULL);

// Hand the thread back to the database setting
database.getTransaction().setWALFlush(null);

// Per async executor
database.async().setTransactionSync(WALFile.FlushType.YES_NOMETADATA);
```

`setUseWAL()` and `setAsyncFlush()` follow the same rule: on `Database` they apply to every thread, on `getTransaction()` to the calling thread only, and the thread's own setting wins. Until either is called, the `arcadedb.txWAL` and `arcadedb.txWalFlush` configuration applies.

A transaction committed without the WAL has no record to replay, so a failure while its pages are being written (a full volume, an I/O error) cannot be repaired. *(Since v26.11.1)* Such a failure fences the database, as one past the WAL append does, and the records the transaction created keep their identities: part of the transaction may be in the files, and a retry must not insert it again. Close and reopen the database, then run `CHECK DATABASE`. In 26.10.2 and earlier the commit was rolled back as if nothing had been written (issue #10768).

`Database.setWALFlush()`, `setUseWAL()` and `setAsyncFlush()` are database-wide (issue #8352). An application that needs a per-thread scope, for example a bulk loader turning the WAL off on its own thread while other threads keep writing, must call the same setter on `database.getTransaction()` instead.

If your storage hardware has battery-backed write cache (BBU/BBWC) or power-loss protection (common in enterprise SSDs and cloud block storage like AWS EBS), `txWalFlush=0` is safe even in production because the hardware guarantees that buffered writes reach persistent storage on power loss.

6.13.6. Optimistic Transaction

This mode uses the well known Multi Version Control System MVCC by allowing multiple reads and writes on the same records. The integrity check is made on commit. If the record has been saved by another transaction in the interim, then an `ConcurrentModificationException` will be thrown. The application can choose either to repeat the transaction or abort it.

ArcadeDB keeps the whole transaction in the host's RAM, so the transaction size is affected by the available RAM (Heap) memory on JVM. For transactions involving many records, consider to split it in multiple transactions.

A `ConcurrentModificationException` always means another transaction wrote the record: it is never raised because of something the transaction did to itself. In particular, updating a record that the same transaction has already deleted is not a conflict, it simply has no effect - the delete wins, since the record will not exist once the transaction commits. The order of the two operations does not matter. Deleting a record that the same transaction has already deleted raises `RecordNotFoundException`, and changes nothing else. This now holds for an edge as well as for a vertex and a document *(since v26.12.1)*: a second delete of the same edge, or a delete of an edge that already went with its deleted vertex, could be accepted and left the record count of the edge type one lower than the records it holds.

6.13.7. A statement that fails halfway

A statement that fails inside a transaction you keep is taken back on its own, and the transaction goes on as if the statement had never run, the way SQLite undoes a failed statement. Take a multi-row `INSERT INTO T (id, k) VALUES (1, 7), (2, 7)` on a unique `k`: the first row is written, the second is refused, and your code catches the error. Nothing of the first row remains: not the record, not its entries in the unique and plain indexes (vector and full-text ones included), not the edges and the edge lists of both vertices when the statement was a `CREATE EDGE`, not the record counters. A record an earlier statement of the transaction updated, and the failed one changed again, holds again what the earlier statement gave it. `commit()` then publishes the rest of the transaction. This is true of a statement in any language: `INSERT`, `UPDATE`, `DELETE` and `CREATE EDGE` in SQL, `SET`, `CREATE` and `DELETE` in openCypher (including the ones Bolt runs inside `BEGIN`), and a `sqlscript`.

A statement that runs on its own, with no transaction begun, is rolled back whole as before, and pays nothing for the feature. A `sqlscript` whose writes are all inside a `BEGIN ⋯ COMMIT RETRY n ELSE ⋯ AND CONTINUE | FAIL` block handles its own failures and is not affected. The events a statement already fired (listeners, triggers, whatever an external system was told) cannot be taken back, and neither can a schema change: it takes effect at once, outside the transaction, as it always did.

When the engine cannot take a statement back exactly, it does not take it back at all, and the transaction can no longer be committed: `commit()` throws a `TransactionException` ("Transaction cannot be committed: a statement failed after part of its records were written ..., and could not be taken back: ...") that names the reason, and nothing of the transaction is published. Roll the transaction back and retry it, or issue the rows as separate statements. That is the case for:

- a statement that changed more than `arcadedb.txStatementSavepointMaxEntries` entries of the transaction (100,000 by default), so that a bulk load pays no memory for a journal it could not

use. Set it to `0` to turn statement savepoints off;

- a statement that changed records a Graph Analytical View follows, or that dropped a file;
- a statement that changed again a record with an `EXTERNAL` property that an earlier statement updated, or any record an earlier statement updated that cannot be kept or that the statement changed without reading it through the transaction;
- a statement that attached state to the transaction (other than the full-text counters, which are taken back), replaced or created again a page of the transaction, retracted a record that an earlier statement created, dropped or replaced an index lane, or ended the transaction under it;
- a statement that a function or a trigger runs through the database, when it fails after writing and the outer statement catches the failure and carries on: the outer statement succeeded, so there is nothing to take back to;
- a Gremlin traversal, which runs while its result is read, after the call has returned, and so is covered at that point by refusing the commit.

An update of a record that the same transaction already updated, which a before-update listener refuses by returning `false` (skipping the record) instead of throwing, is refused the same way: no statement fails, so there is nothing to take back, and the refused values stay on the copy of the record the transaction writes at commit.

(Since v26.11.1: earlier versions committed the rows written before the error. Issue #10194 refused the commit of such a transaction, in every language since issue #10379; statement savepoints, issue #10346, take the statement back instead.)

6.13.8. Nested transactions and propagation

ArcadeDB does support nested transaction. If a `begin()` is called after a transaction is already begun, then the new transaction is the current one until commit or rollback. When this nested transaction is completed, the previous transaction becomes the current transaction again.

6.14. Data Types

ArcadeDB supports several data types natively. Below is the complete table.

Type	SQL type	Description	Java type	Minimum — Maximum	Auto-conversion from/to
Boolean	BOOLEAN	Handles only the values **True** or **False**	`java.lang.Boolean` or `boolean`	0 — 1	String
Integer	INTEGER	32-bit signed Integers	`java.lang.Integer` or `int`	-2,147,483,648 — +2,147,483,647	Any Number, String
Short	SHORT	Small 16-bit signed integers	`java.lang.Short` or `short`	-32,768 — 32,767	Any Number, String

Type	SQL type	Description	Java type	Minimum — Maximum	Auto-conversion from/to
Long	LONG	Big 64-bit signed integers	`java.lang.Long` or `long`	-2^{63} — $+2^{63}-1$	Any Number, String
Float	FLOAT	Decimal numbers	`java.lang.Float` or `float`	2^{-149} — $(2-2^{-23})*2^{127}$	Any Number, String
Double	DOUBLE	Decimal numbers with high precision	`java.lang.Double` or `double`	2^{-1074} — $(2-2^{-52})*2^{1023}$	Any Number, String
Datetime	DATETIME	Any date with the precision up to milliseconds. To know more about it, look at Managing Dates	`java.util.Date`		Date, Long, String
Local time	LOCAL_TIME	A time of day without an offset, e.g. `10:15:30`. *(Since v26.10.2.)* See Managing Dates	`java.time.LocalTime`		String
Offset time	OFFSET_TIME	A time of day with a UTC offset, e.g. `09:00:00-05:00`. *(Since v26.10.2.)*	`java.time.OffsetTime`		String
Zoned datetime	ZONED_DATETIME	A datetime that keeps the zone or offset it was written with, e.g. `2024-01-01T10:00:00+01:00[Europe/Rome]`. *(Since v26.10.2.)*	`java.time.ZonedDateTime`		String

Type	SQL type	Description	Java type	Minimum — Maximum	Auto-conversion from/to
Duration	DURATION	An openCypher duration with months, days, seconds and nanoseconds, e.g. `P1M2DT3S`. *(Since v26.10.2.)*	`com.arcadedb.query.opencypher.temporal.CypherDuration`		String
String	STRING	Any string as alphanumeric sequence of chars	`java.lang.String`		
Binary	BINARY	Can contain any value as byte array	`byte[]`	0 — 2,147,483,647	String
Embedded	EMBEDDED	The Record is contained inside the owner. The contained record has no RID	`EmbeddedDocument`		EmbeddedDocument
Embedded list	LIST	The Records are contained inside the owner. The contained records have no RIDs and are reachable only by navigating the owner record	`List<EmbeddedDocument>`	0 — 41,000,000 items	String

Type	SQL type	Description	Java type	Minimum — Maximum	Auto-conversion from/to
Embedded map	MAP	The Records are contained inside the owner as values of the entries, while the keys can only be Strings. The contained records have no RIDs and are reachable only by navigating the owner Record	`Map<String, EmbeddedDocument>`	0 — 41,000,000 items	`Collection<? extends EmbeddedDocument<?>>`, `String`
Link	LINK	Link to another Record. It's a common one-to-one relationship	`RID`, `<? extends Record>`	1:-1 — 32767:2^{63} -1	String
Byte	BYTE	Single byte. Useful to store small 8-bit signed integers	`java.lang.Byte` or `byte`	-128 — +127	Any Number, String
Decimal	DECIMAL	Decimal numbers without rounding	`java.math.BigDecimal`		Any Number, String
Vector	ARRAY_OF_?	Vector of scalar types: SHORTS, INTEGERS, LONGS, FLOATS, DOUBLES. A full type is for example then ARRAY_OF_FLOATS			Tuples

Embedded types, like `EMBEDDED` (document), (embedded) `LIST`, and (embedded) `MAP` are non-scalar types.

A vector type can be either `LIST OF SHORT|INTEGER|LONG|FLOAT|DOUBLE` or `ARRAY_OF_SHORTS|INTEGERS|LONGS|FLOATS|DOUBLES`.

An `ARRAY_OF_FLOATS` or `ARRAY_OF_DOUBLES` value is stored fixed-width: the element count followed by the raw little-endian IEEE-754 elements, 4 bytes per float and 8 per double *(since v26.11.1)*. Before, each element was written as a zigzag varint of its bit pattern, 5 bytes per float and 9 to 10 per double. A record holding a 1536-dimension float embedding shrinks from about 7,695 to about 6,150 bytes, so 9 such records fit a 64 KiB bucket page instead of 7, about 22% less disk and page cache. Existing records need no migration: the old form is still read, and an update rewrites the value in the new form. Once a database holds records in the new form, a build older than 26.11.1 cannot read them (see Upgrade). In an HA cluster the new form is written only once every peer advertises the `fixed-width-float-arrays` capability, so a rolling upgrade keeps writing the old form until the last node runs 26.11.1 (see High Availability).

A declared `ARRAY_OF_SHORTS|INTEGERS|LONGS|FLOATS|DOUBLES` property takes a list or an array of numbers, primitive (`float[]`, `int[]`, ...) or boxed (`Float[]`, `Object[]`, ...), and converts it to the declared element type, refusing an item out of its range. Since v26.11.1 any other value is refused with an error naming the property: a scalar (`embedding = 0.5`), a string (`'[0.1, 0.2]'`), a map, a list with a `null` or non-numeric item, or a number the element type cannot hold (`Double.MAX_VALUE` for an `ARRAY_OF_FLOATS`, which used to become `Infinity`). Before, such a value was stored as it came, and depending on whether a vector index on the property already existed, the write failed or the row was silently missing from an index created afterwards. Records written before the upgrade are read back unchanged.

An equality lookup (`=`, `<>`, `IN`) on a `FLOAT` or `DOUBLE` property converts the operand to the property's type, with or without an index. A `FLOAT` holding `0.1` is therefore found by the double `0.10000000149011612` (what you get by reading the `FLOAT` back as a double), and a `DOUBLE` is found by the `BigDecimal` that rounds to it. Ordering (`<`, `>`) is unchanged and reads a `FLOAT` by its decimal value, so a value can satisfy both `=` and `<` against the same operand.

A write to a declared property is refused, with an error naming the property, when the value cannot be held by the type: a boolean, list, map or date for a numeric property, an empty string for a numeric property, a number outside the 64-bit range for a `LONG` (or a finite number that overflows a `FLOAT` or `DOUBLE`), an empty or digit-only string for a `DATETIME`, a string for a `BINARY`, a non-link value for a `LINK`, or an array for a `STRING`. A `STRING` holding a lone UTF-16 surrogate (for example `"\uD83D"`, which cannot be encoded as UTF-8) is refused with a validation error instead of being stored as `?`. Such a value is never stored as `NULL`, `0` or as a value of another type, with or without an index. A number written to a `BOOLEAN` is `false` only when it is `0`.

A `java.math.BigInteger` written to a property with no declared type keeps its exact

value: inside the 64-bit range it is stored as an `INTEGER` or `LONG`, outside it as a `DECIMAL` (it reads back as a `BigDecimal` with the same digits), never as a rounded `DOUBLE`.

Equality lookups give the same answer with or without an index. Two `DECIMAL` values are equal when they have the same numeric value, whatever their scale (`19.9` equals `19.90`). A `DATETIME` property is matched by a `java.time.Instant`, `ZonedDateTime` or `OffsetDateTime` of the same moment. A `BOOLEAN` property is matched by `true`/`false` as a string or by a number (`0` is false, anything else is true); an operand that does not read as a boolean matches nothing.

When serializing floating point types `FLOAT` and `DOUBLE` to JSON, `NaN`, `Inf`, and `-Inf` are cast to `null`, wherever they appear, including inside lists and arrays.

Embedded `MAP` vs `EMBEDDED` Document

A `MAP` type and an `EMBEDDED` type are both a hierarchy of key-value pairs (think JSON); but the `EMBEDDED` type needs to know which document type it is an instance of. Hence, to embed a document, a document type needs to be declared beforehand, such that upon embedding validity of constrains (if existing) can be checked.

The embedded type is taken from either of two places:

- a `@type` property in the value's top-level, naming an existing document type, or
- the property's own declared embedded type, when the property was created with one (`CREATE PROPERTY Person.address EMBEDDED OF Address`).

If the property declares an embedded type, a plain map is enough and you do not have to repeat `@type`:

```
person.set("address", Map.of("city", "Rome"));
```

A `@type` in the value still takes precedence where present, which is how you store a subtype of the declared one. If neither is available, the value is rejected.

6.15. Binary Types (BLOB)

While some DBMSs (like OrientDB) specifically support BLOBs (Binary Large OBjects) record types, with ArcadeDB the binary content must be stored as a property in documents, vertices and edges.

To create a BLOB like type, you can define a "Blob" document type and add a property of type `BINARY`. Then you can set and retrieve the binary content as byte array (`byte[]`). Example:

```
database.transaction(() -> {
  // DEFINE THE BLOB TYPE
  DocumentType blobType = database.getSchema().createDocumentType("Blob");
  blobType.createProperty("binary", Type.BINARY);
```

```
  ...
  // STORE SOME BINARY CONTENT IN THE DOCUMENT PROPERTY
  final MutableDocument blob = database.newDocument("Blob");
  blob.set("binary", "This is a test".getBytes());
  blob.save(); // THE DOCUMENT IS SAVED IN THE DATABASE ONLY WHEN `.save()` IS CALLED
  ...
  // RETRIEVE THE BINARY CONTENT FROM THE DOCUMENT PROPERTY
  final Document blob = database.iterateType("Blob", false).next().asDocument();
  byte[] binaryContent = blob.getBinary("binary");
});
```

6.16. Managing Dates

ArcadeDB treats dates as first class citizens. Internally, it saves dates in the Unix time format. Meaning, it stores dates as a `long` variable, which contains the count in milliseconds since the Unix Epoch, (that is, 1 January 1970).

ArcadeDB fully supports the new Java Time API with a custom precision from second to nanoseconds. The data types to manage date/time are:

- `DATE`, to handle dates without a time. Example: `01/01/2023`
- `DATETIME_SECOND`, to handle datetime with the precision of the second. Example: `01/01/2023 10:30:00`
- `DATETIME`, to handle datetime with the precision of the millisecond. Example: `01/01/2023 10:30:00.333`
- `DATETIME_MICROS`, to handle datetime with the precision of the microsecond. Example: `01/01/2023 10:30:00`
- `DATETIME_NANOS`, to handle datetime with the precision of the nanosecond. Example: `01/01/2023 10:30:00`

using high precision types increase the space used on disk. For example, using nanosecond precision instead of millisecond increase the space requested to store the datetime of a factor of 2X.

(Since v26.10.2) a value outside the range its precision can represent is refused with an error instead of being stored as a different instant. Each type stores a 64-bit count of its unit since the epoch, so `DATETIME_NANOS` holds instants from 1677-09-21 to 2262-04-11 and `DATETIME_MICROS` about 292,000 years either way. Before, an instant past the range - for example a "no expiration" date such as `9999-12-31` written to a `DATETIME_NANOS` property - was stored as `2262-04-11T23:47:16.854775807`. Comparing values is unaffected: a date past the range still compares after every date inside it.

Dates Inside Lists and Maps over HTTP

A datetime stored inside a `LIST` or a `MAP` is returned by the HTTP API as a formatted string that keeps

its full precision (for example `"2026-10-03 12:34:56.123457"`), exactly like a scalar datetime property.

Datetimes in the JSON of a Record

`record.toJSON()` writes a `DATETIME` and a `DATETIME_SECOND` property as a number of epoch milliseconds, and a `DATETIME_MICROS` and a `DATETIME_NANOS` property as formatted text that keeps all its digits (for example `"2020-01-02 03:04:05.123456789"`), and `fromJSON()` of that text gives the value back for every precision. A number is epoch milliseconds whichever the precision of the property is, as it is in SQL and over HTTP. Before 26.11.1 (issue #10486) `toJSON()` wrote a `DATETIME_SECOND`, `DATETIME_MICROS` or `DATETIME_NANOS` property as a count in its own unit, which `fromJSON()` read as milliseconds, so a 2020 timestamp came back in 1970, in the year 51972 or in the year 50004715; the remote client sends a record as this JSON, so it stored such a property wrong on the server too. A reader of the JSON that took those counts as seconds, microseconds or nanoseconds, for example a WebSocket subscriber of the change events, must read the milliseconds and the text instead. *(Available since v26.11.1)*

A `DATE` in JSON is a number of epoch milliseconds at UTC midnight, as long as that number fits a 64-bit integer, which is a date within about 292 million years of 1970 (up to `+292278994-08-17` and from `-292275055-05-17`). A date beyond it (`LocalDate.MAX`, `LocalDate.MIN`) is written as its ISO text (for example `"+999999999-12-31"`), where `toJSON()` and the `jsonl` export used to fail with an arithmetic overflow, so a reader of the JSON of such a record sees text instead of a number for that value. *(Available since v26.11.1)*

Precision on Undeclared Properties

If a property is not declared in the schema, ArcadeDB stores every datetime at the precision of the value itself, so a microsecond value keeps its microseconds without any declaration. *(Since v26.12.1)* This holds for a nanosecond value outside 1677-09-21..2262-04-11 too, the range of a `DATETIME_NANOS` count: `localdatetime("9999-12-31T23:59:59.999999999")` is stored exactly and reads back as a local datetime. Before 26.12.1 it was refused with "outside the range a nanos-precision timestamp can represent", and before 26.11.1 it was stored as `2262-04-11T23:47:16.854775807`. A database holding such a value cannot be opened by a version older than 26.12.1.

Creating an index or a uniqueness constraint on such a property with Cypher (`CREATE INDEX`, `CREATE CONSTRAINT ⋯ IS UNIQUE`) leaves it undeclared, so every value keeps its own precision (see property types of a Cypher index). *(Since v26.12.1)* An existence constraint (`IS NOT NULL`, `IS NODE KEY`) declares it in the schema, and the type is inferred from the data already stored: ArcadeDB picks the highest precision it finds, so existing microsecond values give `DATETIME_MICROS`. Milliseconds are the minimum, so a property holding only whole seconds is declared `DATETIME` and not `DATETIME_SECOND`. Before 26.12.1 `CREATE INDEX` and `IS UNIQUE` declared the property the same way.

To choose the precision yourself rather than relying on the inference, declare the property before creating the index:

```
CREATE PROPERTY Event.created_at DATETIME_MICROS
```

If you're working with the Java API, by default ArcadeDB uses the new Java Time API on

deserialization: `DATE` values are returned as `java.time.LocalDate` and `DATETIME` values as `java.time.LocalDateTime`. These defaults are controlled by the `arcadedb.dateImplementation` and `arcadedb.dateTimeImplementation` settings.

The supported implementations are:

- for `DATE`: `java.time.LocalDate` (default), `java.util.Date`, `java.util.Calendar`, `java.time.LocalDateTime` (midnight of the day)
- for `DATETIME`: `java.time.LocalDateTime` (default), `java.util.Date`, `java.util.Calendar`, `java.time.ZonedDateTime`, `java.time.Instant`

`java.util.Date` and `java.util.Calendar` can only handle precision down to the millisecond. When `arcadedb.dateTimeImplementation` is set to one of them, `DATETIME_MICROS` and `DATETIME_NANOS` values are returned as `java.time.LocalDateTime`, so no precision is lost; `DATETIME` and `DATETIME_SECOND` values are returned as the configured class.

before 26.10.2, `arcadedb.dateImplementation=java.time.LocalDateTime` read every `DATE` as a timestamp in January 1970 and the value read could not be written back. A `DATE` now reads as midnight of its day, and writing a `LocalDateTime` to a `DATE` property keeps only the day. The data on disk was never affected. *(Since v26.10.2)* openCypher reads a property declared `DATE` as a Cypher `Date` under every `arcadedb.dateImplementation` (`java.util.Date`, `java.util.Calendar` and `java.time.LocalDateTime` included), and a `DATETIME` as a datetime under every `arcadedb.dateTimeImplementation`, so `WHERE n.d = date('2024-02-29')`, inline patterns, ranges and `toString(n.d)` (`2024-02-29`) behave the same whichever Java class the setting picks.

stored `DATETIME` values keep no time zone. In an openCypher comparison a stored value takes the zone of the datetime it is compared with, so `e.ts = datetime('2026-01-01T11:00:00Z')` and `e.ts = datetime('2026-01-01T12:00:00+01:00')` both match the same instant, whatever `arcadedb.dateTimeImplementation` is set to. Before 26.10.2, with `java.time.ZonedDateTime` a stored value did not equal the `Z` literal for its own instant, and with `java.util.Date` or `java.time.Instant` it matched `Z` but not the same instant in another zone.

`DATE` properties written by an earlier version with `arcadedb.dateImplementation` set to `java.util.Date` or `java.util.Calendar` may hold a timestamp rather than a day, and the extra time of day is still there when you read them.

If you prefer the legacy `java.util.Date` (for example, for compatibility with existing code), you can change the defaults at the database level:

```
alter database `arcadedb.dateTimeImplementation` `java.util.Date`
alter database `arcadedb.dateImplementation` `java.util.Date`
```

(Since v26.10.2) a `REGEXP` on a `DATE` or `DATETIME` property is matched against the text of the class the value is held as, so it depends on these settings: `2024-01-15` with `java.time.LocalDate`, `Mon Jan 15 00:00:00 CET 2024` with `java.util.Date`. Changing `arcadedb.dateImplementation` or `arcadedb.dateTimeImplementation` is refused when a pattern already declared would match no value of the new class, naming the properties: change or remove the pattern first (`ALTER PROPERTY … REGEXP null`). In the same way, changing `arcadedb.dateFormat` or `arcadedb.dateTimeFormat` is refused when an existing `MIN` or `MAX` of a date property could no longer be read with the new format. *(Since v26.11.1)* the same check runs when the class changes any other way, such as a `-Darcadedb.dateImplementation=java.util.Date` on the server command line or the Java API: opening the database cannot be refused, so it logs a `WARNING` naming each property whose `REGEXP` can no longer match any value. Remove the pattern with `ALTER PROPERTY … REGEXP null`, or set the setting back.

(Since v26.11.1) a number written into a `DATE` property is read the same way whatever `arcadedb.dateImplementation` is: the epoch milliseconds of a UTC midnight are that day (what a client sends for a `LocalDate` query parameter); any other number up to about 1.07e11 either side of zero is a count of days since 1970-01-01; a larger number is epoch milliseconds, and the `DATE` holds the UTC day they fall in. So `1705312345678` (a "now" in milliseconds) is stored as `2024-01-15` with every implementation. Before 26.11.1 the default `java.time.LocalDate` refused it, and a number between about 1.07e11 and 3.65e11 was a day hundreds of millions of years away with `java.time.LocalDate` and refused with `java.util.Date` and `java.util.Calendar`; it is now a day between 1973 and 1981. A comparison is not affected: there a number is still a count of days.

Time of Day, Zoned Datetime and Duration Types

(Available since v26.10.2) Four more types hold what the openCypher `time()`, `localtime()`, `datetime()` and `duration()` functions return, without losing the offset, the zone or the months/days parts:

- `OFFSET_TIME`, a time of day with a UTC offset (`java.time.OffsetTime`). Example: `09:00:00-05:00`
- `LOCAL_TIME`, a time of day without an offset (`java.time.LocalTime`). Example: `10:15:30`
- `ZONED_DATETIME`, a datetime with the zone or offset it was written with (`java.time.ZonedDateTime`). Example: `2024-01-01T10:00:00+01:00[Europe/Rome]`
- `DURATION`, an openCypher duration with months, days, seconds and nanoseconds (`DURATION` values read back as `com.arcadedb.query.opencypher.temporal.CypherDuration`). Example: `P1M2DT3S`

A `DURATION` given as text, to `duration('…')` or to a `DURATION` property, is read with the ISO 8601 grammar Neo4j applies: `P[nY][nM][nW][nD][T[nH][nM][nS]]`, or the date-based `PYYYY-MM-DDThh:mm:ss`. Each component is a signed integer, only the last one present may carry a fraction (`P1.5Y`, `PT1,5S`), a sign before the `P` negates the whole duration (`-P1Y2D`), and the text must carry at least one component: `P`, `PT` and `P1YT` are refused *(since v26.10.2)*.

openCypher stores these four natively, so a value always reads back as what it was written as, with no guessing from its text. A String that merely looks temporal, such as the opening hours `09:00-17:00` or a code like `P100D`, is a String and stays a String (on a database upgraded from a version

before v26.10.2, only once it is migrated, see below). Before 26.10.2 these four types were stored as Strings and openCypher guessed their type when reading an undeclared property, so `09:00-17:00` came back as a time and `WHERE l.hours = '09:00-17:00'` matched nothing.

All four can be indexed, with both `LSM_TREE` and `HASH`. A `ZONED_DATETIME` and an `OFFSET_TIME` compare as in openCypher and Neo4j: the same instant in two zones is two different values, so `12:00:00+01:00` does not equal `11:00:00Z`, with or without an index, and a unique index accepts both. Values are ordered by instant, then by UTC offset from west to east, then by zone name.

If a property is declared with one of these types and a record holds a String for it, the String is converted when the record is read and saved in the native type the next time the record is written. A String that cannot be converted is read as it is.

(Since v26.12.1) A value written to an `OFFSET_TIME`, `LOCAL_TIME` or `DURATION` property is converted to the declared type, or the `INSERT`, `UPDATE`, `CREATE` or `SET` fails with an error naming the property, as it does for `DATE`, `DATETIME` and `ZONED_DATETIME`:

- `OFFSET_TIME` takes a time with an offset as it is. A time of day without an offset (`localtime()`, or text such as `'10:00'`) is taken at UTC. A datetime keeps its time of day and its offset: `datetime('2020-01-02T10:00+01:00')` is stored as `10:00+01:00`, and a `localdatetime()` as its time at UTC.
- `LOCAL_TIME` takes a local time as it is. A value that carries an offset is read on the UTC clock, as a `DATETIME` reads one: `time('10:00+01:00')` and the text `'10:00+01:00'` are stored as `09:00`. A datetime keeps its time of day, on the UTC clock when it carries an offset.
- `DURATION` takes a duration, its ISO 8601 text, a `java.time.Duration` or a `java.time.Period`.

Anything else is refused: text that does not parse, a number, a boolean, a list, a `date()` and, for a `DURATION`, any point in time or time of day. Before 26.12.1 an unparseable String was stored as `NULL` (on a `NOT NULL` property the write failed with "cannot be null"), a number, a boolean, a list or a date was stored as it was, and another temporal kind failed with an internal `ClassCastException`.

a zoned `datetime()` written to a property declared `DATETIME` is stored as an instant and reads back as the UTC wall clock, like any other `java.time` value. Before 26.10.2 it kept the local wall clock of its zone.

In JSON, these four types are written as their ISO text, for example `"2026-01-01T00:30+02:00"` for a `ZONED_DATETIME` and `"00:30+02:00"` for an `OFFSET_TIME`, also inside a `LIST` or a `MAP`. This is what the HTTP API always returned, and *(since v26.10.2)* `Result.toJSON()`, the SQL `asJSON()` method and the PostgreSQL wire protocol agree with it. Before 26.10.2 they wrote a `ZONED_DATETIME` as epoch milliseconds, which lost the zone, and an `OFFSET_TIME` as `null`.

Migrating Cypher Temporals Stored as Strings

A database written by a version before v26.10.2 holds these values as Strings in undeclared properties. Such a database keeps the old reading after the upgrade: openCypher still turns a String with the shape of a time, a zoned datetime or a duration on an undeclared property into a temporal value when it reads it, as before, and a new database created with v26.10.2 does not. The legacy reading ends when you run the one-off migration:

```
CALL db.migrateLegacyTemporals() YIELD records, properties
```

The procedure rewrites every String with a temporal shape on a property the schema does not declare into its native type, in batches of 1,000 records each committed in its own transaction, and then marks the database native, so from then on a String is a String. It yields the number of `records` and `properties` it rewrote. A declared property and the elements of a list are left alone. If it fails part way the database stays legacy, and running it again picks up what is left; on a database that is already native it converts nothing and yields `0` and `0`.

- It is one-way, and it converts every String with a temporal shape on an undeclared property, including one that was meant as text. Take a backup first, and quiesce writers while it runs: a String written meanwhile is converted too.
- It changes a database setting, so the user needs the `updateDatabaseSettings` permission.
- Under HA, run it on the leader. A follower refuses the schema change, while a request sent to a follower over HTTP is forwarded to the leader. The rewritten records and the native flag replicate to every follower.

To convert only some properties, or a property you want to keep declared, declare the property with its type and copy the String into it instead. `UPDATE Place SET hours = hours` does not do it: with the property declared the read has already converted the String, so the value looks unchanged and ArcadeDB skips the write.

The script parks the String in a scratch property, declares the property, and copies the String back. Run it with `sqlscript`, once per property, replacing `Place`, `hours` and `OFFSET_TIME` (use `LOCAL_TIME`, `ZONED_DATETIME` or `DURATION` for the other types):

```
BEGIN;
UPDATE Place SET hours_old = hours;
UPDATE Place REMOVE hours;
COMMIT;

CREATE PROPERTY Place.hours OFFSET_TIME;

BEGIN;
UPDATE Place SET hours = hours_old WHERE hours_old IS NOT NULL;
UPDATE Place REMOVE hours_old;
COMMIT;
```

Only run it on a property that holds temporal Strings, and take a backup first. Do not remove `hours_old` until you have checked that every record has its `hours` value (`SELECT count(*) FROM Place WHERE hours_old IS NOT NULL AND hours IS NULL` must be `0`). On a large type, run the `UPDATE` statements in batches with `LIMIT`.

Date and Datetime Formats

In order to make the internal count from the Unix Epoch into something human readable, ArcadeDB formats the count into date and datetime formats. By default, these formats are ISO 8601:

- Date Format: `yyyy-MM-dd`
- Datetime Format: `yyyy-MM-dd HH:mm:ss`

A `DATE` is a calendar day, not an instant, so it is always rendered as the day it was stored: the result does not change with the time zone of the server that answers the request. Only `DATETIME` values carry a time of day.

When a `DATE` is written as a JSON number (for example in the result of a projection such as `SELECT day FROM Event`), it is the milliseconds of that day at midnight UTC, whatever the time zone of the server.

Accepted Input Strings

The configured formats above are what ArcadeDB **renders**. On input, a string assigned to a `DATE` or `DATETIME*` property is accepted in any of these spellings, tried in order:

1. ISO 8601 without a zone, e.g. `2024-02-29T13:45:10.123456`;
2. ISO 8601 with an offset or `Z`, e.g. `2024-02-29T13:45:10.123456+01:00`;
3. the database's `DATETIMEFORMAT`;
4. the database's `DATEFORMAT`;
5. the SQL timestamp spelling: an ISO date, a **space**, an optional time with an optional fractional second of any precision, and an optional offset - e.g. `2024-02-29 13:45:10.123456`, `2024-02-29 13:45:10.123456+01`, `2024-02-29 13:45`, `2024-02-29`. This is what PostgreSQL prints and what `psqlodbc` binds a timestamp as.

The database's own formats are tried before the last one, so customizing them can never be overridden by it.

How an offset in the string is treated depends on the type the value is stored as, which is set by `arcadedb.dateTimeImplementation`. With the default `java.time.LocalDateTime` the string is stored as the UTC wall clock of the instant it names, exactly as the same instant is when a client sends it typed (a Java `OffsetDateTime` or `ZonedDateTime`, a Bolt `DateTime` with an offset or a zone id), so `13:45:10+01:00` is stored as `12:45:10`. A string with no offset, and one in UTC (`Z` or `+00:00`), keeps its wall clock, so `13:45:10Z` is stored as `13:45:10`. With an instant-bearing type such as `java.time.Instant` or `java.util.Date` the offset is honoured and the value is stored as the instant it names. A `DATE` has no time of day to move: the day written is the day stored, whatever the offset.

Before 26.11.1 a string with an offset was stored with the offset dropped (`13:45:10+01:00` as `13:45:10`) while the same instant typed as an `OffsetDateTime` was converted to UTC, so a client writing over HTTP or SQL and one writing over Bolt stored values hours apart. Values written before are not changed; rewrite them if they came from a client in another time zone. The `date()` function and the `.asDatetime()` method read a string the same way.

The rule holds when the caller names the pattern, too: a format that reads an offset or a zone out of the text (`XXX`, `Z`, `VV`, `z`) answers the instant it names, so `'2020-01-02T03:04:05+02:00'.asDateTime("yyyy-MM-dd'T'HH:mm:ssXXX")` and `date('2020-01-02T03:04:05+02:00',`

"yyyy-MM-dd'T'HH:mm:ssXXX") are 2020-01-02T01:04:05, the value the schema's own pattern gives for the same text. .asDate() keeps the day written. The third argument of date(), a time zone, is where a text that names none was written, and changes nothing for one that does. *(Since v26.11.1; before, a named pattern dropped the offset it had read, and the time zone argument was ignored.)* The functions date.parse() and date.fields() and the datetimeFormat of the importer read an offset the same way.

The database's own time zone (setZoneId()) plays no part in where an instant is stored: the same rule holds on every write path, the bulk ones included (GraphBatch, the gRPC service, a TimeSeries column). *(Since v26.11.1)* the conversion the gRPC service and the other callers of DateUtils.dateTimeToTimestamp apply to a string no longer moves an offset onto the database's zone first; before, in a database whose zone was not UTC, a string such as 2020-01-02T03:04:05+02:00 was stored hours away from the same instant written through SQL, and 2020-01-02T03:04:05Z shifted by the zone's offset. A UTC database stores the same value as before.

A string that matches **none** of these is rejected: the INSERT or UPDATE fails with an error naming the property. It is never stored as NULL. The same applies to the date() function and the .asDatetime() method when called without an explicit format, except that date() keeps its documented NULL answer for a value it cannot read.

Every format, built in or configured, and every pattern passed to .asDate(), .asDatetime() or date(), is read strictly: a day or an hour that does not exist is refused, never moved to one that does. '2021-02-30' and '2021-04-31' are not dates, and '2021-02-03 24:00:00' is not a datetime: a write of one fails, and .asDate(), .asDatetime() and date() answer NULL for it. A comparison with such a literal, as in WHERE day = '2021-02-30', matches no record.

In a pattern, yyyy is the proleptic year, the year the ISO formats read, unless the pattern also names the era (G): so '0000-01-01'.asDate('yyyy-MM-dd') is the year 0 (1 BC), exactly as '0000-01-01'.asDate(), and formats back as 0000. A pattern with a 12-hour field (hh, KK) reads the AM/PM marker a when it has one, and the morning when it does not.

Before 26.11.1 an impossible date was moved to the last day of its month (2021-02-30 read as 2021-02-28) and 24:00:00 to midnight of the next day, a pattern with yyyy could not read the year 0, and no pattern with hh ⋯ a could be parsed at all.

Comparing and Subtracting Dates

A DATE compared with a DATETIME (of any precision) is that day at midnight, as SQL compares a date with a timestamp: with day = 2021-02-03 and time = 2021-02-03 12:00, day < time is true and day = time is false, whichever side of the operator each is on, and with an index on day or without one. Before 26.11.1 the datetime was narrowed to its day when it was on the right of the operator, so day = time was true there and false the other way round.

A DATETIME (of any precision) compared with a number reads the number as epoch milliseconds, and compares it exactly, whichever side of the operator it is on: with time at 2,000 milliseconds after the epoch, time = 2000 and 2000 = time are true, time < 2000.5 and 2000.5 > time are true, time = 2000.5 is false, and time ⇐ 1e19, time > -1e19 are true (a number beyond the range of a long is above or below every datetime; a NaN is above every datetime, as it is above every number). An index on the DATETIME follows the same rule, so the indexed and the unindexed query return the

same rows. `BETWEEN` and a `TIMESERIES` timestamp bound follow the same rule, so `DELETE FROM sensor WHERE ts >= 2000.5` removes exactly the samples `SELECT FROM sensor WHERE ts >= 2000.5` returns. Before 26.11.1 a number on the left of the operator matched no datetime (`2000 <> time` was true for the datetime at 2,000 milliseconds, `2000 = time` false), a fractional number was read as the millisecond it falls in (`time = 2000.5` and `time >= 2000.5` were true for the datetime at 2,000 milliseconds), and a number beyond the range of a `long` matched nothing. In a `TIMESERIES` type, `ts >= 2000.5` selected the sample at 2,000 milliseconds, `DELETE ⋯ WHERE ts = 2000.5` removed it, and `ts ⇐ 1e19` selected nothing. *(Available since v26.11.1)*

Subtracting two dates answers the exact duration between them, whatever their precision: `day - time` with a `DATETIME_NANOS` operand three centuries away no longer overflows.

A time field in a `.format()` pattern formats a `DATE` as its midnight: `day.format('yyyy-MM-dd HH:mm:ss')` answers `2021-02-03 00:00:00`, where it used to throw.

In the event that these default formats are not sufficient for the needs of your application, you can customize them through `ALTER DATABASE ⋯ DATEFORMAT` and `DATETIMEFORMAT` commands. For instance,

```
arcadedb> ALTER DATABASE DATEFORMAT "dd MMMM yyyy"
```

This command updates the current database to use the English format for dates. That is, 14 Febr 2015.

SQL Functions and Methods

To simplify the management of dates, ArcadeDB SQL automatically parses dates to and from strings and longs. These functions and methods provide you with more control to manage dates:

SQL	Description
`date()`	Function converts dates to and from strings and dates, also uses custom formats.
`sysdate()`	Function returns the current date.
`.format()`	Method returns the date in different formats.
`.asDate()`	Method converts any type into a date.
`.asDatetime()`	Method converts any type into datetime.
`.asLong()`	Method converts any date into long format, (that is, Unix time).
`.precision()`	Method modifies the precision of a datetime property. For example, `.precision('millisecond')` returns the datetime property with millisecond precision

For example, consider a case where you need to extract only the years for date entries and to arrange them in order. You can use the `.format` method to extract dates into different formats.

```
arcadedb> SELECT @RID, id, date.format('yyyy') AS year FROM Order
```

```
+--------+----+------+
| @RID   | id | year |
+--------+----+------+
| #31:10 | 92 | 2015 |
| #31:10 | 44 | 2014 |
| #31:10 | 32 | 2014 |
| #31:10 | 21 | 2013 |
+--------+----+------+
```

In addition to this, you can also group the results. For instance, extracting the number of orders grouped by year.

```
arcadedb> SELECT date.format('yyyy') AS Year, COUNT(*) AS Total FROM Order ORDER BY
Year
```

```
+------+--------+
| Year |  Total |
+------+--------+
| 2015 |      1 |
| 2014 |      2 |
| 2013 |      1 |
+------+--------+
```

Dates before 1970

While you may find the default system for managing dates in ArcadeDB sufficient for your needs, there are some cases where it may not prove so. For instance, consider a database of archaeological finds, a number of which date to periods not only before 1970 but possibly even before the Common Era. You can manage this by defining an era or epoch variable in your dates.

For example, consider an instance where you want to add a record noting the date for the foundation of Rome, which is traditionally referred to as April 21, 753 BC. To enter dates before the Common Era, first run the [`ALTER DATABASE DATETIMEFORMAT`] command to add the `GG` variable to use in referencing the epoch.

```
arcadedb> ALTER DATABASE DATETIMEFORMAT "yyyy-MM-dd HH:mm:ss GG"
```

Once you've run this command, you can create a record that references date and datetime by epoch.

```
arcadedb> CREATE VERTEX V SET city = "Rome", date = DATE("0753-04-21 00:00:00 BC")
```

```
arcadedb> SELECT @RID, city, date FROM V
```

```
+-------+------+-----------------------+
| @RID  | city | date                  |
+-------+------+-----------------------+
| #9:10 | Rome | 0753-04-21 00:00:00 BC |
+-------+------+-----------------------+
```

Using `.format()` on Insertion

In addition to the above method, instead of changing the date and datetime formats for the database, you can format the results as you insert the date.

```
arcadedb> CREATE VERTEX V SET city = "Rome", date = DATE("yyyy-MM-dd HH:mm:ss GG")
arcadedb>  SELECT @RID, city, date FROM V
```

```
+------+------+-----------------------+
| @RID | city | date                  |
+------+------+-----------------------+
| #9:4 | Rome | 0753-04-21 00:00:00 BC |
+------+------+-----------------------+
```

Here, you again create a vertex for the traditional date of the foundation of Rome. However, instead of altering the database, you format the date field in `CREATE VERTEX` command.

Viewing Unix Time

In addition to the formatted date and datetime, you can also view the underlying count from the Unix Epoch, using the `asLong()` method for records. For example,

```
arcadedb> SELECT @RID, city, date.asLong() FROM #9:4
```

```
+------+------+-----------------------+
| @RID | city | date                  |
+------+------+-----------------------+
| #9:4 | Rome | -85889120400000       |
+------+------+-----------------------+
```

Meaning that, ArcadeDB represents the date of April 21, 753 BC, as -85889120400000 in Unix time. You can also work with dates directly as longs.

A number with a fractional part is rounded down to the unit that contains it (for a datetime in milliseconds, `-1.5` is millisecond `-2`), and a number that is not a real instant (`NaN`, `Infinity`, or a value outside the supported range) is refused with an error instead of being stored as 1970-01-01 or

as an unrelated date.

```
arcadedb> CREATE VERTEX V SET city = "Rome", date = DATE(-85889120400000)
arcadedb> SELECT @RID, city, date FROM V
```

```
+-------+------+-----------------------+
| @RID  | city | date                  |
+-------+------+-----------------------+
| #9:11 | Rome | 0753-04-21 00:00:00 BC |
+-------+------+-----------------------+
```

Use ISO 8601 Dates

According to ISO 8601, Combined date and time in UTC: 2014-12-20T00:00:00. To use this standard change the date time format in the database:

```
ALTER DATABASE DATETIMEFORMAT "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"
```

Arithmetic with dates

Dates can be added and subtracted. If you want to know the difference in terms of seconds between two dates, you can use the `-` (minus) operator. Example:

```
SELECT sysdate() - lastActivity as secondsFromLastActivity FROM UserActivity
```

Returns `1212113.232000000`.

if the date supports the fractional part of the second, then it's returned as nanoseconds as decimal part. In the example above, `sysdate()` function returns a datetime with, by default, precision to the millisecond.

The difference is a duration (a `java.time.Duration` in Java). Over HTTP and in JSON it is written as its exact number of seconds, to the nanosecond. *(Since v26.12.1)* it is a number of seconds in SQL too, so it can be compared and computed with:

- compared with a number, the number is a count of seconds, on either side of the operator and with `=`, `<>`, `<`, `<=`, `>`, `>=`, `IN` and `BETWEEN`: `WHERE sysdate() - lastActivity < 3600` selects the activities of the last hour. A duration also compares with another duration: `WHERE sysdate() - lastActivity < duration(1, 'hour')`;
- `+` and `-` with a number of seconds or with another duration answer a duration: `(sysdate() - lastActivity) + 60`;
- `*`, `/` and `%` with a number scale it into a duration: `(sysdate() - lastActivity) / 3600` is the hours, shown as their number. A duration divided by a duration is a plain number: `(sysdate() - lastActivity) / duration(1, 'hour')`;

- a datetime plus or minus a duration is the datetime that much later or earlier, of the same type: `sysdate() - duration(7, 'day')` is a week ago. A `DATE` moved by whole days stays a `DATE`, and is otherwise that day at midnight moved by the duration;
- a datetime plus or minus a number moves it by that many **milliseconds**, the unit a number has against a datetime everywhere else in SQL (comparisons, a number written into a `DATETIME` property, JSON): `lastActivity + 1000` is one second later. Prefer a `duration()`, which names its unit.

Adding two datetimes, subtracting a datetime from a number or from a duration, multiplying two durations and dividing a duration by zero are refused with an error. A result out of the range of a datetime or of a duration is refused too.

Before 26.12.1 a duration compared greater than every number and less than none, so `WHERE sysdate() - lastActivity < 3600` never matched; `+` with a number concatenated the two as text (`'PT24H1'`); `-`, `*`, `/` and `%` with a number answered `null`; a datetime plus or minus a number answered a duration counted from 1970 instead of a datetime; and a datetime plus or minus a `duration()` failed with an internal error. JSON wrote a duration as a floating point number, which lost the last digits of a duration longer than about 100 days.

Time Units

The units of time used in the `duration()` function and `precision()` method are the following strings:

- `'year'`
- `'month'`
- `'week'`
- `'day'`
- `'hour'`
- `'minute'`
- `'second'`
- `'millisecond'`
- `'microsecond'`
- `'nanosecond'`

6.17. Materialized Views

A materialized view is a schema-level object that stores the result of a SQL `SELECT` query as a backing document type. Unlike regular views (which re-execute the query on every access), a materialized view holds a pre-computed snapshot of data that can be queried directly for fast reads.

A materialized view:

- Wraps a SQL `SELECT` query as its *defining query*
- Stores results in a backing `DocumentType` with standard buckets

- Supports three *refresh modes*: manual, incremental (post-commit), and periodic (scheduled)
- Persists its definition and metadata in `schema.json` alongside other schema objects
- Can be created, dropped, refreshed, and altered via SQL DDL statements or the Java Schema API

6.17.1. Refresh Modes

MANUAL

The view data is never automatically updated. You must trigger a refresh explicitly via `REFRESH MATERIALIZED VIEW` or the Java API. Use this when you control refresh timing yourself or when source data changes infrequently.

INCREMENTAL

After every committed transaction that modifies a source type, ArcadeDB automatically refreshes the view in a post-commit callback. The refresh is:

- **Transactionally safe**: runs after the source transaction commits successfully; rolled-back transactions do not trigger a refresh
- **Batched per transaction**: multiple record changes in a single transaction result in one refresh, not one per record
- For **simple queries** (single source type, no aggregates, no `GROUP BY`, no subqueries, no `JOIN`s): performs a full refresh
- For **complex queries** (aggregates, `GROUP BY`, etc.): also performs a full refresh (per-record incremental optimization is planned for a future release)

If the refresh fails, the view is marked `STALE` and a warning is logged. A manual refresh can recover it.

After a crash. A process killed between a commit and its post-commit refresh leaves the source change committed and the view without it. After such an unclean shutdown, every `INCREMENTAL` view gets a full refresh in the background when the database opens: the view reports `STALE` from the moment the open returns until that refresh ends, so it never reports `VALID` while it may lack a commit. After a clean shutdown only the views saved with a status other than `VALID` get that refresh. In a highly available cluster the same refresh runs on the node that becomes the leader, and followers leave it to the leader. `MANUAL` and `PERIODIC` views are not refreshed at open. *(Since v26.12.1, issue #12220. Before, only a view maintained record by record got this refresh: a view refreshed as a whole, such as a `GROUP BY` with an `ORDER BY`, kept its old rows, often reporting `VALID`, until the next commit on a source type.)*

Which types are tracked. An `INCREMENTAL` view is refreshed when a transaction changes a record of any type its query names, anywhere in the query: the top-level `FROM`, a subquery in `FROM`, and a subquery in `WHERE`, in the projection or in `LET`, at any depth. Each of these types is tracked together with all its subtypes, including subtypes created after the view. They are listed in the `sourceTypes` column of `schema:materializedViews`, and `DROP TYPE` on any of them is refused while the view exists.

Changing the subtypes of a source type. When a type that holds records is attached below a

source type, or a type above it, or detached from it (`ALTER TYPE <type> SUPERTYPE +<source>` or `SUPERTYPE -<source>`, or the same change from the Java API), every `INCREMENTAL` view reading that source type is refreshed as part of the schema change: the rows of the attached records are added and the rows of the detached ones removed. Attaching or detaching a type without records, such as a newly created one, refreshes nothing. The `ALTER TYPE` takes as long as that refresh. If the refresh fails, the schema change still applies and the view is marked `STALE`. `MANUAL` and `EVERY` views are left to their own refresh. *(Since v26.12.1, issue #12031. Before, the view kept the rows of a detached subtype and lacked the rows of an attached type while reporting* `VALID`*, until a manual* `REFRESH MATERIALIZED VIEW`*.)*

Sources that cannot be tracked. `REFRESH INCREMENTAL` is refused with an error, on `CREATE MATERIALIZED VIEW` and on `ALTER MATERIALIZED VIEW ⋯ REFRESH INCREMENTAL`, when the query reads a source that no type's record events report: a `TRAVERSE`, a `MATCH`, a function call in `FROM`, a bucket (`FROM bucket:⋯`), an index, schema metadata (`FROM schema:⋯`), record ids or parameters in `FROM`. Use `REFRESH MANUAL` or `REFRESH EVERY` for such a view.

Values reached by following a link or an edge from a tracked record (for example `address.city` or `out()`) are not tracked, since they cannot be detected from the query text: a change to the linked record does not refresh the view.

This behavior changed in 26.11.1 (issue #10205). Before, only the type of the first `FROM` identifier was tracked, and only its own records, not those of its subtypes: a change to a type read by a subquery, or to a subtype, left the view out of date while its status stayed `VALID`. Existing views loaded from an older database have their source list completed from their query automatically on open, so they start refreshing on those changes and `DROP TYPE` of those types is refused. An existing `INCREMENTAL` view whose query reads one of the untrackable sources above keeps loading; creating a new one, or switching a view to `INCREMENTAL`, is refused.

PERIODIC

A background scheduler thread runs a full refresh at the specified interval after each successful refresh completes. Intervals are specified in seconds, minutes, or hours:

```
REFRESH EVERY 30 SECOND
REFRESH EVERY 5 MINUTE
REFRESH EVERY 1 HOUR
```

The scheduler uses a single daemon thread (`ArcadeDB-MV-Scheduler`) shared across all periodic views. If the database is closed, all scheduled tasks are cancelled automatically.

In a highly available cluster only the leader runs the periodic refresh, *since v26.11.1*: it rewrites the view's records and replicates the rewrite, so the followers receive the refreshed view rather than recomputing it on their own timer. A follower that is promoted to leader starts refreshing at its next interval.

6.17.2. View Status

Each view tracks a `status` field that reflects its current state:

Status	Meaning
`VALID`	Data is up to date with the last refresh
`STALE`	A refresh failed or was interrupted; data may be outdated
`BUILDING`	A refresh is currently in progress
`ERROR`	The last refresh encountered a fatal error

If the database crashes while a view is `BUILDING`, the status is reset to `STALE` on the next startup to signal that the data may be incomplete.

`BUILDING` during a refresh is reported only by the server running that refresh: the schema keeps saving the status the view had before the refresh started, so a schema change made meanwhile (another DDL statement, for example) does not store `BUILDING`. A reopen, and a follower of a highly available cluster, therefore report that status, not `STALE`. The `BUILDING` of a view whose first build has not completed yet is still saved, so a crash during that build reopens the view `STALE`. The same applies to the status of a continuous aggregate. *(Since v26.12.1, issue #12252. Before, such a schema save stored `BUILDING`, and followers reported a view `STALE` that the leader reported `VALID` over the same rows.)*

6.17.3. Querying a Materialized View

Query a materialized view exactly like any other document type:

```
SELECT * FROM ActiveUsers
SELECT name FROM ActiveUsers WHERE name LIKE 'A%'
SELECT count(*) FROM RecentOrders
```

6.17.4. Java API

Creating a view

```
database.transaction(() -> {
    database.getSchema().buildMaterializedView()
        .withName("ActiveUsers")
        .withQuery("SELECT name, email FROM User WHERE active = true")
        .withRefreshMode(MaterializedViewRefreshMode.MANUAL)
        .create();
});
```

Builder options:

Method	Description
`withName(String)`	Name for the view (required)
`withQuery(String)`	Defining SQL `SELECT` query (required)
`withRefreshMode(MaterializedViewRefreshMode)`	`MANUAL`, `INCREMENTAL`, or `PERIODIC` (default: `MANUAL`)
`withTotalBuckets(int)`	Number of buckets for the backing type
`withPageSize(int)`	Page size for the backing type
`withRefreshInterval(long)`	Interval in milliseconds for `PERIODIC` mode. Only `PERIODIC` accepts an interval: `create()` refuses one on a `MANUAL` or `INCREMENTAL` view, embedded and remote alike
`withIgnoreIfExists(boolean)`	When `true`, returns existing view instead of throwing

Querying schema

```
Schema schema = database.getSchema();

// Check existence
boolean exists = schema.existsMaterializedView("ActiveUsers");

// Get a specific view
MaterializedView view = schema.getMaterializedView("ActiveUsers");

// List all views
MaterializedView[] views = schema.getMaterializedViews();
```

Refreshing and dropping

```
// Programmatic refresh
database.getSchema().getMaterializedView("ActiveUsers").refresh();

// Drop via schema
database.getSchema().dropMaterializedView("ActiveUsers");

// Drop via the view itself
database.getSchema().getMaterializedView("ActiveUsers").drop();
```

Changing the refresh mode

```
// Refresh every 5 minutes; pass 0 as the interval for MANUAL or INCREMENTAL
database.getSchema().alterMaterializedView("ActiveUsers", MaterializedViewRefreshMode
.PERIODIC, 300_000);
database.getSchema().alterMaterializedView("ActiveUsers", MaterializedViewRefreshMode
```

```
.INCREMENTAL, 0);
```

`alterMaterializedView()` is the Java form of `ALTER MATERIALIZED VIEW … REFRESH`. *(Since v26.10.2)* it works on a remote database too, where it is sent as that SQL statement, and the embedded and remote calls refuse the same arguments: a `null` mode or a negative interval (`IllegalArgumentException`) and an interval on a `MANUAL` or `INCREMENTAL` view (`SchemaException`). A remote `PERIODIC` interval must be a whole number of seconds, as in SQL.

Inspecting a view

```
MaterializedView view = database.getSchema().getMaterializedView("HourlySummary");

view.getName();             // "HourlySummary"
view.getQuery();            // the defining SQL query
view.getRefreshMode();      // MaterializedViewRefreshMode.PERIODIC
view.getStatus();           // "VALID", "STALE", "BUILDING", or "ERROR"
view.getLastRefreshTime();  // epoch millis of last successful refresh
view.isSimpleQuery();       // true if eligible for per-record optimization
view.getSourceTypeNames();  // every type the query names, at any depth
view.getBackingType();      // the underlying DocumentType
```

6.17.5. Behavior and Constraints

- **Backing type protection**: You cannot `DROP TYPE` on a type that backs a materialized view. Drop the materialized view first.
- **The backing type is read-only**: only the refresh of the view writes its rows. `INSERT`, `UPDATE` and `DELETE` on the backing type, `TRUNCATE TYPE` on it (directly, or reaching it with `POLYMORPHIC` from a supertype), `TRUNCATE BUCKET` on one of its buckets and `ALTER TYPE <view> BUCKET -<bucket>` are refused, from SQL, the Java API and every protocol, with a message naming `REFRESH MATERIALIZED VIEW <view>` and `DROP MATERIALIZED VIEW <view>`. Change the source data instead, and refresh. A refused `TRUNCATE` deletes nothing. Reading the backing type and indexing it (`CREATE INDEX ON <view> …`) are unaffected. *(Since v26.11.1, issue #11020. Before, a `TRUNCATE TYPE` of the backing type emptied the view while it kept reporting `VALID`, and a row inserted into it was answered beside the rows of the query until the next refresh.)*
- **The backing type cannot be extended**: `CREATE … TYPE <type> EXTENDS <view>` and `ALTER TYPE <type> SUPERTYPE <view>` are refused, as is a type below another one that extends the backing type, since `SELECT FROM <view>` also returns the rows of its subtypes, which no refresh computed. The backing type itself can still be given a supertype. A database where such a link already exists still opens, logs a warning, and the link can be removed with `ALTER TYPE <type> SUPERTYPE -<view>`. _(Since v26.11.1, issue #11204.)_ In such a database a `DROP TYPE` that would link a sub type directly under the backing type, or under a type below it (dropping `Mid` out of `+Deep -> Mid -> <view>`), is refused before anything is changed, naming the link to take out first (`ALTER TYPE Deep SUPERTYPE -Mid`); after that the drop goes through. The same holds for the backing type of a continuous aggregate. *(Since v26.12.1, issue #11272. Before, the drop linked the sub type under the backing type.)*
- **Source type protection**: You cannot `DROP TYPE` on a type the defining query reads (any type

listed in `sourceTypes`). Drop the materialized view first.

- **Renaming a type**: `ALTER TYPE <source> NAME <new>` on a source type of a view carries the view with it: the source type name the view stores and the `FROM` of its defining query are rewritten to the new name, and the change is saved with the schema. A continuous aggregate follows the rename of its source type the same way, and so does a trigger bound to the type. The rename is refused, before anything moves, when the type is the **backing** type of a view (a view is named after its backing type, so it cannot be renamed on its own) or when the query names the type in a form that cannot be re-pointed with certainty (for example a subquery that reads the same type again): drop the view, rename the type and create the view again. A view defined through an alias of the type keeps working without being rewritten, since the alias survives the rename. *(Since v26.11.1. Before, the rename left the view naming a type that no longer existed: its refresh failed, and `DROP TYPE` on the renamed type was let through. A database that already holds a view in that state is protected from the drop: `DROP TYPE` is refused, naming the view that is out of date, until that view is dropped.)*
- **Name uniqueness**: The view name must not match any existing type or materialized view.
- **Source type validation**: All types referenced by the query, including those in subqueries, must exist when the view is created.
- **Persistence**: View definitions are stored in `schema.json` under a `"materializedViews"` key and survive database restarts. Listener registration for `INCREMENTAL` views and scheduler tasks for `PERIODIC` views are re-established on startup.
- **Transaction safety**: The initial full refresh and all subsequent refreshes run inside their own transactions.
- **Query result columns**: Only non-internal properties (those not starting with `@`) are copied into the backing type during refresh.
- **No schema on backing type**: The backing document type is schema-less; property types are not enforced.

6.17.6. Error Handling

- If a post-commit refresh fails (`INCREMENTAL` mode), the view is marked `STALE` and a `WARNING` is logged. The source transaction is unaffected.
- If a periodic refresh fails, the view is marked `ERROR` and a `SEVERE` log entry is written. The scheduler continues running and will retry on the next interval.
- Callback errors in the transaction callback system are logged at `WARNING` level and do not affect the triggering transaction or other callbacks.

6.17.7. Limitations

- Per-record incremental refresh (tracking `_sourceRID` to update individual view rows) is a planned future optimization. Currently, all refresh operations perform a full truncate-and-reload.
- No support for cross-database queries in the defining query.
- Server replication: materialized view data lives in the local backing type and is replicated like any other document type in an HA cluster, but refresh triggering is local to the node that

executes the write.

6.17.8. Example: Sales Dashboard

```
-- Source type
CREATE DOCUMENT TYPE Sale;

-- A periodic summary refreshed every minute
CREATE MATERIALIZED VIEW SalesByProduct
  AS SELECT product, sum(amount) AS total, count(*) AS count
     FROM Sale
     GROUP BY product
  REFRESH EVERY 1 MINUTE;

-- An incremental view of recent activity (simple query)
CREATE MATERIALIZED VIEW RecentSales
  AS SELECT product, amount, date
     FROM Sale
     WHERE date >= '2026-01-01'
  REFRESH INCREMENTAL;

-- Query the views
SELECT * FROM SalesByProduct ORDER BY total DESC;
SELECT product, amount FROM RecentSales WHERE amount > 1000;

-- Manual refresh after a bulk import
REFRESH MATERIALIZED VIEW SalesByProduct;

-- Teardown
DROP MATERIALIZED VIEW SalesByProduct;
DROP MATERIALIZED VIEW RecentSales;
```

For more information, see:

- `CREATE MATERIALIZED VIEW`
- `DROP MATERIALIZED VIEW`
- `REFRESH MATERIALIZED VIEW`
- `ALTER MATERIALIZED VIEW`

6.18. Vector Embeddings

This guide covers practical decisions for working with vector embeddings in ArcadeDB: choosing dimensions, creating indexes, tuning parameters, and combining vector search with other query types.

6.18.1. Choosing an Embedding Model

Your embedding model determines the `dimensions` parameter for the index:

Model	Dimensions	Notes
OpenAI `text-embedding-3-small`	1536	General purpose, high quality
OpenAI `text-embedding-3-large`	3072	Highest quality, largest memory footprint
Sentence Transformers `all-MiniLM-L6-v2`	384	Fast, open source, good quality
Sentence Transformers `all-mpnet-base-v2`	768	Better quality, slower
Cohere `embed-english-v3.0`	1024	Good balance of quality and size
CLIP (image + text)	512	Multi-modal image/text

Start with 384 dimensions (MiniLM) for prototyping. Move to 768+ for production quality. Use quantization to manage memory at higher dimensions.

6.18.2. Creating a Vector Index

Recommended index creation with INT8 quantization:

```
CREATE VERTEX TYPE Document
CREATE PROPERTY Document.content STRING
CREATE PROPERTY Document.embedding LIST OF FLOAT

CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  dimensions: 384,
  similarity: 'COSINE',
  quantization: 'INT8'
}
```

INT8 quantization is recommended for all production workloads. It provides 4x lower memory usage with negligible accuracy loss, and it reads vectors from index pages instead of documents, which usually lowers search latency; how much depends on the data and the recall target (see concepts/vector-search.pdf). Only omit quantization for very small datasets (< 10K vectors) where maximum precision matters.

Production-ready index with additional tuning:

```
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  dimensions: 384,
  similarity: 'COSINE',
  quantization: 'INT8',
  maxConnections: 16,
  beamWidth: 100
```

```
}
```

A `CREATE INDEX … LSM_VECTOR`, a Cypher `CREATE VECTOR INDEX` or a `REBUILD INDEX` over existing records leaves out a record whose value the index cannot take: something that is not a vector (a scalar, a string), a vector with a different number of dimensions, or a `NaN`/`Infinity` component. This matches Neo4j, which does not index such a node either. Since v26.11.1 the build says so: one `WARNING` counts them and names the first ten, for example `Vector index 'Doc[embedding]': 2 records are not indexed, holding a value that is not a vector of 3 dimensions (first: #1:1 (…), #1:3 (…))`, the SQL `CREATE INDEX` and `REBUILD INDEX` results carry the count as `skipped`, and the index statistics as `lastBuildSkippedRecords` (the last build since the database was opened: the counter is not persisted). Fix or remove those values: a record updated with a valid vector is indexed by the write, and `REBUILD INDEX` picks up the others. A record with no value for the property is not counted. Before, the build logged one `SEVERE` line per record and reported success with nothing else to tell the caller that the index lacked them.

6.18.3. Choosing a Similarity Function

Function	Choose When	Avoid When
COSINE	Using text embedding models (most common). Vectors may have varying magnitudes.	Vectors represent absolute quantities (distances, counts).
DOT_PRODUCT	Vectors are already L2-normalized. You need maximum query speed.	Vectors are not normalized (results will be incorrect).
EUCLIDEAN	Working with spatial data, sensor readings, or continuous measurements.	Comparing text embeddings of different lengths.

A vector the similarity function cannot score is kept on its record but never returned by a vector search. Under `COSINE` that is an all-zero vector or one whose squared norm is below 2^{-63} or above 2^{63} (a nearly zero or a huge vector, typically an unnormalized or corrupt embedding); under `DOT_PRODUCT` it is an all-zero vector; `EUCLIDEAN` scores every vector. A `COSINE` query vector outside that range is refused with an error. Since v26.12.1 this also holds for a vector written after the index graph was built, for a search inside the transaction that wrote it and for an index with `BINARY` quantization: before, such a vector could be returned first, at distance `-Infinity`, or make the search fail. *(Issues #13118, #13143.)*

6.18.4. Quantization Trade-offs

Use INT8 quantization for most use cases. It provides 4x memory savings with minimal accuracy loss and usually lower search latency (see concepts/vector-search.pdf):

- **< 10K vectors**: `NONE` is fine, but `INT8` works well too

- **10K - 1M vectors**: Use `INT8` (4x memory savings, < 2% accuracy loss) — **recommended**
- **> 1M vectors**: Use `INT8` for general use, or `PRODUCT` for zero-disk-I/O graph construction on very large datasets. A `PRODUCT` search walks on the in-memory PQ codes and re-ranks `rerankOversample` (default 10) candidates per neighbor on the stored vectors, so it still reads the stored vectors of those candidates and returns exact distances
- **Extreme compression**: Use `BINARY` for first-pass filtering, then rerank with full vectors

```
-- INT8: recommended for most workloads
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  dimensions: 768,
  similarity: 'COSINE',
  quantization: 'INT8'
}

-- PRODUCT: for very large datasets, enables in-memory graph build
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  dimensions: 1024,
  similarity: 'COSINE',
  quantization: 'PRODUCT'
}
```

6.18.5. INT8 Pre-Quantized Ingest

See Vector Encoding for the underlying concept.

When your embedding provider already emits signed-int8 vectors (Cohere `int8` endpoints, OpenAI `text-embedding-3-large` reduced precision, Sentence Transformers with int8 quantization), use `encoding: 'INT8'` to keep the bytes byte-shaped end-to-end:

```
CREATE PROPERTY Doc.embedding BINARY;
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  "dimensions": 1024,
  "similarity": "COSINE",
  "encoding": "INT8"
};
```

What this saves:

- HTTP payload: 4x smaller (1 byte/dim vs 4 bytes/dim) when sending vectors with the typed-marker convention — see HTTP wire convention.
- Document bucket storage: 4x smaller, since the property column is `BINARY` instead of `ARRAY_OF_FLOATS`.
- No client-side `int8 → float32 → server` round trip; the precision the provider already discarded does not get padded back out for the wire.

What this does **not** change today:

- HNSW graph and search internally still run on `float32` (JVector 4.0.0-rc.8 contract). The engine dequantizes once on the read path. Native int8 HNSW is tracked upstream at datastax/jvector#665.
- `encoding` is independent of `quantization`. Combining `encoding: 'INT8'` with `quantization: 'INT8'` is rejected at index creation — pick one, not both.

When **not** to use INT8 encoding:

- Your provider emits `float32` (or `float64`) and you do not have an int8 quantizer client-side. The default `FLOAT32` encoding skips the dequantize hop on every search and keeps full precision in the documents.
- You want the index-internal compression benefit (search-time memory footprint). That is `quantization: 'INT8'`, not `encoding: 'INT8'`. They are orthogonal.

6.18.6. Tuning for Recall vs Speed

Adjust `maxConnections` and `beamWidth` based on your priorities:

Profile	maxConnections	beamWidth	Trade-off
Default	32	100	Balanced for most workloads; matches hnswlib `M=16` density
High recall	48	200	Better accuracy, slower builds, more memory
Fast indexing	24	80	Faster builds, slightly lower recall
Memory constrained	16	60	Minimal memory footprint

`maxConnections` is the Vamana per-layer graph degree; unlike hnswlib's `M` it is not doubled at the base layer, so to match an hnswlib `M` set `maxConnections = 2 * M`.

For datasets over 100K vectors or with 1024+ dimensions, enable hierarchical mode:

```
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  dimensions: 1536,
  similarity: 'COSINE',
  quantization: 'INT8',
  addHierarchy: true,
  maxConnections: 32,
  beamWidth: 200
}
```

6.18.7. Tuning efSearch

The `efSearch` parameter controls how many candidates the search explores at query time. By default, ArcadeDB uses an adaptive strategy that works well for most workloads. You only need to

tune efSearch if you have specific recall or latency requirements.

Profile	efSearch	Trade-off
Adaptive (default)	auto	Two-pass: fast first pass (`2xk`), wider retry (`10xk`) if needed
High recall	200-500	Consistent high accuracy, higher latency
Low latency	20-50	Fast responses, lower recall on hard queries

You can override efSearch per-query without changing the index:

```
-- High recall for a critical search
SELECT expand(vector.neighbors('Doc[embedding]', :queryVector, 10, 500))

-- Low latency for autocomplete/typeahead
SELECT expand(vector.neighbors('Doc[embedding]', :queryVector, 5, 30))
```

Or set a default on the index:

```
CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {
  dimensions: 768,
  similarity: 'COSINE',
  quantization: 'INT8',
  efSearch: 200
}
```

6.18.8. Returning Only RIDs and Distances

A quantized index (`INT8`, `BINARY`, `PRODUCT`) walks its graph on the quantized vectors and reads no document. Since v26.11.1 `vector.neighbors()` does not read the record of a neighbour either until a property of it is asked, so a query that projects only `@rid`, `@type` and `distance` keeps the documents, and the full-precision embeddings they hold, out of the page cache:

```
-- Reads no document: the application keeps the entity fields elsewhere
SELECT @rid, distance FROM (SELECT expand(vector.neighbors('Doc[embedding]',
:queryVector, 10)))

-- Reads the 10 records of the answer, and nothing else
SELECT @rid, title, distance FROM (SELECT expand(vector.neighbors('Doc[embedding]',
:queryVector, 10)))
```

The record is still read as the row is made, as before, when a `groupBy` is given, when a read listener or trigger is registered on the database or on the type, and when the calling transaction has written to the index.

Behavior change in v26.11.1. A row used to be checked against its bucket when it

was made, and a neighbour whose record was gone was skipped and replaced. A row is now checked against the index, which a delete updates in the same commit, so a committed delete (a concurrent one included) still costs no row. What the index cannot see is a record removed from its bucket behind its back, which only a damaged database has (a crash, a partial restore): such a row is returned with its `@rid` and no property, instead of being skipped, and it takes one of the `k` places, so the answer holds one real neighbour fewer. The first read of such a row logs a `WARNING` (at most once a minute) naming the type. `REBUILD INDEX` on the vector index removes those entries. *(Issue #11202.)*

6.18.9. Multi-Modal Embeddings

Store multiple embeddings per record for different search modalities:

```
CREATE VERTEX TYPE Product
CREATE PROPERTY Product.imageEmbedding ARRAY_OF_FLOATS
CREATE PROPERTY Product.textEmbedding  ARRAY_OF_FLOATS

CREATE INDEX ON Product (imageEmbedding) LSM_VECTOR METADATA {dimensions: 512,
similarity: 'COSINE'}
CREATE INDEX ON Product (textEmbedding)  LSM_VECTOR METADATA {dimensions: 768,
similarity: 'COSINE'}
```

Query each index independently:

```
-- Search by image similarity
SELECT name, distance FROM (
  SELECT expand(vector.neighbors('Product[imageEmbedding]', :imageVector, 10))
)

-- Search by text similarity
SELECT name, distance FROM (
  SELECT expand(vector.neighbors('Product[textEmbedding]', :textVector, 10))
)
```

6.18.10. Searching a Type Hierarchy

The type in the index spec decides which records are searched. A parent spec covers the parent and all its sub-types, like `SELECT FROM Parent`. A sub-type spec covers only that sub-type and its own sub-types:

```
CREATE VERTEX TYPE Embedding
CREATE PROPERTY Embedding.vector ARRAY_OF_FLOATS
CREATE INDEX ON Embedding (vector) LSM_VECTOR METADATA { dimensions: 768, similarity:
'COSINE' }
CREATE VERTEX TYPE EmbeddingImage EXTENDS Embedding
CREATE VERTEX TYPE EmbeddingText  EXTENDS Embedding
```

```
-- images and texts (and any record of Embedding itself)
SELECT expand(vector.neighbors('Embedding[vector]', :q, 10))

-- images only
SELECT expand(vector.neighbors('EmbeddingImage[vector]', :q, 10))
```

The same rule applies to `vector.sparseNeighbors`, the Cypher procedure `db.index.vector.queryNodes` and the Java Select API `select().fromType(..).nearestTo(..)`.

6.18.11. Hybrid Search: Dense + Sparse + Full-Text

Combine dense vector similarity with sparse retrieval and/or keyword matching in a single server-side query. `vector.fuse` accepts any number of ranked sub-pipelines plus a fusion strategy (`RRF`, `DBSF`, `LINEAR`) and returns one ranked top-K:

```
-- Schema: dense + sparse properties + indexes (sparse needs LSM_SPARSE_VECTOR).
CREATE PROPERTY Document.dense   ARRAY_OF_FLOATS;
CREATE PROPERTY Document.tokens  ARRAY_OF_INTEGERS;
CREATE PROPERTY Document.weights ARRAY_OF_FLOATS;

CREATE INDEX ON Document (dense) LSM_VECTOR
  METADATA { dimensions: 384, similarity: 'COSINE' }

CREATE INDEX ON Document (tokens, weights) LSM_SPARSE_VECTOR
  METADATA { dimensions: 30000, modifier: 'IDF' }

-- Hybrid retrieval in one statement.
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Document[dense]', :denseVec, 50),
    `vector.sparseNeighbors`('Document[tokens,weights]', :qIdx, :qVal, 50),
    { fusion: 'RRF', groupBy: 'source_file', groupSize: 1 }
)) LIMIT 10
```

- `vector.neighbors` exposes `distance` (lower = better); `vector.fuse` auto-flips it so dense and sparse sources compose without manual rescaling.
- `groupBy` + `groupSize` collapse same-source duplicates server-side. Drop the option to return chunk-level results.
- Pre-fusion grouping is also possible by attaching `{ groupBy: 'source_file', groupSize: 1 }` to each individual source.

To include full-text alongside dense/sparse, add a third source built from `SEARCH_INDEX`:

```
SELECT expand(`vector.fuse`(
    `vector.neighbors`('Document[dense]', :denseVec, 100),
    `vector.sparseNeighbors`('Document[tokens,weights]', :qIdx, :qVal, 100),
    (SELECT @rid, $score FROM Document
```

```
     WHERE SEARCH_INDEX('Document[content]', 'machine learning') = true),
    { fusion: 'RRF' }
)) LIMIT 10
```

Pick the strategy that matches your scoring shape:

- `RRF` — rank-only, indifferent to score scales. Default, safest with mixed source types.
- `DBSF` — mean +/- 3sigma normalisation per source then weighted sum. Use when scores are roughly Gaussian on each side.
- `LINEAR` — per-source min-max normalisation then weighted sum. Use with already-tuned offline weights.

For the legacy two-query workaround (still supported via `vector.rrfScore` and `vector.hybridScore` on already-computed scores), see the SQL Vector Functions reference.

6.18.12. Batch Ingestion

For bulk loading vectors, batch your inserts within transactions:

```
BEGIN

CREATE VERTEX Document SET content = 'First document',  embedding = [0.1, 0.2, ...]
CREATE VERTEX Document SET content = 'Second document', embedding = [0.3, 0.4, ...]
-- ... more inserts ...

COMMIT
```

For a large load through SQL or the record API, increase `mutationsBeforeRebuild` and `inactivityRebuildTimeoutMs` so the graph is not rebuilt part-way (a pause longer than `inactivityRebuildTimeoutMs`, 15 seconds by default, can trigger a rebuild whatever `mutationsBeforeRebuild` is), then run `REBUILD INDEX` once the load is finished, for example `REBUILD INDEX `Document[embedding]``. `GraphBatch` loads need neither.

A bulk load through `GraphBatch` - which is what `GraphImporter`, the `arcadedb-integration` importers and the HTTP batch endpoint use - needs none of that tuning. ArcadeDB holds the automatic graph rebuild off for the whole load and runs it once at the end. Before that, the rebuild was triggered by the index going quiet, which during a load means the loader paused for an index compaction or a garbage collection rather than that it had finished: a 4.2M-vertex load rebuilt the graph four times, each over a partial dataset and each made obsolete by the rest of the load, and had not completed after six and a half hours. The same release also bounds the memory the waiting vectors occupy (see `deltaCacheSize` under Settings) - together these are what let a load with a vector index finish in the same time as one without.

When vectors are inserted below the rebuild threshold, an inactivity timer ensures the graph is still rebuilt after a period of no new mutations (default: 15 seconds). On a small graph (under 1,000 vectors) this rebuild is cheap and fires for any number of pending mutations, so buffered vectors never sit in the brute-force delta buffer for long during low-volume ingestion. On a larger graph a rebuild re-indexes the whole graph, so the timer only fires once pending mutations reach at least 10% of the effective rebuild threshold - a single stray insert into a large, otherwise-settled index no longer costs a full graph rebuild on the next quiet period. "How large" is measured against the vectors the index actually holds, not against the part of the graph the current session has loaded: the graph loads lazily on the first query, so a process that reopens a database, writes and then goes idle without ever querying it is gated exactly like one that queried first. Configure the timer via `inactivityRebuildTimeoutMs` (per-index metadata or `arcadedb.vectorIndex.inactivityRebuildTimeoutMs` globally); set to 0 to disable it.

A search waits for the same threshold before it rebuilds a small graph (under 1,000 vectors). Since v26.12.1 a write below `mutationsBeforeRebuild` no longer makes the next search rebuild the graph first: the new vectors are answered exactly from the delta buffer until the threshold is reached, and the next search after that rebuilds the graph as before. Previously every search after any write rebuilt a small graph, and the other searches of that index queued behind the rebuild. The 1,000 is the size of the graph of one bucket and a type has one graph per bucket, so a type with 16 buckets and 15,000 vectors has 16 small graphs, which under a steady trickle of writes were rebuilt over and over. An index that has no graph yet is still built by its first search.

The same thresholds also decide what happens on the **first search after reopening a database**. Vectors written since the graph was last persisted are read back into the delta buffer and answered from there, and the rebuild that folds them into the graph is scheduled only once they reach the threshold, or once the delta scan they cause grows expensive (see `maxDeltaScanRatio`). Previously that first search always started a full rebuild however small the gap, so a process that opened a database, searched once and exited re-indexed the whole graph for a single buffered vector, waited for that rebuild on `close()`, and found the same vector waiting again on the next open - the rebuild having been cancelled before it could be persisted.

6.18.13. Searching Vectors Not Yet in the Graph

Between rebuilds, everything written since the last one lives in an in-memory delta buffer rather than in the HNSW graph. Those vectors are still searchable: every query scores the buffer and merges the result into the graph's own, in distance order, so a search never returns a corpus older than the last write. The cost is linear in the size of the buffer, which is what `mutationsBeforeRebuild`, `rebuildGraphRatio` and the inactivity timer between them decide.

A graph build also occasionally leaves a vector with no incoming edges, which no search can reach at any `efSearch` however close it is to the query. Those are served from the delta buffer too, and the

list of them is stored next to the graph, so they stay findable after a restart. `unreachableGraphNodes` reports how many the current graph has, readable straight after a reopen and before the first query:

```
-- Normally 0, or a handful in a large index. A high value means the corpus holds many
near-identical vectors,
-- which a build cannot link into a navigable graph, and every query pays a scan for
them
SELECT unreachableGraphNodes, deltaVectorsCount FROM (SELECT expand(stats) FROM
schema:indexes WHERE name = 'Document[embedding]')
```

A grouped search (one using `groupBy`) scores the delta buffer and merges it in the same way.

The `groupedSearchesMergingDelta` counter reports how many grouped searches actually took rows out of the buffer:

```
-- Rising in step with your grouped query rate means the graph is persistently behind
the write rate,
-- and every grouped query is paying a linear scan of the buffer; deltaVectorsCount is
how long that scan is
SELECT groupedSearchesMergingDelta, deltaVectorsCount FROM (SELECT expand(stats) FROM
schema:indexes WHERE name = 'Document[embedding]')
```

A high value is not itself a fault - it is the index doing what it should under ingestion. It is the signal to shorten the scan by lowering `mutationsBeforeRebuild` or `rebuildGraphRatio`, if query latency matters more than the CPU the extra rebuilds cost.

6.18.14. Rebuild Memory and Deferred Rebuilds

A graph rebuild is the most memory-hungry thing a vector index does, and an *online* rebuild - one triggered by the mutation threshold or the inactivity timer while the index keeps serving queries - is the most expensive kind. The graph being replaced stays resident so searches keep working, and the new build pays for its own working set on top of it (the build cache, the graph under construction, the ordinal map). Measured on a 50,000 x 128 corpus built and rebuilt inside one session, that is roughly 1.7x the peak of building the same index from nothing. It is much less after a reopen, where the graph being replaced is served from pages rather than from the heap: the gate measures what that graph actually costs instead of assuming a second copy of it in memory.

Two behaviours follow from that:

- **The auto-sized search cache budgets against available heap, not total heap.** `searchCacheMaxHeapPercent` is a share of the heap actually free at the time, so a rebuild holding the old graph resident asks for a smaller cache instead of the same one. Since v26.12.1 that share is one budget for all the search caches of the database, not one per index: with a vector index per bucket, the first caches to grow take the room, and an index that finds none left has no cache and reads its vectors from the pages.
- **A rebuild that will not fit is deferred instead of attempted.** Before starting, the estimated

peak is compared against `arcadedb.vectorIndex.rebuildMaxHeapPercent` (default 90) of the available heap. If it does not fit, the cycle is skipped rather than run into an `OutOfMemoryError`. Nothing is lost: the pending vectors remain fully searchable through the in-memory delta buffer, so the cost is a longer brute-force scan per query until a later rebuild succeeds. Since v26.10.2, before deferring, the rebuild first tries to keep the old graph on disk instead of in memory (the persisted copy is read from pages while the new graph builds), which usually makes it fit. Searches are somewhat slower until the new graph is published; this is counted as `residentGraphDemotions` in the index statistics, and a deferral warning is logged only once until a rebuild gets through.

Both sides of that comparison are more accurate, and a large index in a tight heap that used to log deferrals indefinitely may now rebuild:

- **What keeping the old graph resident costs is measured, not assumed.** After a reopen the graph being replaced lives in pages, so it retains a fraction of what a freshly built one does. Charging it as a second in-memory graph could refuse a rebuild that fitted comfortably - and one no larger than the rebuild the close path runs without asking.
- **The page cache counts as reclaimable, because it is.** Available heap is measured after the last collection, and cached pages count as used even though the engine can drop any of them and read them back from disk. When a rebuild fits only by giving some of that up, exactly the shortfall is evicted first - oldest pages, across every open database, since the cache is shared - and if the cache cannot actually hand those bytes over the rebuild is still deferred rather than run on heap it was not given. Expect a few slower queries afterwards while the evicted pages are read back.

The graph-build cache is sized differently from the search cache, and the difference matters when you are planning a large ingest:

- `graphBuildCacheMaxHeapPercent` **is a share of** `-Xmx`**, capped at 90% of the heap currently available.** Taking the percentage of free heap made the same corpus build very differently depending on how the data got in. A server that has just ingested the corpus over HTTP holds it in its own heap, so the free-heap reading is small and the default paid for a fraction of the corpus; an embedded build of the identical data, held outside the JVM until the build, saw almost the whole heap free and cached everything. On a 10M x 96-dim corpus with the same `-Xmx` that was a 3.4x difference in build time from the same default, and two identical runs could differ 4x between themselves depending on when the JVM had last collected. Taking the percentage of the ceiling makes the choice a property of your configuration rather than of the moment; the available-heap cap is what still shrinks the cache when an online rebuild is genuinely holding the old graph.
- **The percentage now applies to inline-quantized (**`INT8`**,** `BINARY`**) indexes too.** It previously did nothing for them: their build cache was pinned at 100,000 vectors whatever the setting said, and only an explicit `graphBuildCacheSize` moved it. Note the consequence for a tight heap - an `INT8` index that used to hold ~45 MB of build cache at 96 dimensions will now use up to its share of `-Xmx`, because the cache holds decoded `float` vectors regardless of how they are stored (from v26.11.1 this holds only while the cache covers the corpus: past that the same budget is spent on quantized rows, see below). Set `graphBuildCacheMaxHeapPercent` lower, or pin `graphBuildCacheSize`, if that share is not available to you.

- **The chosen capacity is logged next to the corpus size**, as `cache enabled: size=3674697 of 9990000`, so a cache that is too small to cover the corpus is visible in one line at `INFO`.

Since v26.11.1 an `INT8` or `BINARY` index whose corpus does not fit that cache is built from the quantized rows instead of from decoded `float` vectors:

- **The same memory covers four times as many `INT8` vectors (thirty times as many `BINARY` ones).** A 768-dimension vector is 3,136 bytes as a `float` cache entry and 776 bytes as the row the index pages store, so the budget `graphBuildCacheMaxHeapPercent` (or `graphBuildCacheSize`) pays for is spent on rows. A vector is decoded only at the moment a distance needs it, into a scratch the build thread owns, so a read allocates nothing.
- **Why it matters on a large corpus.** The decoded cache used to re-read every vector it did not hold from the index pages, decode it into a new array and publish it into the cache, evicting an older one. On a 13.8M x 768 `INT8` corpus in a 24 GB heap that churn kept the heap between 17 and 24 GB and slowed the build from about 2,000 vectors per second to one vector every five seconds. On a 1M x 768 corpus in a 6 GB heap, with default settings, the build went from 444 s (8,461 collector pauses, 751 of them full) to 196 s (157 pauses, none full).
- **A corpus that fits the decoded cache is built exactly as before.** The quantized path only starts when the float cache cannot hold every vector, and the build log says which one is in use: `all kept as INT8 rows in memory` when the budget covers the corpus, or `N of them kept as INT8 rows in memory: …; the other M are read from the index pages` together with the percentage to set to keep all of them.
- **`INT8` vectors are decoded slightly differently.** The reconstruction `(q + 128) * ((max - min) / 255) + min` replaces `((q + 128) / 255) * (max - min) + min`: the same values to within 3e-5 of the vector's range, which is under 1% of one quantization step, and about three times faster. It applies to searches as well, so scores can differ in the last bits from earlier versions.

A deferral is deliberately visible. It logs a warning naming what to change, and increments the `rebuildsDeferredForMemory` counter in the index statistics:

```
-- Non-zero and climbing means rebuilds keep being declined for lack of heap:
-- the graph is going stale and every query is paying a longer delta scan for it
SELECT rebuildsDeferredForMemory FROM (SELECT expand(stats) FROM schema:indexes WHERE
name = 'Document[embedding]')
```

If it keeps climbing, give the JVM more heap, lower `arcadedb.vectorIndex.graphBuildCacheMaxHeapPercent` so the build asks for less, or split the index. Setting `arcadedb.vectorIndex.rebuildMaxHeapPercent` to `0` disables the check and restores the previous attempt-regardless behaviour.

Because a deferral does not consume the mutations that triggered it, the trigger would otherwise fire again on the very next query. `arcadedb.vectorIndex.rebuildDeferralCooldownMs` (default 30 seconds) is the minimum gap before another attempt; a rebuild that completes clears it immediately.

Only online rebuilds are subject to this check. A first build, the rebuild performed when a database closes, an explicit `REBUILD INDEX` and `COMPACT INDEX` are never

> declined - nothing would retry them, so declining one would mean it never happens at all.

If you create the index *before* inserting data (e.g., during schema setup), set `buildGraphNow: false` to skip the initial (empty) graph build. The graph will be built lazily on the first search:

```
-- Schema setup phase: defer graph build since no data exists yet
CREATE INDEX ON Document (embedding) LSM_VECTOR METADATA {
  dimensions: 384,
  similarity: 'COSINE',
  quantization: 'INT8',
  buildGraphNow: false
}

-- Bulk load data...
-- Graph is built automatically on first vector.neighbors() query
```

If you create the index *after* data is already loaded, leave `buildGraphNow` at its default (`true`) so the index is immediately ready to query.

6.18.15. Global Configuration

Set database-wide defaults for vector index parameters:

```
ALTER DATABASE `arcadedb.vectorIndex.locationCacheSize` 100000
ALTER DATABASE `arcadedb.vectorIndex.graphBuildCacheSize` 10000
ALTER DATABASE `arcadedb.vectorIndex.mutationsBeforeRebuild` 100
ALTER DATABASE `arcadedb.vectorIndex.inactivityRebuildTimeoutMs` 15000
```

Per-index metadata overrides these global settings.

6.18.16. Further Reading

- Vector Search Concepts — Architecture and algorithm details
- Vector Search Tutorial — Step-by-step hands-on guide
- Java Vector API — Programmatic index management
- SQL Vector Functions — Complete function reference

Chapter 7. Import & Migration

7.1. Importer

ArcadeDB provides some basic ETL capabilities for automatically importing datasets in any of the following formats:

- OrientDB database export
- Neo4j database export
- GraphML database export
- GraphSON database export
- Generic XML files
- Generic JSON files
- Generic JSONL files
- Generic CSV files
- Generic RDF files

From file of types:

- Plain text
- Compressed with ZIP (the first entry in the archive is read, unless you select one - see below)
- Compressed with GZip

Located on:

- **local** file system (just provide the path or use `file://` in the URL)
- **remote**, by specifying `http://` or `https://` in the URL
- **classpath**, by using `classpath://` as a prefix

A remote source that stops sending data without closing the connection makes the import wait rather than fail immediately - there is no read timeout yet.

To import one specific entry out of a ZIP archive that contains several, append `:::` and the entry name to the URL:

```
IMPORT DATABASE file://data.zip:::nodes.csv
```

Without `:::`, the first entry in the archive is imported.

The easiest way is to use the console and the SQL `IMPORT DATABASE` command. You can also use directly the Java API located in `com.arcadedb.integration.importer.Importer`.

To restore an export into a database you reach **over the network**, send `IMPORT`

DATABASE as a command like any other: the server runs the import against its own local database, so the file path is resolved on the server. The Java Importer class loads into an embedded database only.

```
IMPORT DATABASE file:///path/on/the/server/export.jsonl.tgz
```

To start importing it's super easy as providing the URL where the source file to import is located. URLs can be local paths (use file://) or from the Internet by using http:// and https://.

For security reasons, importing from an http://https:// URL refuses by default to reach a local-file, loopback, link-local, or private-network address, including a redirect that lands on one. This protects the server from being made to fetch from internal-only hosts (e.g. a cloud metadata endpoint) via a crafted or hijacked import URL. Running IMPORT DATABASE directly (from the console or the SQL API) is gated by arcadedb.server.security.importBlockLocalNetworks, which defaults to true; set it to false on the server if you need to import from such an address (for example, an internal artifact server). The import database <dbname> <url> server command is gated instead by arcadedb.server.restoreImportAllowLocalUrls - the same setting documented for restoring a database - which must be set to true for the equivalent override.

the import: startup command of arcadedb.server.defaultDatabases obeys arcadedb.server.security.importBlockLocalNetworks as set on that server.

Example of loading the Freebase RDF dataset:

```
> CREATE DATABASE FreeBase
{FreeBase}> IMPORT DATABASE http://commondatastorage.googleapis.com/freebase-public
/rdf/freebase-rdf-latest.gz
```

```
Analyzing url: http://commondatastorage.googleapis.com/freebase-public/rdf/freebase-
rdf-latest.gz... [SourceDiscovery]
Recognized format RDF (limitBytes=9.54MB limitEntries=0) [SourceDiscovery]
Creating type 'Node' of type VERTEX [Importer]
Creating type 'Relationship' of type EDGE [Importer]
Parsed 144951 (28990/sec) - 0 documents (0/sec) - 143055 vertices (28611/sec) - 144951
edges (28990/sec) [Importer]
Parsed 362000 (54256/sec) - 0 documents (0/sec) - 164118 vertices (5260/sec) - 362000
edges (54256/sec) [Importer]
...
```

An RDF source is recognised from the shape of its first statement - subject predicate object with an optional closing . - so quoted literal objects, language tags (@en), typed literals (^^<datatype>), blank nodes (_:b1) and Windows (CRLF) line endings are all read as RDF, and the character separating the terms is used as the field delimiter. If a file still cannot be recognised, set the separator explicitly

with the `delimiter` setting.

Each statement becomes a `Relationship` edge, with the predicate in its `label` property, between two `Node` vertices keyed by the `typeIdProperty` (`id` by default). If the database has no `Node` type, the import creates it with a unique index on that property. An IRI is keyed by its text without the angle brackets and a blank node by its label (`_:b1`). A literal object is keyed by its canonical N-Triples form, quotes included (`"hello world"`, `"bonjour"@fr`, `"42"^^http://www.w3.org/2001/XMLSchema#integer`), so it never shares a vertex with an IRI of the same text, and `"a"` and `"a"^^http://www.w3.org/2001/XMLSchema#string` are the same literal. The vertex of a literal also gets `value`, its text with escapes such as `é` and `\n` decoded and without quotes, plus `lang` for a language-tagged literal and `datatype` for a typed one. A malformed statement, or a term longer than `maxPropertySize`, is a row error that `onRowError = skip` can skip. In a comma-delimited triple file a quoted cell is an N-Triples literal, not a CSV field. *(Since v26.12.1: before, a plain literal was keyed without its quotes and could merge with the IRI of the same text, the empty literal `""` was stored as `.`, escapes were not decoded, and an import into a database without the `Node` type failed. A database filled by an older version keeps the old unquoted keys, so importing the same data into it again stores those literals a second time under the new form.)*

Every line of an RDF source is data: unlike CSV, these formats have no header row, so nothing is skipped unless you ask for it - with `edgesSkipEntries` on the `edges` route, `verticesSkipEntries` on `vertices`, or `documentsSkipEntries` on `documents` and on a plain `url` import.

If you supply the column names yourself - with `documentsHeader`, `verticesHeader` or `edgesHeader` - the file is taken to have no header row, so its first line is imported as data. Setting a `⋯SkipEntries` option explicitly always wins over both defaults.

Lines at the top of a source that start with `#` or `//` are treated as comments and ignored, for every format - CSV, RDF, XML, JSON and JSONL alike.

A JSON source is recognised by its first character that is not whitespace: `{` for an object, `[` for an array of records. A source whose name ends in `.json` or `.xml` but whose content does not begin like that format is refused, naming the extension, rather than analysed as delimited text; if the content is what you meant, set the format explicitly with `fileType` (or `documentsFileType` on the `documents` route). *(Since v26.11.1: a top-level JSON array was not recognised, so a one-line array was imported as a CSV file whose header was the whole array, creating nothing while reporting success, and a JSON document opening with whitespace was not recognised either.)*

An ArcadeDB export is recognised by its first line, the `"info"` header the exporter writes, whatever the file is named: `backup.tgz` imports like `backup.jsonl.tgz`. A JSON source holds one top-level value, so a source with more content after its first object or array, such as newline-delimited JSON that is not an ArcadeDB export, is refused rather than imported in part; wrap its objects in an array. An ArcadeDB export forced to `fileType = 'json'` is refused naming `jsonl`. *(Since v26.12.1, issue #11729: only the `.jsonl` extension recognised an export, so one named `backup.tgz` was read as JSON: its header line was imported as one `Document`, every record after it was dropped, and the import reported `OK`.)*

Example of loading the Discogs dataset in the database on path "/temp/discogs":

```
> IMPORT DATABASE https://discogs-data.s3-us-west-2.amazonaws.com/data/2018
```

```
/discogs_20180901_releases.xml.gz
```

Note that in this case the URL is `https` and the file is compressed with `GZip`.

Example of importing New York Taxi dataset in CSV format. The first line of the CSV file set the property names:

```
> IMPORT DATABASE file:///personal/Downloads/data-society-uber-pickups-in-nyc/
original/uber-raw-data-april-15.csv/uber-raw-data-april-15.csv
```

See also:

- SQL Import Database command,
- Graph Importer (high-performance, declarative graph import from XML, CSV, JSONL with JSON configuration)
- JSON Importer
- Neo4j Importer,
- OrientDB Importer

7.1.1. Additional Settings

The Importer takes additional settings as pairs of setting name and value. With the SQL `IMPORT DATABASE` command, this is the syntax:

```
IMPORT DATABASE <url> [ WITH ( <setting-name> = <setting-value> [,] )* ]
```

Example:

```
> IMPORT DATABASE file:///import/file.csv WITH forceDatabaseCreate = true, commitEvery
= 100
```

7.1.2. Transactions

By default an import manages its own transactions, committing every `commitEvery` records, so nothing is left open when it finishes.

If a transaction is already open when the import starts - because you are using the Java API against your own `Database`, or you ran `IMPORT DATABASE` inside an explicit transaction - the import does **not** take it over. It adds the imported records to your transaction and leaves it as it found it: still open, still yours to commit or roll back. Nothing is made durable until you commit, and rolling back discards the imported records along with your own pending changes.

In that case `commitEvery` has no effect, since the import commits nothing of its own. For large imports prefer letting the import manage its own transaction, so it can commit in batches instead of

holding everything in memory until you commit.

Over HTTP, `IMPORT DATABASE` sent to `POST /api/v1/command` runs without the transaction the server otherwise opens around every command, so it commits in batches like the console does. Inside a session transaction (`/begin` ... `/commit`), or with `"autoCommit": true` in the request, it is added to that transaction as described above. *(Since v26.12.1, issue #11728: the transaction the server opened around the command was taken for yours, so the whole import was held in memory until it ended and a large one could run the server out of heap;* `"autoCommit": false` *was the workaround.)*

An import that fails still leaves your transaction alone: it discards only its own pending work and reports the error. *(Since v26.10.2: a GraphML import, or a GraphSON import with RID ids, committed your open transaction, so your rollback did not take the imported records back. A standalone GraphML or GraphSON import that failed could leave its last batch open on the thread instead of rolling it back.) (Since v26.12.1, issue #11271: a JSONL import whose commit was refused - a rolled-back-only transaction, a server no longer accepting writes - left its transaction open on the thread, so your next* `begin()`/`commit()` *pair nested inside it. It is now rolled back. With* `onRowError = skip`*, a JSONL record whose own commit is refused, for example by a duplicate key, is skipped and counted in* `errors` *instead of failing the whole import, and a later edge or link naming it is treated as naming a record that is not in the file.)*

7.1.3. CSV rows and the header

A CSV row is matched against the header by column count.

A row with **fewer** columns than the header is imported with the columns it does supply; the remaining properties are left unset, and the row is counted under `warnings`.

A row with **more** columns than the header cannot be imported as it stands - the extra values have no field name to be stored under - so by default the import fails, naming the line and both column counts. Set `onRowError` to `skip` to have such rows skipped and counted under `errors` instead.

Note the consequence for large files: `analysisLimitEntries` and `analysisLimitBytes` bound only the analysis, so a row with extra columns beyond that window fails the import too.

If the analysis reads only rows it has to refuse, it cannot derive any property and the import fails saying so, naming the analysis limit when one is set - widening it is usually the fix.

7.1.4. Column types of CSV and XML sources

The analysis of a CSV or XML source declares a property `LONG` when every value of it is the canonical decimal text of an integer (`0`, `42`, `-12`), `DOUBLE` when every value is a plain decimal number (`1.5`, `-0.25`, `1e5`), and `STRING` otherwise. A value that only a programming language reads as a number keeps the column a `STRING`, so its text is stored as it is: a leading `+`, leading zeros (`007`, the ZIP code `02134`), a type suffix (`10D`, `3F`), hexadecimal (`0x1p3`), `NaN`, `Infinity`, surrounding whitespace, a dangling dot (`1.`), an integer too large for a `LONG` and a number too large for a `DOUBLE`. *(Since v26.11.1: those values were accepted as numbers, so* `02134` *was stored as* `2134` *and* `10D` *as* `10.0`*, with the original text kept nowhere. A column that relied on it - for example* `+5`*-style values - is now imported as* `STRING`*; create the property with the type you want before the import to convert it explicitly.)*

7.1.5. TimeSeries samples in a JSONL file

The samples of a TimeSeries type are read back through the conversion every other way of writing to such a type shares (the line protocol, gRPC and the Prometheus remote write): a value is stored as the type of its column, so text a column can hold is stored as the value it names - `"80"` in an `INTEGER` tag, `"false"` in a `BOOLEAN` tag, a date in a `DATE` tag. The timestamp is a column like the others: text of digits, or a date, is read as the instant it names. A sample carrying a value no column can hold is refused by name, as `<type>.<column> (tag|field|timestamp): <reason>`, and `onRowError` decides what follows: by default the import fails and nothing of that chunk of samples is imported; with `skip` only the refused samples are dropped and counted under `errors`, and the others of their chunk are kept.

(Since v26.11.1: the importer handed the engine the raw values of the file, so text for a typed column aborted the whole import with `Failed to append timeseries samples`, naming the column but not the sample, and `skip` dropped the entire chunk.)

7.1.6. Import report

An import returns a summary of what it did, with the counters that apply to the source: `parsedRecords`, `createdRecords`, `createdDocuments`, `createdVertices`, `createdEdges`, `skippedRecords`, `skippedEdges`, `errors` and `warnings`. `parsedRecords` and `createdRecords` (the records of every kind the import created) are always present; every other counter is left out of the summary when it is zero.

A source that holds no records is a valid import, such as a CSV file with only its header, an empty JSON array or an XML document with only its root element. The import ends with `result OK`, and `warningMessages` names the source and the format it was read as, which is what tells such a source from one read as the wrong format. An import that cannot have read anything fails instead: one that names no source (`url`, `documents`, `vertices` or `edges`), a setting that is not in the list below, an import whose every source has zero bytes, whatever the format, and an HTML page in place of data, unless it was asked for as XML. A source of zero bytes named beside one that is not empty is skipped, with a message in `warningMessages`.

(Since v26.12.1.) These four cases used to end with `result OK`, or depended on the format. A script that passes a setting the importer never read now fails with `Unknown importer setting`. That includes `efConstruction` (use `beamWidth`), `ef` and `typeIdPropertyIsUnique` (use `typeIdUnique`), none of which had any effect.

`parsedRecords` counts every row read from every file of the import, including any header row and any row that was skipped. It is therefore normally higher than the number of records created.

`skippedRecords` counts rows that were skipped on purpose - by a `*SkipEntries` setting, or by the default header-row skip - and `skippedEdges` counts edges that were not created because their `from` or `to` reference did not match any vertex. Neither is counted as an error: `parsedRecords` equals `createdDocuments` + `createdVertices`
`createdEdges` + `skippedRecords` + `skippedEdges` + `errors`.

The Neo4j, GraphSON and OrientDB imports report the same way: an edge whose endpoint is not in the source is not created and is counted in `skippedEdges`. *(Since v26.12.1: the Neo4j import counted every relationship of the file in `skippedEdges` and the dropped ones under `warnings`, and the GraphSON and OrientDB imports counted them nowhere in the report.)*

Below you can find all the supported settings for the Importer.

Setting	Default	Description
url		url of the file to import
fileType		The format of the `url` source (csv, json, jsonl, xml, graphml, graphson), when its content or extension is not enough to recognise it. The `url` counterpart of `documentsFileType` *(since v26.11.1)*
database	./databases/imported	Path of the final imported database
forceDatabaseCreate	false	If true, the database is created brand new at every import
wal	false	Use the WAL (journal) for the importing. If the WAL is enabled the importing process will be much slower and will require much more RAM
commitEvery	5000	Create transactions that commit every X records
onRowError	abort	`abort` fails the whole import as soon as one record (document, vertex or edge) cannot be imported. `skip` logs the error, counts it in the `errors` result, and continues with the next record instead - trading import throughput (each record then commits in its own transaction) for the guarantee that a bad record can never take a good one down with it, nor leave a partially-written record behind. Applies to the JSON, JSONL, CSV and RDF importers. It also governs a CSV row with more columns than the header - see CSV rows and the header. It does not rescue a source from which the analysis could derive no property at all: that fails either way, since the alternative is an import that completes having produced nothing.
parallel	Half of the available cores - 1. If you have 8 cores, the default is 3	The number of concurrent threads used
typeIdProperty		Property that represents the ID of the vertex. Also used by an RDF/N-Triples source to key its subjects and objects, where it defaults to `id`. An explicit value is honoured for RDF too
typeIdUnique	false	True creates a unique index on the type id property, otherwise a non unique index
typeIdType	String	Type of the id property
trimText	true	True if the imported text is trimmed from heading and tailing spaces

Setting	Default	Description
maxProperties	512	Maximum number of properties per type (CSV)
maxPropertySize	4096	Maximum size of a property in bytes (CSV). For an RDF source, the maximum length in characters of each term
delimiter	,	Delimiter used to separate fields (CSV). When not set, it is auto-detected from the first line for files whose name does not end in .csv. A delimiter set explicitly always wins over the detected one; the precedence is the per-entity delimiter (documentsDelimiter, verticesDelimiter, edgesDelimiter), then delimiter, then the detected one. Each file of the same import resolves its own delimiter. A delimiter longer than one character is supported. An empty delimiter is refused with an error
analysisLimitBytes	100,000	Maximum number of bytes parsed from the source to determine the source file type. It also applies to XML sources
analysisLimitEntries	10,000	Maximum number of entries (if applicable) parsed from the source to determine the source file type. It also applies to XML sources
parsingLimitBytes		Maximum number of bytes parsed from the source to be imported. Enforced on every format
parsingLimitEntries		Maximum number of entries imported. This is a cap of exactly N entries, enforced on every format
mapping	null	
probeOnly	false	Only probe if url is reachable or file path is readable
documents		url of the file to import containing documents only. This is useful when the database is split in separate files
documentsFileType		The format of the file containing documents (csv, json, jsonl, xml, graphml, graphson). graphml and graphson require the optional arcadedb-gremlin module on the classpath; without it the import is refused
documentsDelimiter		Delimiter used to separate documents
documentsHeader		Header containing the properties in the CSV document. One property per column. If not defined it is parsed from the first line. When you do define it, the file is assumed to contain no header row, so its first line is imported as data. The names are separated by the delimiter in force, not by a comma

Setting	Default	Description
documentsSkipEntries		Number of rows to skip from the documents file. When not set, the first line is skipped as the header for delimited text - unless you supplied `documentsHeader`, in which case nothing is skipped - and nothing is skipped for RDF sources, which have no header row. Setting it explicitly always wins
documentPropertiesInclude	*	List of property to import from documents. * means all
documentType	Document	Name of the type defined in the schema when importing documents
vertices		url of the file to import containing vertices only. This is useful when the database is split in separate files
verticesFileType		The format of the file containing vertices (csv, graphml, graphson). See `documentsFileType` for the `arcadedb-gremlin` requirement
verticesDelimiter		Delimiter used to separate vertices
verticesHeader		Header containing the properties in the CSV vertices. One property per column. If not defined it is parsed from the first line. When you do define it, the file is assumed to contain no header row, so its first line is imported as data. The names are separated by the delimiter in force, not by a comma
verticesSkipEntries		Number of rows to skip from the vertices file. When not set, the first line is skipped as the header for delimited text - unless you supplied `verticesHeader`, in which case nothing is skipped - and nothing is skipped for RDF sources, which have no header row. Setting it explicitly always wins
expectedVertices	0	Number of vertices expected. This is useful to determine the ETA of the importing process of vertices. 0 means unknown
vertexType	Vertex	Name of the type defined in the schema when importing vertices
vertexPropertiesInclude	*	List of property to import from vertices. * means all
edges		url of the file to import containing edges only. This is useful when the database is split in separate files
edgesFileType		The format of the file containing edges (csv, graphml, graphson). See `documentsFileType` for the `arcadedb-gremlin` requirement
edgesDelimiter		Delimiter used to separate edges

Setting	Default	Description
edgesHeader		Header containing the properties in the CSV edges. One property per column. If not defined it is parsed from the first line. When you do define it, the file is assumed to contain no header row, so its first line is imported as data. The names are separated by the delimiter in force, not by a comma
edgesSkipEntries		Number of rows to skip from the edges file. When not set, the first line is skipped as the header for delimited text - unless you supplied `edgesHeader`, in which case nothing is skipped - and nothing is skipped for RDF sources, which have no header row. Setting it explicitly always wins
expectedEdges	0	Number of edges expected. This is useful to determine the ETA of the importing process of edges. 0 means unknown
maxRAMIncomingEdges	256MB	Maximum RAM used to create edges. The more RAM, the faster.
edgeType	Edge	Name of the type defined in the schema when importing edges
edgePropertiesInclude	*	List of property to import from edges. * means all
edgeFromField		Name of the property containing the starting vertex
edgeToField		Name of the property containing the ending vertex
edgeBidirectional	true	When creating edges, create bidirectional edges if true, otherwise unidirectional
distanceFunction	`innerproduct`	Type of distance measure, see similarity measures.
beamWidth	100	Size of the candidate list kept while the vector index is built. Higher values improve recall but slow the build
m	16	Maximum number of connections per layer in the HNSW index. Higher values improve recall but increase memory usage
vectorType	`float`	The data type of a vector element, for example 'float'.
vectorProperty	`"vector"`	Name of the property holding the vector of a Word2Vec or GloVe import. *(Since v26.11.1: it had no default, and an import that did not set it failed with a `NullPointerException`)*
idProperty	`"name"`	Name of the property that will be used as the unique identifier for vertices during import. A blank `idProperty` or `vectorProperty` is refused, naming the setting

The `probeOnly` setting can also be used to send a GET request to another service or HTTP API, for example to report a previous import is finished.

7.2. Graph Importer

The `GraphImporter` is a high-performance, declarative graph importer that uses a two-pass CSR-first architecture to bulk-load graph data from XML, CSV, and JSONL files into ArcadeDB. It is designed for importing large datasets (millions of vertices and edges) with minimal memory overhead.

The importer is located in the `integration` module (`com.arcadedb.integration.importer.graph.GraphImporter`).

Never run the importer against a database directory that an ArcadeDB Server (or any other process) already has open. It opens the database files directly, bypassing the server, and two processes writing to the same database at once can corrupt it. To load data into a database a server is serving, use the server's remote protocol instead, or stop the server for the duration of the import.

7.2.1. How It Works

The import runs in two passes:

1. **Pass 1** — Process each vertex source once: create vertices with full properties, and collect the edges a vertex source declares (foreign keys, split fields) as compressed int arrays
2. **Pass 2** — Create all edges using GraphBatch, one batch per edge type with bidirectional edges for full IN+OUT traversal: first the collected ones, then each edge-only source, streamed from its file straight into its batch *(since v26.10.2; before, edge-only sources were collected in memory first)*. The first edge of each vertex the import created, in each direction, starts its edge list without reading the vertex back from disk; only a vertex that an earlier edge type has already given edges is read, once per edge type *(since v26.11.1; before, the edge pass read every vertex it connected, which on a graph larger than the memory stalled it for minutes every 500,000 edges)*. `GraphImporter.getVertexHeadReads()` reports how many vertices were read. A vertex that comes back in a later flush of the same edge type, as happens when the file is not sorted by that end of the edge, gets a new edge-list segment linked to the previous one without reading it back *(since v26.11.1; before, each of those reads was a random disk read on a graph larger than the memory, slowing the edge pass to about 1,400 edges/s)*
3. **Vector graphs** — Build the graph of every `LSM_VECTOR` index on a type the import wrote to, so the index answers at index speed the moment `run()` returns (see Loading into a type with a vector index)

The speculative background maintenance of the database's indexes (the vector index's inactivity rebuild) is held off for the whole run, not only while one of the batches is open: a stall between the two passes must not be read as "the load is over".

7.2.2. Memory and Progress

What the importer keeps in memory grows with the number of vertices, not with the number of edges *(since v26.10.2)*. Per vertex it holds:

- its RID (8 bytes);

- its identity in a hash map (11 to 17 bytes, or 17 to 26 when the key does not fit in an `int`);
- while the edges are written, the head of its two edge lists in the edge batch (32 to 48 bytes).

That is roughly 12 to 17 GB of heap for 200 million vertices. The page cache (`arcadedb.maxPageRAM`, a quarter of the heap by default) comes on top of it. Size `-Xmx` for both, or lower `arcadedb.maxPageRAM` to leave the importer more of the heap.

Edge segments are created at the size their edges need. No empty segment is pre-allocated for each vertex: before v26.10.2 the vertex pass wrote 2 KB per vertex, about 400 GB for 200 million vertices.

Each pass logs how far it has got every 30 seconds, with the rate, the heap in use and the share of the time spent in garbage collection pauses:

```
  WORK         48,350,000 vertices so far (152,331/s, heap 9.12GB of 16.00GB, 3% of
the time in GC)
  CITE         412,876,800 edge rows so far (391,004/s, heap 13.71GB of 16.00GB, 4% of
the time in GC)
```

When more than half of an interval goes to garbage collection, a warning says the heap is too small for the load. A JVM short of heap rarely fails with an `OutOfMemoryError`: it collects back to back and the load slows to a crawl, which from the outside looks like a hang. Restart the import with a larger `-Xmx` or a smaller `arcadedb.maxPageRAM`.

A row of an edge-only source that cannot be read aborts the import. The edges of the rows before it are already written and connected, and a `PARTIAL` warning says how many, the same way a failing vertex source reports the vertices its earlier commits made durable.

7.2.3. Command-Line Usage

```
java -cp arcadedb-integration-*.jar
com.arcadedb.integration.importer.graph.GraphImporter \
  <json-config-file> <database-path> [data-dir]
```

- `json-config-file` — Path to the JSON configuration file (see JSON Configuration below)
- `database-path` — Path where the database will be created (any existing database at this path is deleted)
- `data-dir` — Optional base directory for resolving relative file paths in the JSON config (defaults to the JSON file's parent directory)

7.2.4. Java API

The importer can also be used programmatically via a fluent Builder API. It commits in batches of its own, so `run()` must be called with no transaction open: it fails with an `IllegalStateException` otherwise, rather than committing work your rollback could not undo.

```
try (GraphImporter importer = GraphImporter.builder(database)
```

```
    .vertex("User", new CsvRowSource("users.csv"), v -> {
        v.id("Id");
        v.intProperty("reputation", "Reputation");
        v.property("name", "DisplayName");
    })
    .vertex("Question", new XmlRowSource("posts.xml"), v -> {
        v.id("Id");
        v.filter("PostTypeId", "1");
        v.property("title", "Title");
        v.edgeIn("OwnerUserId", "ASKED", "User");
        v.splitEdge("Tags", "TAGGED_WITH", "Tag", "|");
    })
    .edgeSource("LINKED_TO", new CsvRowSource("links.csv"), e -> {
        e.from("PostId", "Question");
        e.to("RelatedId", "Question");
        e.intProperty("linkType", "LinkTypeId");
    })
    .limit(10000)
    .build()) {

  importer.run();
  System.out.printf("Vertices: %,d, Edges: %,d%n",
      importer.getVertexCount(), importer.getEdgeCount());
}
```

Or from a JSON configuration file:

```
String json = new String(Files.readAllBytes(jsonFile.toPath()));
JSONObject config = new JSONObject(json);

GraphImporter.createSchemaFromConfig(database, config);

try (GraphImporter importer = GraphImporter.fromJSON(database, config, dataDir)) {
  importer.run();
}

GraphImporter.executePostImportCommands(database, config);
```

7.2.5. JSON Configuration

The JSON configuration file defines vertex types, edge types, data sources, property mappings, and optional post-import commands.

Vertex Definitions

Each entry in the `vertices` array defines a vertex type and its data source:

Key	Required	Description
`type`	Yes	ArcadeDB vertex type name (auto-created if it does not exist)
`file`	Yes	Source file path, relative to the data directory. Format is auto-detected from extension: `.xml`, `.csv`, `.jsonl`
`id`	Yes	Source attribute used as the primary key for edge resolution between types. The value is taken as written, so an integer, a value too large for an integer, and a string such as `W13696992` all work. Two spellings the file keeps apart stay two different keys: `007` and `7` are not the same vertex. An empty value registers no key
`nameId`	No	Secondary key, used by `split` edges and by `byName` edges to resolve values by name. Declare it only when a type is referenced through two different keys — a numeric id from one file and a name from another. A single string key needs nothing more than `id`
`filter`	No	Row filter in the format `attribute=value`. Only matching rows are imported. This enables splitting one file into multiple vertex types. An empty value (`attribute=`) selects the rows that do not set the attribute
`element`	No	For XML files: element name to read (defaults to `row`)
`properties`	No	Maps ArcadeDB property names to source attributes, optionally with a type prefix. See Property Types
`edges`	No	Array of edge definitions derived from foreign key attributes in this vertex's source file

Property Types

A property mapping is `"dbPropertyName": "sourceAttribute"`, where the source attribute may carry a type prefix. Without a prefix the value is imported as a string. The same prefixes are accepted in vertex `properties` and in edge-source `properties`.

An empty value means "not set" in every source format.

Prefix	Stored type	Notes
(none)	`STRING`	Null or empty source values are skipped
`int:`	`INTEGER`	Missing or empty values default to `0`
`long:`	`LONG`	Missing or empty values default to `0`
`double:`	`DOUBLE`	Missing or empty values default to `0.0`
`bool:`	`BOOLEAN`	`true` when the value is `True`/`true`
`datetime:`	`DATETIME`	Default format `yyyy-MM-dd HH:mm:ss`. For a custom pattern use `datetime:FORMAT|attribute`, with both halves present

Prefix	Stored type	Notes
`vector:`	`ARRAY_OF_FLOATS` (`float[]`)	A dense float vector (an embedding). JSONL reads the JSON array natively; CSV and XML parse the textual form `[0.1,0.2,0.3]`. See Arrays in a CSV file
`list:`	`LIST`	A generic array, e.g. a list of tags

Every prefix has to be followed by the name of the source attribute to read: a value such as `"int:"` on its own, or a `datetime:` pattern missing one half of its `FORMAT|attribute` pair, is reported as a configuration error. The same goes for a missing `type`, `edge`, `target`, `attribute` or `file` key: the message names the key, the source that declares it, and what the value is for.

Importing Vector Embeddings

Embedding pipelines usually emit JSONL with the vector inline as a JSON array:

```
{"id": 1, "title": "The Hitchhiker's Guide to the Galaxy", "tags": ["scifi",
"comedy"],
 "embedding": [-0.31142, 0.51346, -0.02326, -0.29451, 0.57215]}
```

Declare the column with the `vector:` prefix and create the index afterwards with a post-import command:

```
{
  "vertices": [
    {
      "type": "Book", "file": "books.jsonl", "id": "id",
      "properties": {
        "title": "title",
        "tags": "list:tags",
        "embedding": "vector:embedding"
      }
    }
  ],
  "postImportCommands": [
    { "language": "sql",
      "command": "CREATE INDEX ON Book (embedding) LSM_VECTOR METADATA { dimensions:
5, similarity: 'COSINE' }" }
  ]
}
```

The property is stored as a primitive `float[]`, not as a list of boxed numbers: a 768-dimension embedding costs roughly 3KB per vertex instead of 18KB, which matters on a bulk load of millions of rows, and it is the exact representation the vector index consumes, so building the index converts nothing.

Use `vector:` only for numeric arrays. `list:` keeps the array as a generic list and is the right choice

for heterogeneous or non-numeric arrays such as tags.

The equivalent Java API calls are `floatArrayProperty(name, attribute)` and `listProperty(name, attribute)`.

Loading into a type with a vector index

The index can also exist before the import. Every vector the load writes goes into the index's delta buffer, where it is searchable by a linear scan, and the HNSW graph that makes the search fast is built afterwards. Left to the index, that build starts once the index has been quiet for its inactivity window, on a background thread - and a `database.close()` right after `run()`, which is what a loader normally does, cancels it and throws the work away.

So the importer builds it itself: when `run()` returns, every `LSM_VECTOR` index on a type the import wrote to has its graph built and persisted, and the process can close the database at once. The build is logged with its progress and takes as long as `CREATE INDEX` would over the same vectors. An index whose graph is already current is skipped.

To leave the build to the index's background rebuild instead - only worth it when the database stays open long enough for that build to complete - set `"vectorGraphBuild": false` in the JSON configuration, or `.withVectorGraphBuild(false)` on the builder.

Arrays in a CSV file

CSV and XML have no array of their own, so an array column is read from its textual form `[0.1,0.2,0.3]`. In a CSV file, choose a delimiter the array does not contain - `;` rather than the default `,` - because the CSV reader splits on the delimiter without honouring quotes, and a comma-delimited file would cut the array across several columns:

```
Id;Title;Embedding
1;The Hitchhiker's Guide to the Galaxy;[-0.31142,0.51346,-0.02326]
```

If the array is split anyway, the import stops and says so, naming the property and the delimiter.

Edge Definitions (within a vertex)

Each entry in a vertex's `edges` array defines how to create edges from foreign key attributes:

Key	Required	Description
`attribute`	Yes	Source attribute containing the foreign key value
`edge`	Yes	ArcadeDB edge type name (auto-created if it does not exist)
`target`	Yes	Target vertex type the foreign key references
`direction`	No	`out` (default): this vertex → target. `in`: target → this vertex
`split`	No	Delimiter for multi-value fields (e.g., `|`). One edge is created per value, resolved by the target's `nameId`

Edge-Only Sources

The `edgeSources` array defines edges where both endpoints already exist as vertices. No vertices are created from these sources, and their rows are streamed into the database rather than held in memory, so an edge file of billions of rows needs no more heap than a small one (see Memory and Progress):

Key	Required	Description
`edge`	Yes	ArcadeDB edge type name
`file`	Yes	Source file path
`from`	Yes	Compact format `attribute:vertexType` — source attribute and its vertex type. Matched against that type's `id`, whatever the value looks like
`to`	Yes	Compact format `attribute:vertexType` — target attribute and its vertex type
`properties`	No	Property mappings (same format as vertex properties)

General Options

Key	Required	Description
`limit`	No	Maximum records per source (for testing). Omit or set to 0 for unlimited
`vectorGraphBuild`	No	Whether to build the graph of every `LSM_VECTOR` index on a type the import wrote to before `run()` returns (default `true`). See Loading into a type with a vector index

Post-Import Commands

The `postImportCommands` array defines commands to execute automatically after the graph import completes. This is useful for creating indexes, analytical views, or running any database command that depends on the imported data being present.

Key	Required	Description
`language`	Yes	Query language to use: `sql`, `opencypher`, etc.
`command`	Yes	The command text to execute

Commands are executed sequentially in the order they appear. If a command fails, a warning is logged and the remaining commands continue to execute.

The entries are validated before the import starts, so a missing or empty `language` or `command` is reported straight away rather than after every row has been read.

If any post-import command triggers an asynchronous Graph Analytical View build, the importer automatically waits (up to 10 minutes) for all views to reach READY status before returning.

Example:

```
"postImportCommands": [
  {
    "language": "sql",
    "command": "CREATE INDEX ON Question (Id) UNIQUE"
  },
  {
    "language": "sql",
    "command": "CREATE GRAPH ANALYTICAL VIEW IF NOT EXISTS myGraph PROPERTIES
(`!Body`, `!Text`) UPDATE MODE SYNCHRONOUS"
  }
]
```

7.2.6. Complete Example

Below is a complete JSON configuration for importing a StackOverflow data dump:

```
{
  "vertices": [
    {
      "type": "Tag", "file": "Tags.xml", "id": "Id", "nameId": "TagName",
      "properties": { "Id": "int:Id", "TagName": "TagName", "Count": "int:Count" }
    },
    {
      "type": "User", "file": "Users.xml", "id": "Id",
      "properties": {
        "Id": "int:Id", "DisplayName": "DisplayName", "Reputation": "int:Reputation",
        "CreationDate": "CreationDate", "Views": "int:Views"
      }
    },
    {
      "type": "Question", "file": "Posts.xml", "id": "Id", "filter": "PostTypeId=1",
      "properties": {
        "Id": "int:Id", "Title": "Title", "Body": "Body",
        "Score": "int:Score", "ViewCount": "int:ViewCount", "Tags": "Tags"
      },
      "edges": [
        { "attribute": "OwnerUserId", "edge": "ASKED", "target": "User", "direction":
"in" },
        { "attribute": "Tags", "edge": "TAGGED_WITH", "target": "Tag", "split": "|" }
      ]
    },
    {
      "type": "Answer", "file": "Posts.xml", "id": "Id", "filter": "PostTypeId=2",
      "properties": {
        "Id": "int:Id", "Body": "Body", "Score": "int:Score"
      },
      "edges": [
        { "attribute": "OwnerUserId", "edge": "ANSWERED", "target": "User",
"direction": "in" },
```

```
        { "attribute": "ParentId", "edge": "HAS_ANSWER", "target": "Question",
"direction": "in" }
      ]
    }
  ],

  "edgeSources": [
    {
      "edge": "ACCEPTED_ANSWER", "file": "Posts.xml",
      "from": "Id:Question", "to": "AcceptedAnswerId:Answer"
    },
    {
      "edge": "LINKED_TO", "file": "PostLinks.xml",
      "from": "PostId:Question", "to": "RelatedPostId:Question",
      "properties": { "LinkType": "int:LinkTypeId" }
    }
  ],

  "postImportCommands": [
    {
      "language": "sql",
      "command": "CREATE GRAPH ANALYTICAL VIEW IF NOT EXISTS stackoverflow PROPERTIES
(`!Body`, `!Text`) UPDATE MODE SYNCHRONOUS"
    }
  ]
}
```

7.2.7. Identities and Missing Edges

Every edge is resolved by matching the value of a foreign key against the `id` of the vertex type it points at. The importer compares those values as text, so any key your files already use works without being converted first:

```
{"id": "W13696992", "title": "A paper", "publication_year": 2008}
```

```
from_id,to_id
2257721487,28031933
```

When an edge names a key that no vertex carries, the edge is skipped — there is nothing to attach it to — but the importer now counts it and logs a warning per source, and the total is available from `getUnresolvedEdgeCount()` in the Java API. A graph that comes out smaller than the files you fed in no longer does so silently.

Four configurations that could only ever produce zero edges are now rejected with an error before anything is written, instead of importing a quietly incomplete graph:

- an edge pointing at a vertex type that no source imports (usually a typo in the type name)

- an edge pointing at a vertex type that is imported **after** the source referencing it. A vertex source can only resolve against types already imported, so declare the target's source first
- an edge resolving against the `id` of a type that declares only `nameId`, or the reverse
- two vertex sources declaring the same type name

7.2.8. Supported File Formats

The file format is auto-detected from the file extension:

Extension	Format
`.xml`	XML elements (configurable element name, defaults to `row`)
`.csv`	CSV with header row (first line defines property names)
`.jsonl`	JSON Lines (one JSON object per line)

7.2.9. Data Sources (Java API)

When using the Java API, you can use the following `RecordSource` implementations:

- `CsvRowSource` — reads CSV files
- `XmlRowSource` — reads XML files
- `JsonlRowSource` — reads JSONL files

Custom data sources can be implemented via the `RecordSource` interface.

7.3. JSON Importer

ArcadeDB is able to import data from JSON format. Thanks to the flexible mapping, it's possible to define the rules of conversion between the input json file and the graph.

Without a `mapping`, a JSON source is imported as plain documents of the type set by `documentType` (default `Document`): a single top-level object becomes one document, and a top-level array becomes one document per object in it, each committed in its own transaction so that `onRowError = skip` can discard one without the others. An array entry that is not an object is refused, naming its position. Numbers keep their exact value: with or without a mapping, every JSON number is decoded exactly as the HTTP API decodes it. An integer is stored as an INTEGER when it fits 32 bits, a LONG when it fits 64 bits and a DECIMAL beyond. Any other number is stored as a DOUBLE, unless a DOUBLE would lose some of its digits (it has more than 17 significant digits) or cannot hold its magnitude (such as `1e309`), and then as a DECIMAL. A number token that only a hostile payload writes (over 1000 characters, or an exponent beyond 1000) is refused as a row error. *(Since v26.11.1: a top-level array was not recognised as JSON at all - a one-line array was imported as a CSV file holding nothing and the import reported success - and every number was stored as a DOUBLE, so `42` became `42.0` and `9007199254740993` became `9007199254740992`.)*

Let's start from this simple JSON file to import as an example. The file contains a flat structure of records with a relationship of employee and manager:

```
{
  "Users":[
    {
      "EmployeeID":"1234",
      "ManagerID":"1230",
      "Name":"Marcus"
    },
    {
      "EmployeeID":"1230",
      "ManagerID":"1231",
      "Name":"Win"
    }
  ]
}
```

First, create a new database "employees" by using the console:

```
> CREATE DATABASE employees
```

Then execute the following command (assuming the file above is saved in the local directory as `employees.json`:

```
{employees}> IMPORT DATABASE employees.json WITH mapping = {
    "Users": [{
            "@cat": "v",
            "@type": "User",
            "@id": "id",
            "@idType": "string",
            "@strategy": "merge",
            "EmployeeID": "@ignore",
            "ManagerID": {
                    "@cat": "e",
                    "@type": "HAS_MANAGER",
                    "@cardinality": "no-duplicates",
                    "@in": {
                            "@cat": "v",
                            "@type": "User",
                            "@id": "id",
                            "@idType": "string",
                            "@strategy": "merge",
                            "EmployeeID": "@ignore",
                            "id": "<../ManagerID>"
                    }
            }
    }]
}
```

The mapping file is a JSON snippet with directives about what to import, what to ignore and how to map edges with JSON objects. Below all the supported tags:

- `@cat`: is the category of record to use between "v" for vertex, "e" for edge and "d" for document
- `@type`: is the type name to use on record creation. If the type doesn't exist, it's implicitly created during the import
- `@id`: is the property that works as primary key. if not already defined, a unique index is created on the configured `@id` property
- `@idType`: is the type of the primary key in the `@id` property. If not defined, the type is taken from the first value found, widened for a number: an integral first id declares the property LONG and any other number DECIMAL, whatever the size of that first value, so a later id past 2^{31} still fits and two decimal ids that differ past the 17th digit stay two keys. You can, for example, use `@idType` DOUBLE to force a DOUBLE key anyway *(Since v26.11.1: every number was read as a DOUBLE, so an integer key was declared DOUBLE, an integer above 2^{53} lost its last digits, and two decimal ids that differ past the 17th digit collided in the unique index, so the second record was silently skipped as already imported)*
- `@cardinality`: if "no-duplicates", the edges are not created if there is an edge of the same type between the two vertices
- `@strategy`: represents the strategy to use when the record with the configured `@id` already exists By default the existent record is used, but the "merge" strategy allows to merge the current record with the existent one. The current properties will overwrite the existent ones
- `@in`: used for edges and represents the destination vertex for the edge
- `@ignore`: ignore the property, do not import into the database

Note the "id" property in the manager is taken by using `"<../ManagerID>"` that means get the property "ManagerID" from the parent (`../`).

How to import a JSON file with a hierarchical structure? Let's look at the second example coming from the JSON export of food and nutrients from the U.S. DEPARTMENT OF AGRICULTURE website (USDA). USDA provides updated files to download in both CSV and JSON format.

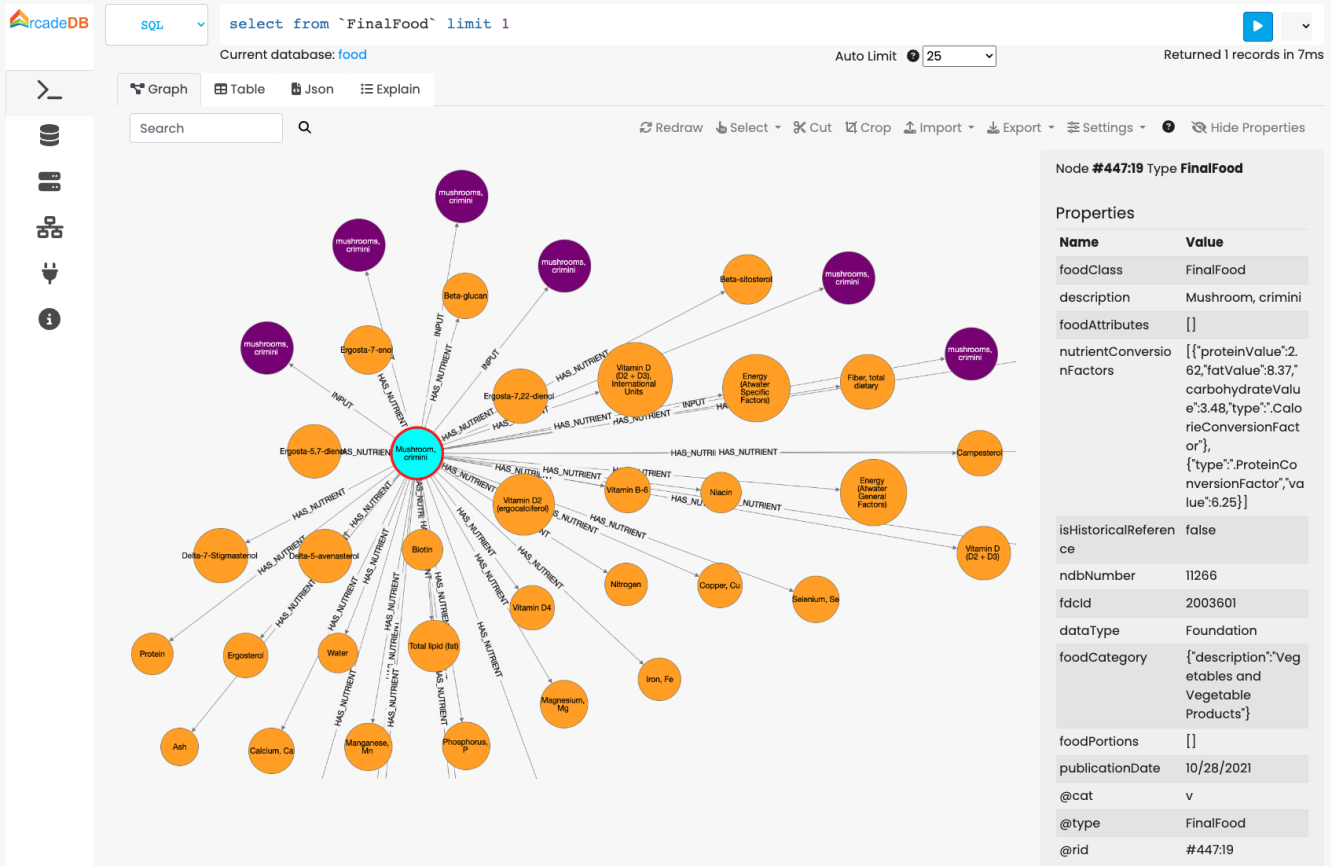
First, create a new database "food" by using the console:

```
> CREATE DATABASE food
```

Then execute the following command:

```
{food}> IMPORT DATABASE https://fdc.nal.usda.gov/fdc-datasets
/FoodData_Central_foundation_food_json_2022-10-28.zip WITH mapping = {
    "FoundationFoods": [{
        "@cat": "v",
        "@type": "<foodClass>",
        "foodNutrients": [{
            "@cat": "e",
            "@type": "HAS_NUTRIENT",
```

```
            "@in": "nutrient",
            "@cardinality": "no-duplicates",
            "nutrient": {
                "@cat": "v",
                "@type": "Nutrient",
                "@id": "id",
                "@idType": "long",
                "@strategy": "merge"
            },
            "foodNutrientDerivation": "@ignore"
        }],
        "inputFoods": [{
            "@cat": "e",
            "@type": "INPUT",
            "@in": "inputFood",
            "@cardinality": "no-duplicates",
            "inputFood": {
                "@cat": "v",
                "@type": "<foodClass>",
                "@id": "fdcId",
                "@idType": "long",
                "@strategy": "merge",
                "foodCategory": {
                    "@cat": "e",
                    "@type": "HAS_CATEGORY",
                    "@cardinality": "no-duplicates",
                    "@in": {
                        "@cat": "v",
                        "@type": "FoodCategory",
                        "@id": "id",
                        "@idType": "long",
                        "@strategy": "merge"
                    }
                }
            }
        }]
    }]
}
```



This command downloads the zip file from the USDA website and uses the mapping to create the graph from the JSON file.

The mapping file is a JSON snippet with directives about what to import, what to ignore and how to map edges with JSON objects. Below all the supported tags:

- `@cat`: is the category of record to use between "v" for vertex, "e" for edge and "d" for document
- `@type`: is the type name to use on record creation. If the type doesn't exist, it's implicitly created during the import
- `@id`: is the property that works as primary key. if not already defined, a unique index is created on the configured `@id` property
- `@idType`: is the type of the primary key in the `@id` property. If not defined, the type is taken from the first value found, widened for a number: an integral first id declares the property LONG and any other number DECIMAL, whatever the size of that first value, so a later id past 2^{31} still fits and two decimal ids that differ past the 17th digit stay two keys. You can, for example, use `@idType` DOUBLE to force a DOUBLE key anyway *(Since v26.11.1: every number was read as a DOUBLE, so an integer key was declared DOUBLE, an integer above 2^{53} lost its last digits, and two decimal ids that differ past the 17th digit collided in the unique index, so the second record was silently skipped as already imported)*
- `@cardinality`: if "no-duplicates", the edges are not created if there is an edge of the same type between the two vertices
- `@strategy`: represents the strategy to use when the record with the configured `@id` already exists By default the existent record is used, but the "merge" strategy allows to merge the current record with the existent one. The current properties will overwrite the existent ones

- `@in`: used for edges and represents the destination vertex for the edge
- `@ignore`: ignore the property, do not import into the database

7.4. OrientDB Importer

Migrating from OrientDB? The ArcadeDB Academy offers a **free** course on OrientDB-to-ArcadeDB migration with hands-on exercises and a certification at the end.

ArcadeDB is able to import a database exported from OrientDB in JSON format.

For more information about how to export a database from OrientDB, look at Export Database.

To import a database use the Import Database command from API, Studio or Console. Below you can find an example of importing a OrientDB database by using ArcadeDB Console.

```
> CREATE DATABASE MyDatabase
{MyDatabase}> IMPORT DATABASE file:///temp/orientdb-export.tgz
```

7.4.1. What the import preserves

- **Property types**: numbers are imported with the type OrientDB exported them with, not as floating point values. Integers stay integers, LONG values keep every digit (including those above 2^53), DECIMAL keeps its precision and FLOAT does not widen to DOUBLE. This applies both to properties the OrientDB schema declared and to schemaless ones.
- **Indexes**: unique and non-unique indexes are recreated, including multi-column (composite) ones, so a uniqueness constraint the source database enforced is still enforced after the migration. Any property an index needs that the export did not declare is created for it.
- An index whose key type has no ArcadeDB equivalent (`LINKBAG`, for example) is skipped and reported in the import log. The rest of the import is unaffected.

7.4.2. Differences with OrientDB

ArcadeDB was created from the same original developers of OrientDB, but it was built from scratch with a completely new codebase and architecture. While many concepts will feel familiar to OrientDB users, there are important differences to be aware of when migrating.

7.4.3. Architecture

- **No classes hierarchy**: ArcadeDB uses Types instead of Classes. Types support single inheritance but do not support the polymorphic class hierarchy that OrientDB offered.
- **Buckets instead of Clusters**: What OrientDB called "clusters" are called "buckets" in ArcadeDB. They serve a similar purpose of physically grouping records, but the API and behavior differ.
- **No Record-level security**: ArcadeDB does not implement record-level security. Security is managed at the database and server level.

- **Embedded only or Client-Server**: ArcadeDB runs either as an embedded database or in client-server mode. There is no "remote" storage engine concept as in OrientDB.
- **Component-based storage**: ArcadeDB uses an LSM-Tree based storage engine, which is fundamentally different from OrientDB's plocal/memory storage engines.

7.4.4. Data Model

- **No BLOB/Binary type**: ArcadeDB does not have a dedicated BLOB or ORecordBytes type. Binary data should be stored as Base64-encoded strings or managed externally.
- **No LINKBAG**: ArcadeDB uses simpler edge management for graphs and does not use the LINKBAG (ridbag) structure.
- **No CUSTOM field type**: The `CUSTOM` field type from OrientDB is not available.
- **No embedded geometry types**: ArcadeDB has no `OPoint`, `OLineString`, or `OPolygon` embedded types. Geometries are stored as WKT strings in `STRING` properties (see Geospatial Data below).
- **Record IDs**: ArcadeDB uses Record IDs (RIDs) with the same `#bucket:position` format, but the internal storage is different.

7.4.5. Geospatial Data (OPoint and other spatial types)

OrientDB's spatial module stored geometries as embedded documents of dedicated classes such as `OPoint`, `OLineString`, and `OPolygon`, each with a `coordinates` field. ArcadeDB has no embedded geometry type: geometries are stored as a plain `STRING` property containing the geometry in Well-Known Text (WKT) format, and are queried with the built-in `geo.*` SQL functions (backed by JTS and Spatial4j). See Geospatial Index for the full reference.

Both engines use the same axis order, **longitude first then latitude** (the GeoJSON/WKT convention), so coordinate values migrate directly without swapping.

	OrientDB	ArcadeDB
Storage	Embedded `OPoint` with `coordinates: [lon, lat]`	`STRING` property with WKT: `POINT (lon lat)`
Construction	`ST_GeomFromText('POINT(12.5 41.9)')`	`geo.point(12.5, 41.9)` or the literal WKT string
Functions	`ST_*` (e.g. `ST_Distance`, `ST_Within`)	`geo.*` (e.g. `geo.distance`, `geo.within`)
Index	Lucene spatial index	`CREATE INDEX … GEOSPATIAL` (see Creating a Geospatial Index)

To migrate `OPoint` data:

1. Define the target property as `STRING`:

```
CREATE PROPERTY City.location STRING
```

2. Convert each `OPoint` to WKT on insert. An OrientDB export serializes an `OPoint` as

`{"@class":"OPoint","coordinates":[12.5,41.9]}`; map `coordinates[0]` to longitude (x) and `coordinates[1]` to latitude (y):

```
INSERT INTO City SET name = 'Rome', location = 'POINT (12.5 41.9)'
-- or build it from the coordinates programmatically:
INSERT INTO City SET name = 'Rome', location = geo.point(12.5, 41.9)
```

3. Create a geospatial index (used automatically by `geo.within` and `geo.intersects`):

```
CREATE INDEX ON City (location) GEOSPATIAL
```

4. Translate `ST_*` queries to their `geo.*` equivalents:

```
-- distance with the Haversine formula (unit: 'km' default, or 'm')
SELECT geo.distance(location, geo.point(12.49, 41.90), 'km') AS dist_km FROM City

-- containment / proximity (index-accelerated). geo.buffer distance is in degrees
-- (~0.01 deg  1.1 km at the equator)
SELECT FROM City WHERE geo.within(location, geo.buffer(geo.point(12.5, 41.9), 0.
01)) = true
```

The same pattern applies to other OrientDB spatial types: use `geo.lineString(…)` for `OLineString` and `geo.polygon(…)` for `OPolygon`, or insert the corresponding `LINESTRING (…)` / `POLYGON …` WKT directly. Use `geo.x()` / `geo.y()` to extract raw coordinates (for example to validate migrated values against the originals) and `geo.asGeoJson()` for GeoJSON output.

7.4.6. SQL

- **Mostly compatible**: ArcadeDB SQL is largely compatible with OrientDB SQL, with some differences:
  - `LET` blocks in queries have some syntax differences.
  - `TRAVERSE` is supported but deprecated in favor of `MATCH` and recursive queries.
  - The `.asSet()`, `.asList()`, `.asMap()` methods are kept for OrientDB compatibility.
  - `DELETE … UNSAFE` is kept only for compatibility but has no effect.
  - Some OrientDB-specific functions may not be available.
- **New SQL extensions**: ArcadeDB adds SQL commands not found in OrientDB, such as `ALIGN DATABASE`, `IMPORT DATABASE` with multiple format support, and enhanced `MATCH` syntax.

7.4.7. API

- **No ODocument/OVertex/OEdge**: ArcadeDB has its own API with `Document`, `Vertex`, and `Edge` interfaces that are similar in concept but different in implementation.
- **No ODatabaseDocument**: Use `Database` or `DatabaseFactory` in ArcadeDB.

- **No OClass**: Use `DocumentType`, `VertexType`, and `EdgeType` instead.
- **Multi-model API**: ArcadeDB provides a unified multi-model API from the start, supporting Document, Graph, Key/Value, Search Engine, Time Series, Vector, and Geospatial models.

7.4.8. Connectivity

- **HTTP/JSON API**: ArcadeDB provides a REST API that is different from OrientDB's REST API. The endpoints and JSON format are not compatible.
- **Postgres wire protocol**: ArcadeDB supports the PostgreSQL wire protocol, allowing connections from any PostgreSQL-compatible client. OrientDB did not offer this.
- **MongoDB wire protocol**: ArcadeDB supports the MongoDB wire protocol natively.
- **Redis wire protocol**: ArcadeDB supports a subset of the Redis protocol.
- **Bolt protocol**: ArcadeDB supports the Neo4j Bolt protocol, enabling connections from Neo4j-compatible clients and drivers.
- **gRPC protocol**: ArcadeDB supports gRPC for high-performance client-server communication.
- **No binary protocol**: ArcadeDB does not use a custom binary protocol like OrientDB's. Instead, it relies on HTTP/JSON and standard database wire protocols (PostgreSQL, MongoDB, Redis, Bolt, gRPC).
- **Gremlin**: ArcadeDB supports Apache TinkerPop Gremlin, similar to OrientDB, but through an embedded Gremlin engine rather than as a TinkerPop server.

7.4.9. Migration

To migrate from OrientDB to ArcadeDB:

1. **Export from OrientDB**: Export your OrientDB database using the OrientDB console `EXPORT DATABASE` command in JSON format.
2. **Import into ArcadeDB**: Use the ArcadeDB OrientDB Importer or the `IMPORT DATABASE` SQL command:

   ```
   IMPORT DATABASE url=file:///path/to/orientdb-export.json.gz
   ```

3. **Update application code**: Adapt your application to use ArcadeDB's API, drivers, and connection protocols.
4. **Test thoroughly**: Verify that queries, data integrity, and application logic work correctly with ArcadeDB.

For more details on importing, see the OrientDB Importer section.

7.5. Neo4j Importer

Migrating from Neo4j? The ArcadeDB Academy offers a **free** course on Neo4j-to-ArcadeDB migration with hands-on exercises and a certification at the end.

ArcadeDB is able to import a database exported from Neo4j in JSONL format (one json per line).

To export a Neo4j database follow the instructions in Export in JSON. The resulting file contains one json per line.

7.5.1. Performance

The Neo4j importer uses the high-performance `GraphBatch` API internally:

- **Vertices** are created with pre-allocated edge segments, eliminating lazy allocation during edge creation.
- **Edges** are buffered in flat primitive arrays and flushed sorted by source vertex, converting random I/O into sequential I/O. Light edges (no edge record on disk) are used when an edge has no properties.
- **WAL is disabled** during import for maximum throughput.
- **ID mapping** uses a primitive `long[]`-based hash map when Neo4j IDs are numeric (the common case with APOC exports), using only ~24 bytes per vertex. For any other ID, the importer automatically falls back to a standard `HashMap`. This makes it possible to import databases with hundreds of millions of vertices within a few gigabytes of heap.

7.5.2. Multi-label handling

Neo4j supports multiple labels per node, while in ArcadeDB a node (vertex) must have only one type. The Neo4j importer will simulate multiple labels by creating new types with the following name: `<label1>[~<labelN>]*`. Example:

```
{"type":"node","id":"1","labels":["User", "Administrator"],"properties":{"name":"Jim"
,"age":42}}
```

This vertex will be created in ArcadeDB with type "Administrator~User" (the labels are always sorted alphabetically) that extends both "Administrator" and "User" types.

In this way you can use the polymorphism of ArcadeDB to retrieve all the nodes of type "User" and the record of User and all its subtypes will be returned.

7.5.3. Importing via SQL

To import a database use the Import Database command from API, Studio or Console. Below you can find an example of importing the Neo4j's PanamaPapers database by using ArcadeDB Console.

```
> CREATE DATABASE PanamaPapers
{PanamaPapers}> IMPORT DATABASE file:///temp/panama-papers-neo4j.jsonl
```

```
ArcadeDB 26.11.1 - Neo4j Importer
Importing Neo4j database from file 'panama-papers-neo4j.jsonl' to
'databases/PanamaPapers'
```

```
- Creation of the schema: types, properties and indexes
- Creation of vertices started
- Creation of vertices completed: created 3 vertices (0 vertices/sec elapsed=0 secs)
- ID mapping mode: numeric (primitive long[])
- Creation of edges started: creating edges between vertices
- Creation of edges completed: created 1 edges, (0 edges/sec elapsed=0 secs)
******************************************************************************
*************
Import of Neo4j database completed in 0 secs with 0 errors, 0 warnings and 0 skipped
edges.
```

A relationship whose start or end node is not in the file, or that has no label, is not created and is counted as a skipped edge, the `skippedEdges` counter of the import report. *(Before v26.12.1, `skippedEdges` counted every relationship in the file, created or not, and the dropped ones were counted only under `warnings`.)*

7.5.4. Importing via command line

The Neo4j importer can also be used directly from the command line:

```
java -cp lib/* com.arcadedb.integration.importer.Neo4jImporter -i <input-file> -d
<database-path> [options]
```

Options:

Option	Default	Description
`-i <file>`		Path to the Neo4j JSONL export file (required)
`-d <path>`		Path where the ArcadeDB database will be created (required)
`-o`	false	Overwrite the database if it already exists
`-b <size>`	10,000	Number of records per transaction batch
`-decimalType <type>`	DECIMAL	Type for decimal values: FLOAT, DOUBLE, or DECIMAL
`-bucketBits <n>`	10	Bits allocated for bucket IDs in the internal RID packing. The default supports up to 1,023 buckets, which is sufficient for most databases. Increase this value only if you have a very large number of types and buckets (e.g. `-bucketBits 16` supports up to 65,535 buckets)

Example:

```
java -cp lib/* com.arcadedb.integration.importer.Neo4jImporter \
  -i /data/neo4j-export.jsonl -d /data/arcadedb/mydb -o -decimalType double
```

7.5.5. Memory considerations

For large imports (hundreds of millions of vertices), the main memory consumer is the ID mapping table that translates Neo4j node IDs to ArcadeDB record IDs. The table size depends on the ID format:

ID format	Memory per vertex	Example: 100M vertices
Numeric (e.g. "0", "12345")	~24 bytes	~2.2 GB
String (e.g. "node-abc")	~140 bytes	~13 GB

Neo4j APOC exports use numeric IDs by default, so most imports will use the compact primitive map. An ID counts as numeric only when it is written the plain way a number is written: digits with an optional leading minus sign, and no leading zero (`"0"` itself is fine). IDs such as `"01"`, `"+1"` or `"-0"`, zero-padded business keys like zip codes, and numbers beyond the 64-bit range are treated as strings, so `"1"` and `"01"` stay two different nodes. If the importer encounters such an ID, it automatically migrates to the string-based map and logs a message:

```
- Neo4j ID that is not the canonical form of a number detected, switching to string-based ID mapping
```

(Before v26.12.1, IDs that spell the same number, such as `"1"` and `"01"`, shared one mapping entry: the relationships of one node were attached to the other or skipped as "vertex not found", while the import still reported `OK`.)

For very large imports, allocate enough heap memory. For example, to import a database with 500M vertices using numeric IDs, you would need approximately 12 GB for the ID mapping table alone, plus memory for ArcadeDB's internal buffers. A setting of `-Xmx24G` or more is recommended.

7.5.6. Differences with Neo4j

ArcadeDB is fully compatible with Neo4j at the wire-protocol and query-language level — your existing applications can connect to ArcadeDB by changing the connection URL alone — but the engine underneath is multi-model, embeddable, and Apache 2.0 licensed. This page summarises what carries over, what is new, and how to migrate.

7.5.7. What's compatible

ArcadeDB ships several drop-in compatibility layers so existing Neo4j codebases keep working with minimal changes:

- **OpenCypher** — ArcadeDB implements OpenCypher (97.8% TCK pass rate) on every supported data model. Most Cypher queries written against Neo4j run unmodified.
- **Bolt protocol** — ArcadeDB exposes a Bolt server compatible with Bolt v3.0, v4.0, and v4.4. Official Neo4j drivers (Java, Python, JavaScript, .NET, Go, Ruby) connect by switching the URL

and credentials; existing application code stays untouched.

- **APOC** — A subset of Neo4j's APOC procedures is available through ArcadeDB's Extended Functions.

7.5.8. Where ArcadeDB goes further

Beyond Neo4j compatibility, ArcadeDB adds capabilities that typically require multiple databases on the Neo4j stack:

- **Multi-model** — graph, document, key/value, search, time-series, vector, and geospatial data live in a single engine and a single transaction. Neo4j is graph-only.
- **Multi-language** — beyond Cypher, ArcadeDB speaks SQL, Gremlin, GraphQL, MongoDB Query Language, and Redis commands.
- **Native vector search** — built-in JVector (DiskANN + HNSW with SIMD acceleration). No external vector database required.
- **Built-in full-text search** with fuzzy matching, integrated with the query languages.
- **High Availability and replication** are part of the open-source distribution. Neo4j requires the (paid) Enterprise edition for HA.
- **Embeddable** — ArcadeDB runs inside a JVM application with a few-megabyte footprint (as low as 16 MB heap). Neo4j Embedded is Enterprise-only.
- **Apache 2.0 licence** — free for commercial use, no copyleft. Neo4j Community is GPL, which forces distributed applications to publish their source.

7.5.9. Performance comparison

The LDBC Graphalytics benchmark shows the following timings on identical workloads (lower is better):

Algorithm	ArcadeDB	Neo4j
PageRank	**0.48 s**	11.15 s
Weakly Connected Components (WCC)	**0.30 s**	0.75 s
Breadth-First Search (BFS)	**0.13 s**	1.91 s
Local Clustering Coefficient (LCC)	**27.41 s**	45.78 s
Single-Source Shortest Path (SSSP)	**3.53 s**	not available
Community Detection (CDLP)	**3.67 s**	6.43 s

ArcadeDB's Graph OLAP engine delivers up to 400× faster analytics than Neo4j on the same hardware.

7.5.10. Connecting an existing Neo4j application

The quickest path is the Bolt server: keep using Neo4j's official drivers and only change the connection URL and credentials.

```
// Existing Neo4j code works unchanged when pointed at ArcadeDB's Bolt port.
Driver driver = GraphDatabase.driver(
    "bolt://localhost:7687",
    AuthTokens.basic("root", "arcadedb_password"),
    Config.builder().withoutEncryption().build());

try (Session session = driver.session(SessionConfig.forDatabase("mydb"))) {
  Result rs = session.run(
      "MATCH (p:Person)-[:KNOWS]->(f) WHERE p.name = $n RETURN f.name AS name",
      Map.of("n", "Alice"));
  rs.list().forEach(r -> System.out.println(r.get("name").asString()));
}
```

See Neo4j Bolt Protocol Plugin for server configuration and supported clients.

7.5.11. Importing a Neo4j database

ArcadeDB ships a Neo4j importer that reads APOC's JSONL export format and rebuilds the graph on top of ArcadeDB's storage.

1. Export the source database from Neo4j using APOC:

   ```
   CALL apoc.export.json.all("neo4j-export.jsonl", {})
   ```

   APOC writes one JSON object per line — vertices first, then relationships:

   ```
   {"type":"node","id":"0","labels":["User"],"properties":{"name":"Adam","age":42}}
   {"type":"node","id":"1","labels":["User"],"properties":{"name":"Jim","age":42}}
   {"type":"relationship","label":"KNOWS","properties":{"since":1993},
    "start":{"id":"0","labels":["User"]},"end":{"id":"1","labels":["User"]}}
   ```

2. Import into ArcadeDB through the console:

   ```
   > CREATE DATABASE MyDatabase
   {MyDatabase}> IMPORT DATABASE file:///path/to/neo4j-export.jsonl
   ```

   or programmatically through the Java API:

   ```
   Neo4jImporter importer = new Neo4jImporter(
       "-i", "/path/to/neo4j-export.jsonl",
       "-d", "./databases/MyDatabase",
       "-o");                                // overwrite if exists
   importer.run();
   ```

The importer runs three passes: schema reconciliation, vertex creation, and edge creation.

7.5.12. Multi-label nodes

Neo4j allows a single node to carry several labels (for example `[User, Administrator]`). ArcadeDB models the same idea with type inheritance: the importer creates a synthetic type `Administrator~User` (labels sorted alphabetically and joined with `~`) that extends both `Administrator` and `User`. Polymorphic queries work as expected — `SELECT FROM User` returns ordinary `User` vertices and every `Administrator~User` vertex.

7.5.13. Schema and constraints

Neo4j's `CREATE CONSTRAINT … REQUIRE … IS UNIQUE` translates to a small block of ArcadeDB SQL DDL:

```
-- Neo4j
-- CREATE CONSTRAINT FOR (p:Person) REQUIRE p.id IS UNIQUE

CREATE VERTEX TYPE Person
CREATE EDGE TYPE KNOWS
CREATE PROPERTY Person.id LONG
CREATE INDEX ON Person (id) UNIQUE
```

7.5.14. Embedding ArcadeDB

The native Java API skips the Bolt round-trip entirely and gives you full multi-model access in-process. This mode is exclusive to Apache 2.0 — no Enterprise licence required.

```
Database database = new DatabaseFactory("./databases/mydb").open();

database.transaction(() -> {
  MutableVertex alice = database.newVertex("Person");
  alice.set("name", "Alice");
  alice.set("born", 1985);
  alice.save();

  MutableVertex bob = database.newVertex("Person");
  bob.set("name", "Bob");
  bob.set("born", 1990);
  bob.save();

  alice.newEdge("KNOWS", bob, "since", 2015);
});

// Native graph traversal — no string queries, no parser overhead.
Vertex alice = database.lookupByKey("Person", "name", "Alice").next().asVertex();
for (Vertex friend : alice.getVertices(Vertex.DIRECTION.OUT, "KNOWS"))
  System.out.println(friend.getString("name"));
```

For sustained inserts, use the asynchronous API to fan operations across all available threads:

```
database.async().onError(Throwable::printStackTrace);
for (int i = 0; i < 1_000_000; i++) {
  MutableVertex v = database.newVertex("Person");
  v.set("id", i);
  v.set("name", "Person_" + i);
  database.async().createRecord(v, null);
}
```

7.5.15. Further reading

- Cypher in ArcadeDB — language overview and compatibility notes.
- Cypher Compatibility — the exact subset of OpenCypher TCK ArcadeDB passes.
- Bolt Protocol — wire-protocol setup for Neo4j drivers.
- Neo4j Importer — full reference for the JSONL importer used above.
- Graph OLAP Engine — the analytics engine behind the benchmark numbers.

Chapter 8. Self-Managed Deployment

8.1. Installations from Binaries

ArcadeDB is released for Java 21 or later. A Java 17 backport is built from the separate `java17` branch, usually some time after the main release. On Java 25 or later, `bin/server.sh` also turns on compact object headers (`-XX:+UseCompactObjectHeaders`), which shrinks every object header; set `ARCADEDB_OPTS_HEADERS` to an empty value to opt out.

Packages are available through the GitHub releases page or Maven Central. The GitHub releases page also carries the experimental native executables, which need no Java at all: see how-to/operations/native-image.pdf.

Java 17 packages, with a `-java17` version suffix, are available through the GitHub packages page.

Download the package suitable for your platform and follow the instructions below.

1. Unpack `tar -xzf arcadedb-26.11.1.tar.gz`
    - Change into directory: `cd arcadedb-26.11.1`
2. Launch server
    - Linux / MacOS: `bin/server.sh`
    - Windows: `bin\server.bat`
3. Exit server via **CTRL+C**
4. Interact with server
    - Studio: `http://localhost:2480`
    - Console:
        - Linux / MacOS: `bin/console.sh`
        - Windows: `bin\console.bat`
5. Exit server via **CTRL+C**

8.1.1. Binaries

	Linux / Mac	Windows
Server	`bin/server.sh`	`bin\server.bat`
Console	`bin/console.sh`	`bin\console.bat`

8.1.2. macOS via Homebrew

ArcadeDB is available as an official Homebrew formula. Install it with:

```
brew install arcadedb
```

After installation the following commands are available on your `PATH`:

```
arcadedb-server
arcadedb-console
```

Use these instead of `bin/server.sh` and `bin/console.sh` mentioned above.

On the first launch, set the `root` password by passing it on the command line:

```
arcadedb-server -Darcadedb.server.rootPassword=<your-password>
```

Replace `<your-password>` with a strong password. The server stores the (hashed) credential under its configuration directory, so subsequent launches can be started with `arcadedb-server` alone.

To upgrade to the latest release:

```
brew update
brew upgrade arcadedb
```

8.1.3. Manual installation on macOS

If you prefer not to use Homebrew:

1. Download the latest release from https://github.com/ArcadeData/arcadedb/releases
2. Extract the archive to your preferred location (e.g., `/usr/local/arcadedb`)
3. Add the `bin` directory to your `PATH`:

```
echo 'export PATH="/usr/local/arcadedb/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

8.1.4. Windows via Scoop

Instead of using manual install you can use Scoop installer, instructions are available on their project website.

```
scoop bucket add extras
scoop install arcadedb
```

This downloads and installs ArcadeDB on your box and makes following two commands available:

```
arcadedb-console
arcadedb-server
```

You should use these instead of `bin\console.bat` and `bin\server.bat` mentioned above.

8.1.5. Custom Package Builder

The ArcadeDB Modular Distribution Builder (`arcadedb-builder.sh`) allows you to create custom ArcadeDB packages containing only the modules you need. This results in smaller distributions, reduced dependencies, and simplified deployments.

8.1.6. Prerequisites

The builder requires the following tools:

- `curl` or `wget` - for downloading files
- `tar` - for extracting and creating archives
- `unzip` and `zip` - for creating zip archives
- `sha256sum` or `shasum` - for checksum verification
- `docker` (optional) - for Docker image generation

8.1.7. Quick Start

Run directly with curl (one-liner):

```
curl -fsSL https://github.com/ArcadeData/arcadedb/releases/download/26.11.1/arcadedb-builder.sh | bash -s -- --version=26.11.1 --modules=gremlin,studio
```

Download and run interactively:

```
curl -fsSLO https://github.com/ArcadeData/arcadedb/releases/download/26.11.1/arcadedb-builder.sh
chmod +x arcadedb-builder.sh
./arcadedb-builder.sh
```

Preview without building (dry run):

```
./arcadedb-builder.sh --version=26.11.1 --modules=gremlin,studio --dry-run
```

8.1.8. Available Modules

Core Modules (always included)

- `engine` - Database engine
- `server` - HTTP/REST API, clustering
- `network` - Network communication

Optional Modules

Module	Description
`console`	Interactive database console
`studio`	Web-based administration interface
`gremlin`	Apache Tinkerpop Gremlin support
`postgresw`	PostgreSQL wire protocol compatibility
`mongodbw`	MongoDB wire protocol compatibility
`redisw`	Redis wire protocol compatibility
`grpcw`	gRPC wire protocol support
`graphql`	GraphQL API support
`metrics`	Prometheus metrics integration
`mcp`	Model Context Protocol server for LLM clients. See MCP Server.

Omitting `mcp` produces a distribution that registers no `/api/v1/mcp` endpoint at all.

8.1.9. Usage Examples

Minimal build (PostgreSQL protocol only):

```
./arcadedb-builder.sh --version={revnumber} --modules=postgresw
```

Development build:

```
./arcadedb-builder.sh \
  --version=26.11.1 \
  --modules=console,gremlin,studio \
  --output-name=arcadedb-dev
```

Production build (no Studio):

```
./arcadedb-builder.sh \
  --version=26.11.1 \
  --modules=postgresw,metrics \
  --output-name=arcadedb-prod
```

CI/CD build:

```
./arcadedb-builder.sh \
  --version=26.11.1 \
  --modules=gremlin,studio \
  --quiet \
```

```
  --skip-docker \
  --output-dir=/tmp/builds
```

8.1.10. Command-Line Options

Option	Description
`--version=VERSION`	ArcadeDB version to build (required for non-interactive mode)
`--modules=MODULES`	Comma-separated list of modules
`--output-name=NAME`	Custom name for distribution
`--output-dir=DIR`	Output directory (default: current directory)
`--local-repo[=PATH]`	Use local Maven repository or custom JAR directory
`--local-base=FILE`	Use local base distribution file
`--docker-tag=TAG`	Build Docker image with specified tag
`--skip-docker`	Skip Docker image build
`--dockerfile-only`	Only generate Dockerfile, don't build image
`--dry-run`	Show what would be done without executing
`-v, --verbose`	Enable verbose output
`-q, --quiet`	Suppress non-error output
`-h, --help`	Show help message

8.1.11. Output Files

The builder creates:

- `{output-name}.zip` - Zip archive
- `{output-name}.tar.gz` - Compressed tarball
- Docker image with tag `{docker-tag}` (if not skipped)

For more detailed documentation, see the Builder README and Modular Builder Guide.

8.2. Installation with Docker

ArcadeDB is published as a Docker image on Docker Hub: arcadedata/arcadedb. The image runs on Java 25 (`eclipse-temurin:25-alpine`) and is built for `linux/amd64` and `linux/arm64`.

To pull a specific release:

```
docker pull arcadedata/arcadedb:26.11.1
```

The JVM image is published under these tags:

Tag	Content
`<version>` (for example `26.11.1`)	The release with that version number.
`latest`	The most recent build of the `main` branch, pushed on every merge. It is a development snapshot, not the latest release, so pin a version tag in production.
`<next-version>-SNAPSHOT`	The same development build, tagged with the version under development.

A Java 17 image, built from the separate `java17` branch, is published on GitHub Container Registry with a `-java17` tag suffix, usually some time after the main release:

```
docker pull ghcr.io/arcadedata/arcadedb:26.11.1-java17
```

An experimental GraalVM native image of the server is also published, under the `-native` tags (`<version>-native` for a release, `latest-native` for the nightly build of `main`). It starts faster and needs less memory, but leaves out Gremlin and has other limitations: see how-to/operations/native-image.pdf. It also needs a recent CPU (`x86-64-v3` on amd64, `armv8-a` on arm64) and exits at startup on an older one: see how-to/operations/native-image.pdf.

To run the container, configure it and persist its data, see how-to/operations/docker.pdf.

8.3. Native Image (Experimental)

ArcadeDB is also distributed as a native executable of the server, compiled ahead of time with GraalVM Native Image. The native server needs no Java runtime, starts in a fraction of a second and uses far less memory than the same server on a JVM, which suits containers, serverless-style deployments and quick local testing.

The native image is an **experimental add-on**. It does not replace the standard JVM distribution described in how-to/operations/binaries.pdf and how-to/operations/install-docker.pdf, which remains the default, fully supported way to run ArcadeDB and the one the rest of this manual assumes. Use the native image when fast startup and a small memory footprint matter more than having every module available, and treat it as best-effort outside Linux.

8.3.1. Getting the Native Image

Docker images. The native server is published on Docker Hub in the same `arcadedata/arcadedb` repository as the JVM image, under its own tags:

Tag	Content
`<version>-native` (for example `26.11.1-native`)	A release, as a multi-architecture image for `linux/amd64` and `linux/arm64`.

Tag	Content
`<version>-native-amd64`, `<version>-native-arm64`	The same release for one architecture only.
`latest-native`	The nightly build of the `main` branch, a development snapshot. Pin a version tag for anything that matters.
`<next-version>-SNAPSHOT-native`	The same nightly build, tagged with the version under development.

The nightly tags also exist per architecture (`latest-native-amd64`, `latest-native-arm64` and so on). The two architectures use different base images: the `linux/amd64` binary is fully static (musl libc) and runs on an empty `scratch` image, while the `linux/arm64` binary links glibc dynamically and runs on `gcr.io/distroless/base-debian12`.

Release archives. Each GitHub release carries the native executable for several platforms, each with a `.sha256` checksum file:

Platform	Asset	Status
Linux x86-64	`arcadedb-<version>-linux-x86_64.tar.gz`	Required for every release. Fully static, runs on any Linux.
Linux ARM64	`arcadedb-<version>-linux-aarch_64.tar.gz`	Required for every release. Needs glibc on the host.
Windows x86-64	`arcadedb-<version>-windows-x86_64.zip`	Best-effort: missing from a release whose Windows build failed.
macOS ARM64 (Apple Silicon)	`arcadedb-<version>-osx-aarch_64.tar.gz`	Best-effort: missing from a release whose macOS build failed.

The archive contains a single executable with the same name as the archive, minus the extension (`.exe` on Windows). Studio and the configuration templates are embedded in it.

8.3.2. CPU Requirements

A native binary is compiled for one CPU baseline and starts only on a CPU that has every feature of it. `native/pom.xml` states the baseline explicitly through the `native.march` property, which a profile sets per build architecture, instead of inheriting the builder's default. The published images and release archives use these baselines:

Architecture	Baseline	Required CPU features	Excludes, for example
amd64 (Linux, Windows)	`x86-64-v3`	AVX, AVX2, BMI1, BMI2, FMA, F16C, LZCNT, MOVBE	Intel before Haswell (2013), AMD before Excavator, Atom, Celeron and Pentium parts without AVX, and VMs whose CPU model hides AVX2 (the QEMU default `qemu64`, the Proxmox defaults `x86-64-v2-AES` and `kvm64`)
arm64 (Linux, macOS Apple Silicon)	`armv8-a`	ASIMD (the ARMv8.0 baseline)	None: runs on every 64-bit ARM core, including the Cortex-A53 and A72 (Raspberry Pi 3 and 4) and Graviton1

`x86-64-v3` is also GraalVM 25's own default on amd64. On arm64 the builder's own default is `armv8.1-a` (CRC32 and LSE atomics), and `armv8-a` is a deliberate choice below it, so that ARMv8.0 cores keep working. *(Since v26.11.1: the arm64 baseline was briefly recorded as `armv8.1-a`, which excluded those cores, and is `armv8-a` again.)* On an older CPU the binary exits as soon as it starts, with:

```
The current machine does not support all of the following CPU features that are
required by the image: [...].
Please rebuild the executable with an appropriate setting of the -march option.
```

To check a Linux amd64 host, run `/lib64/ld-linux-x86-64.so.2 --help`: on glibc 2.33 and later it lists `x86-64-v3 (supported, searched)` when the CPU qualifies. Otherwise look for the `avx2`, `bmi2` and `fma` flags in `/proc/cpuinfo`. In a VM, expose the host CPU (`-cpu host` in QEMU, CPU type `host` in Proxmox) or a model that supports `x86-64-v3`.

If the CPU does not qualify, either:

- download the `linux/amd64` compatibility archive of the release, `arcadedb-<version>-linux-x86_64-compat.tar.gz`, built with `-march=compatibility` (the default archive is `arcadedb-<version>-linux-x86_64.tar.gz`). It starts on any x86-64 CPU, including pre-Haswell hardware, QEMU's `qemu64` and Proxmox's `x86-64-v2-AES`. It is an archive only: there is no Docker image for it, and the published images keep the baselines in the table above. To run it in a container, build the image yourself with `./native/scripts/build-native-docker.sh --march compatibility`,
- run the JVM image (`arcadedata/arcadedb:<version>`) or the JVM distribution, whose JIT compiles for the CPU it runs on, or
- build a native binary for a lower baseline with `-Dnative.march=compatibility`, which runs on any x86-64 or ARMv8 CPU (see Building the Native Image Yourself).

8.3.3. Running the Native Server

Run the executable directly, passing settings as `-D` arguments exactly as you would pass them to the JVM:

```
tar -xzf arcadedb-26.11.1-linux-x86_64.tar.gz
./arcadedb-26.11.1-linux-x86_64 -Darcadedb.server.rootPassword=<your-password>
```

The server uses the current directory as its root path, so it creates `databases/` and its other directories there unless you set `arcadedb.server.rootPath`. Studio is then available at http://localhost:2480.

To run the Docker image:

```
mkdir -p databases && sudo chown 1000:1000 databases   # only for a bind mount, see
below
docker run --rm -p 2480:2480 -p 5432:5432 \
  -v "$(pwd)/databases:/home/arcadedb/databases" \
  arcadedata/arcadedb:26.11.1-native \
  -Darcadedb.server.rootPassword=<your-password>
```

The native images differ from the JVM image in a few ways, because neither base image contains a shell:

- **Settings are trailing arguments, not environment variables.** The JVM image's `bin/server.sh` expands `ARCADEDB_SETTINGS` and `JAVA_OPTS` onto the command line. The native image starts the binary directly, so pass `-Darcadedb.server.rootPassword=⋯` and any other setting as arguments after the image name, as in the example above.
- **User.** *(Since v26.11.1)* Both native images run as UID/GID `1000:1000`, the same user as the JVM image, so no `--user` is needed and a volume written by any of the three images can be mounted by the others. Anonymous and named volumes start owned by `1000:1000` and are writable out of the box. A bind mount keeps the ownership of the host directory, so every bind-mounted path the server writes (`databases`, but also `config` and `log`) must be writable by UID 1000, otherwise the server fails to start with an `AccessDeniedException`. To run as another UID, pass `--user <uid>:<gid>` and give that UID ownership of every volume and bind mount, including `config` and `log`.
- **Upgrading from an earlier native image.** Before v26.11.1 the `linux/amd64` image ran as root and the `linux/arm64` image as the distroless `nonroot` user (65532), so the volumes they wrote are owned by those UIDs. Re-own each one once before starting the new image:

  ```
  docker run --rm --user 0 -v <volume>:/data busybox chown -R 1000:1000 /data
  ```

  The same command makes those volumes usable by the JVM image.

- **Ports.** The image exposes `2480` (HTTP API and Studio), `2434` (Raft replication), `5432` (PostgreSQL), `6379` (Redis), `27017` (MongoDB), `7687` (Bolt) and `50051` (gRPC). There is no port `8182`, because Gremlin is not included. Publish the ports of the protocol plugins you enable in `arcadedb.server.plugins`.
- **Volumes.** `config`, `databases`, `backups`, `replication` and `log` under `/home/arcadedb`, as in the JVM image.

8.3.4. What the Native Image Includes

- The SQL and OpenCypher query engines
- The HTTP API and Studio
- The PostgreSQL, Redis, MongoDB, Neo4j Bolt and gRPC wire protocols
- JavaScript scripting through GraalJS (see the security caveat below)
- The `console` and `graphql` modules
- *(Since v26.10.2)* Raft high availability, enabled the same way as on the JVM (`-Darcadedb.ha.enabled=true` and `-Darcadedb.ha.serverList=…`): see how-to/operations/ha.pdf
- *(Since v26.10.2)* Metrics: the Prometheus endpoint (`/prometheus`, enabled by adding `com.arcadedb.metrics.prometheus.PrometheusMetricsPlugin` to `arcadedb.server.plugins`) and OTLP metrics push (`-Darcadedb.serverMetrics.otlp.enabled=true`)
- *(Since v26.10.2)* Tracing: OpenTelemetry spans over OTLP (`-Darcadedb.serverMetrics.tracing.enabled=true`)

A plugin named in `arcadedb.server.plugins` that is not in the image is skipped without a log line. If a setting appears to do nothing, check that its module is part of the native image before looking for a configuration mistake.

8.3.5. Limitations Compared to the JVM Distribution

- **No Gremlin.** Apache TinkerPop is not part of the native build. If you need Gremlin, use the JVM distribution.
- **No SIMD vector math.** *(Since v26.10.2)* A native image does not compile the Java Vector API into SIMD instructions, so ArcadeDB switches vector index builds, vector functions, Graph OLAP and time series to their scalar implementations when it detects a native image. Results are identical, but vector-heavy workloads run slower than on the JVM, where the Vector API is compiled to SIMD. To force a different JVector provider anyway, set the `jvector.vectorization_provider` system property.
- **Thai full-text analyzer.** `org.apache.lucene.analysis.th.ThaiAnalyzer` fails with "This JRE does not have support for Thai segmentation", because the JDK's Thai dictionary is not available in the image. The other analyzers work.
- **JavaScript is not hardened for untrusted callers.** Embedding GraalJS required relaxing a Truffle safety check for a few standard builtins (`Atomics.*`, `SharedArrayBuffer`, `TypedArray.prototype.set`), and any authenticated user who can run `js` commands can reach them. Do not expose the native image's JavaScript surface to untrusted users in a shared or multi-tenant deployment.
- **Certificate trust store.** The image carries a snapshot of the build machine's JDK `cacerts` trust store, taken at build time. The CA bundle copied into the Docker images serves only code paths that read the operating system's bundle explicitly; it does not change the trust anchors baked into the binary.
- **Size.** The executable is large, around 907 MiB on macOS ARM64, because every classpath resource, including the Studio assets, is embedded in it.

8.3.6. Building the Native Image Yourself

Most users should download a published binary or image. Two scripts in the ArcadeDB source tree build one. `build-native.sh` needs GraalVM Community Edition 25.0.2 installed (the exact version the build pins), with both `JAVA_HOME` and `GRAALVM_HOME` pointing at the GraalVM home. `build-native-docker.sh` needs only Docker: it fetches the pinned GraalVM into a throwaway builder image.

```
./native/scripts/build-native.sh                # binary for the current platform
./native/scripts/build-native-docker.sh         # Linux Docker image for the current
architecture
```

By default the build targets the baseline of the published binaries (see CPU Requirements). To build for an older CPU, lower it: `compatibility` runs on any x86-64 or ARMv8 CPU, and `x86-64-v2` (amd64 only) needs SSE4.2 and POPCNT but no AVX. A lower baseline gives up the AVX2, FMA and BMI2 (or LSE) code of the default build.

```
./native/scripts/build-native.sh -- -Dnative.march=compatibility
./native/scripts/build-native-docker.sh --march compatibility
mvn -Pnative -pl native -am -DskipTests -Dnative.march=compatibility package
```

Do not distribute a binary built with `-march=native`: it requires every feature of the CPU that built it.

Native Image cannot cross-compile: each build produces a binary for the platform it runs on, and `build-native-docker.sh` builds the Linux binary inside a container so it also works on macOS. The build is memory hungry. `build-native-docker.sh` requires a Docker VM with about 16 GiB of memory, and for a local `build-native.sh` you can set the builder heap with `NATIVE_IMAGE_OPTIONS="-J-Xmx24g"` if the default runs out of memory. See `docs/native-image.md` in the ArcadeDB repository for the full build reference.

8.4. Running ArcadeDB on Kubernetes

ArcadeDB officially supports deployment on Kubernetes via the arcadedb-helm chart, published at `https://helm.arcadedb.com/`. The chart deploys ArcadeDB as a `StatefulSet` and, when `replicaCount` is greater than 1, automatically configures a Raft-based High Availability cluster. The reference for the configuration values below targets chart version **26.4.2**; confirm against your installation with `helm show values arcadedb/arcadedb`.

8.4.1. Prerequisites

- Kubernetes 1.27 or later
- Helm 3.16 or later
- `kubectl` configured for your target cluster
- A storage class that provides `ReadWriteOnce` volumes (the cluster defaults on EKS, GKE, AKS, and DigitalOcean all work)
- (Optional) `kind` 0.24 or later and Docker for the local quickstart below

8.4.2. Quickstart with kind

This walkthrough mirrors `ArcadeData/arcadedb-deployments/kubernetes` and produces a 3-node ArcadeDB cluster on a local kind cluster in roughly five minutes.

1. Create a local Kubernetes cluster:

```
kind create cluster --name arcadedb
```

2. Create the secret holding the `root` user password (replace `<password>`):

```
kubectl create secret generic arcadedb-credentials \
  --from-literal=rootPassword='<password>'
```

3. Add the ArcadeDB Helm repository and update:

```
helm repo add arcadedb https://helm.arcadedb.com/
helm repo update
```

4. Save the following overrides to `values.yaml`:

```
replicaCount: 3
image:
  tag: "26.4.2"
service:
  http:
    type: ClusterIP
arcadedb:
  credentials:
    rootPassword:
      secret:
        name: arcadedb-credentials
        key: rootPassword
```

5. Install the chart:

```
helm install my-arcadedb arcadedb/arcadedb -f values.yaml --wait
```

6. Forward the HTTP port to your laptop:

```
kubectl port-forward svc/my-arcadedb-http 2480:2480
```

7. Open http://localhost:2480 in a browser to reach Studio, or use `curl`:

```
curl -u root:<password> http://localhost:2480/api/v1/server
```

For a scripted version (with `start.sh`, `stop.sh`, and `test.sh`), see the reference deployment.

8.4.3. Installing the chart

For production deployments, follow the same broad shape as the quickstart but tailor `values.yaml` to your environment.

1. Add the Helm repository:

```
helm repo add arcadedb https://helm.arcadedb.com/
helm repo update
```

2. Create a `Secret` containing the `root` password. The chart references this secret by name — it does not store the password in the chart values:

```
kubectl create secret generic arcadedb-credentials \
  --from-literal=rootPassword='<password>'
```

3. Prepare a `values.yaml` override file that fits your environment. At a minimum, set `replicaCount`, `image.tag`, and the `arcadedb.credentials.rootPassword.secret` block. See the configuration reference for the full set of options including persistence, ingress, resources, autoscaling, and security context.
4. Install the chart:

```
helm install my-arcadedb arcadedb/arcadedb \
  --namespace arcadedb --create-namespace \
  --values values.yaml \
  --wait
```

5. Verify the deployment:

```
kubectl -n arcadedb get statefulset
kubectl -n arcadedb get pods
kubectl -n arcadedb get svc
```

The pods are named `my-arcadedb-0`, `my-arcadedb-1`, … and reach the `Ready` state once the readiness probe succeeds.

8.4.4. Configuration reference

The values exposed by the chart are grouped below. This reference targets chart version **26.4.2**; confirm the current set against your installation with:

```
helm show values arcadedb/arcadedb
```

The complete and authoritative source is the chart's `values.yaml`. The example snippets below show keys at the top level (the chart is installed directly as `arcadedb/arcadedb`); if you consume the chart as a **subchart** under a parent named `arcadedb`, wrap each snippet under a top-level `arcadedb:` key.

Image

Selects the container image, pull policy, and pull secrets. Override `image.tag` to pin to a specific ArcadeDB release; otherwise the chart's `appVersion` is used.

Key	Default	Description
`image.registry`	`arcadedata`	Container registry hosting the image
`image.repository`	`arcadedb`	Image repository name
`image.tag`	(chart `appVersion`)	Image tag; pin to a specific ArcadeDB version for reproducible deployments
`image.pullPolicy`	`IfNotPresent`	Kubernetes image pull policy
`imagePullSecrets`	`[]`	List of `Secret` references for private registries
`nameOverride`	`""`	Overrides the chart name used in resource names
`fullnameOverride`	`""`	Overrides the fully-qualified app name used in resource names

Example override:

```
image:
  tag: "26.4.2"
  pullPolicy: IfNotPresent
imagePullSecrets:
  - name: my-registry-secret
```

Replicas

`replicaCount` controls how many ArcadeDB pods the `StatefulSet` runs. A single replica (`1`) gives a standalone instance; three replicas is the smallest cluster that tolerates one node failure under Raft majority quorum. See High Availability on Kubernetes for the implications of running a multi-replica cluster.

Key	Default	Description
`replicaCount`	`1`	Number of ArcadeDB pods. Values greater than 1 enable Raft HA automatically; the minimum recommended HA size is 3.

Example override:

```
replicaCount: 3
```

Raising `replicaCount` on a running HA cluster is enough: the new pods join the existing cluster on their own, and the pods already running accept them without any change to `ha.serverList`. This relies on the headless service publishing the address of a pod that is not Ready yet (`publishNotReadyAddresses: true`, which the chart sets), because a pod becomes Ready only once it has joined.

Credentials

The chart never stores the root password in its own values; it reads it from a Kubernetes `Secret` at install time. If `arcadedb.credentials.rootPassword.secret.name` is `null`, a 32-character random password is auto-generated in a `Secret` named `arcadedb-credentials-secret` — note that GitOps tools such as ArgoCD regenerate this secret on every render, so supply your own secret for production. Pods that start without a resolvable secret crash-loop immediately; see Troubleshooting for the symptom and fix.

Key	Default	Description
`arcadedb.credentials.rootPassword.secret.name`	`null`	Name of an existing `Secret` containing the root password. `null` triggers auto-generation.
`arcadedb.credentials.rootPassword.secret.key`	`null`	Key within the named `Secret` whose value is the root password.

Create the secret first (see the Quickstart for the full command), then reference it in your `values.yaml`:

```
arcadedb:
  credentials:
    rootPassword:
      secret:
        name: arcadedb-credentials
        key: rootPassword
```

Database

These keys control where ArcadeDB stores databases inside the container, which databases to create at startup, additional JVM arguments, and arbitrary environment variables passed to the container process.

Key	Default	Description
`arcadedb.databaseDirectory`	`/home/arcadedb/databases`	Path inside the container where database files are stored. Must be the `persistence` mount path, or a subdirectory of it, if a PVC is used.

Key	Default	Description
`arcadedb.defaultDatabases`	`""`	Databases to create automatically at first startup. Use the form `Name[user:password]`, e.g. `Universe[admin:secret]`. Empty string means no databases are pre-created.
`arcadedb.extraCommands`	`[-Darcadedb.server.mode=production]`	List of additional JVM `-D` arguments appended to the ArcadeDB startup command.
`arcadedb.extraEnvironment`	`[]`	List of extra environment variables (in `name`/`value` map form) injected into the container.

Example override:

```
arcadedb:
  extraCommands:
    - -Darcadedb.server.mode=production
    - -Darcadedb.txWalFiles=2
  extraEnvironment:
    - name: ARCADEDB_OPTS_GC
      value: "-XX:+UseG1GC"
```

Select the collector with `ARCADEDB_OPTS_GC`, which replaces the image's ZGC. Passing it through `JAVA_TOOL_OPTIONS` or `JAVA_OPTS` adds a second collector to the command line, and the JVM refuses to start ("Multiple garbage collectors selected"). See JVM settings inside a fixed memory budget.

Plugins

ArcadeDB supports multiple wire protocols alongside its native HTTP API; each protocol is enabled by adding the corresponding key under `arcadedb.plugins`. The chart ships with named slots for `gremlin`, `postgres`, `mongo`, `redis`, and `prometheus`, plus support for any user-supplied key (e.g. `myPlugin`) for custom plugins. By default, `arcadedb.plugins` is an empty map (`{}`), meaning no wire-protocol plugins are activated.

Key	Default	Description
`arcadedb.plugins.gremlin.enabled`	(absent)	Enable the Apache TinkerPop Gremlin server plugin.
`arcadedb.plugins.gremlin.port`	(absent — see example)	Port the Gremlin plugin listens on; conventionally `8182`.
`arcadedb.plugins.postgres.enabled`	(absent)	Enable the PostgreSQL wire-protocol plugin.
`arcadedb.plugins.postgres.port`	(absent — see example)	Port the PostgreSQL plugin listens on; conventionally `5432`.

Key	Default	Description
`arcadedb.plugins.mongo.enabled`	(absent)	Enable the MongoDB wire-protocol plugin.
`arcadedb.plugins.mongo.port`	(absent — see example)	Port the MongoDB plugin listens on; conventionally `27017`.
`arcadedb.plugins.redis.enabled`	(absent)	Enable the Redis wire-protocol plugin.
`arcadedb.plugins.redis.port`	(absent — see example)	Port the Redis plugin listens on; conventionally `6379`.
`arcadedb.plugins.prometheus.enabled`	(absent)	Enable the Prometheus metrics endpoint.

Example override (enabling the PostgreSQL and MongoDB wire protocols):

```
arcadedb:
  plugins:
    postgres:
      enabled: true
      port: 5432
    mongo:
      enabled: true
      port: 27017
```

Service

The chart creates two Kubernetes `Service` objects for each ArcadeDB deployment. The **HTTP service** (named `<release>-http`) exposes port 2480 where Studio and all REST/HTTP clients connect; its type is configurable below. The **RPC service** is a headless `ClusterIP` service used only for Raft peer discovery between pods — it is internal-only and not directly controlled by these values.

Key	Default	Description
`service.http.type`	`ClusterIP`	Kubernetes service type for the HTTP service. Use `LoadBalancer` to obtain an external IP, or leave `ClusterIP` and configure Ingress for a single-entry-point setup.
`service.http.port`	`2480`	Port exposed by the HTTP service. Studio and all REST/HTTP clients connect to this port.
`service.rpc.port`	`2434`	Raft gRPC port used for HA peer communication. This port is exposed only on the headless internal service; do not expose it externally.

Example override:

```
service:
  http:
    type: LoadBalancer
```

```
    port: 2480
```

Ingress

Use an `Ingress` resource when you want a single external entry point with host-based routing and TLS termination at the edge controller, without assigning a cloud load-balancer IP per service. For simpler setups or clouds where a managed load-balancer IP is acceptable, `service.http.type: LoadBalancer` is the lower-configuration alternative.

Key	Default	Description
`ingress.enabled`	`false`	Set to `true` to create an `Ingress` resource for the HTTP service.
`ingress.className`	`""`	IngressClass name (e.g. `nginx`, `traefik`). Empty string uses the cluster-default class.
`ingress.annotations`	`{}`	Annotations added to the `Ingress` object. Commonly used to configure the ingress controller behavior (e.g. `cert-manager.io/cluster-issuer`).
`ingress.hosts`	(see description)	List of host rules. Each entry has a `host` string and a `paths` list (each with `path` and `pathType`). Default example uses host `chart-example.local` with path `/`.
`ingress.tls`	`[]`	TLS configuration. Each entry specifies a `secretName` (holding the TLS certificate) and the list of `hosts` it covers.

Example override (single-host ingress with TLS via cert-manager):

```yaml
ingress:
  enabled: true
  className: nginx
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
  hosts:
    - host: arcadedb.example.com
      paths:
        - path: /
          pathType: Prefix
  tls:
    - secretName: arcadedb-tls
      hosts:
        - arcadedb.example.com
```

Studio and the HTTP API authenticate every request, but do not expose them directly to the public internet without an extra layer. Place public access behind the ingress controller with TLS (as above), an OIDC/SSO authentication proxy, or a VPN, so the database is never the sole gatekeeper of a publicly reachable endpoint.

Persistence

ArcadeDB stores its database files under `arcadedb.databaseDirectory` (default `/home/arcadedb/databases`). The chart uses a `volumeClaimTemplates` block in the `StatefulSet`, so Kubernetes provisions one `PersistentVolumeClaim` per pod rather than sharing a single volume across replicas.

Only directories that contain a database (a `schema.json` file) are opened at startup. Other directories in the database directory, such as the `lost+found` folder of a freshly formatted ext4 volume, are skipped and left untouched.

PVCs created by the StatefulSet survive `helm uninstall` by design — Kubernetes preserves data volumes when a workload is deleted. To remove them along with the release, run:

```
kubectl delete pvc -l app.kubernetes.io/name=arcadedb -n <namespace>
```

Key	Default	Description
`persistence.enabled`	`true`	Enable PVC-backed storage. Set to `false` only for ephemeral development deployments; all data is lost when the pod restarts.
`persistence.size`	`8Gi`	Requested PVC capacity. Size the volume to hold your databases plus headroom for growth.
`persistence.accessMode`	`ReadWriteOnce`	PVC access mode. `ReadWriteOnce` is correct for the `StatefulSet` pattern (one pod per volume).
`persistence.storageClass`	`""`	StorageClass name. Empty string uses the cluster-default class (e.g. `standard` on GKE, `gp2` on EKS). Set explicitly to select a faster or differently-billed class.

Example override (50 Gi PVC on the `gp3` storage class on AWS EKS):

```
persistence:
  enabled: true
  size: 50Gi
  storageClass: gp3
```

Resources

Container resource requests and limits are deliberately left unset in the chart defaults (`resources: {}`), so you must supply them for any production deployment. Set `limits.memory` first: the container image derives the JVM heap from it (`-XX:MaxRAMPercentage=75`) unless you pin the heap yourself, so the limit is what decides how much memory ArcadeDB actually uses. Then size the memory request to the heap you expect, so the scheduler places the pod on a node with enough RAM.

Do **not** set a CPU limit on ArcadeDB pods. Kubernetes CPU throttling operates on a

> 100 ms accounting window; when a CPU limit is hit mid-GC cycle, the JVM thread is suspended until the next window, turning a millisecond collection into a multi-hundred-millisecond pause. Set a CPU **request** so the scheduler reserves the right amount of CPU for scheduling purposes, and rely on node-level CPU sharing — not a hard limit — to bound consumption. Memory limits are safe and recommended: the JVM respects its heap ceiling, and a hard memory limit prevents a runaway process from affecting other pods on the node.

Key	Default	Description
`resources.requests.cpu`	(absent)	CPU request used by the scheduler to place the pod. Recommended starting point: `500m`. No CPU limit should be set — see the note above.
`resources.requests.memory`	(absent)	Memory request. Should equal or slightly exceed the heap to guarantee the node has enough RAM at scheduling time.
`resources.limits.memory`	(absent)	Memory limit, and the value the heap is sized against unless you pin it: 75% of this becomes the heap, leaving the rest for off-heap buffers (e.g. `4Gi` gives a 3 Gi heap). Also prevents a single pod from consuming all node memory.

Example override (2 Gi heap, 4 Gi limit, CPU request only — no CPU limit):

```
resources:
  requests:
    cpu: 500m
    memory: 2Gi
  limits:
    memory: 4Gi
```

Probes

Kubernetes uses **liveness probes** to decide when to kill and restart a container, and **readiness probes** to decide when a pod is safe to receive traffic. Both probes matter for ArcadeDB: on a cold start the JVM and optional persistence rehydration can take several seconds, so tuning `initialDelaySeconds` prevents premature restarts before the server is ready.

The chart configures both probes to poll `GET /api/v1/ready` on the HTTP port (2480). Only `httpGet.path` and `httpGet.port` are set by default; all other sub-keys (`initialDelaySeconds`, `periodSeconds`, `timeoutSeconds`, `failureThreshold`, `successThreshold`) are absent and fall back to Kubernetes defaults (0 s delay, 10 s period, 1 s timeout, 3 failures, 1 success).

Key	Default	Description
`livenessProbe.httpGet.path`	`/api/v1/ready`	HTTP path polled by the liveness probe. Returns 200 when the server is alive.

Key	Default	Description
`livenessProbe.httpGet.port`	`http`	Named port (`http`, resolving to 2480) used by the liveness probe.
`livenessProbe.initialDelaySeconds`	(absent — Kubernetes default: `0`)	Seconds to wait after container start before the first liveness check. Increase for slow-start environments (cold JVM + persistence rehydration).
`livenessProbe.periodSeconds`	(absent — Kubernetes default: `10`)	How often (in seconds) to run the liveness probe.
`livenessProbe.timeoutSeconds`	(absent — Kubernetes default: `1`)	Seconds after which a probe attempt times out.
`livenessProbe.failureThreshold`	(absent — Kubernetes default: `3`)	Number of consecutive failures before the container is killed and restarted.
`livenessProbe.successThreshold`	(absent — Kubernetes default: `1`)	Consecutive successes required to mark the container as live again after a failure.
`readinessProbe.httpGet.path`	`/api/v1/ready`	HTTP path polled by the readiness probe. Returns 200 when the server is ready to accept traffic.
`readinessProbe.httpGet.port`	`http`	Named port (`http`, resolving to 2480) used by the readiness probe.
`readinessProbe.initialDelaySeconds`	(absent — Kubernetes default: `0`)	Seconds to wait after container start before the first readiness check. Increase when the server needs time to rehydrate databases from a PVC before accepting connections.
`readinessProbe.periodSeconds`	(absent — Kubernetes default: `10`)	How often (in seconds) to run the readiness probe.
`readinessProbe.timeoutSeconds`	(absent — Kubernetes default: `1`)	Seconds after which a probe attempt times out.
`readinessProbe.failureThreshold`	(absent — Kubernetes default: `3`)	Number of consecutive failures before the pod is removed from Service endpoints.
`readinessProbe.successThreshold`	(absent — Kubernetes default: `1`)	Consecutive successes required to add the pod back to Service endpoints.

Example override (bumping the initial delay for a slow-start environment with large persisted databases):

```yaml
livenessProbe:
  httpGet:
    path: /api/v1/health
    port: http
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 6
readinessProbe:
  httpGet:
    path: /api/v1/ready
    port: http
  initialDelaySeconds: 20
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 3
```

Prefer the dedicated liveness endpoint `/api/v1/health` for the liveness probe: it never reports a busy state, so a warming-up node is not restarted. Keep `/api/v1/ready` for the readiness probe. The chart's default value tables below reflect the chart's current defaults; override `livenessProbe.httpGet.path` to `/api/v1/health` as shown above. See Health probes.

On a High Availability cluster, set `arcadedb.server.readinessRequiresHA=true` (env `ARCADEDB_SERVER_READINESSREQUIRESHA=true`) to withhold traffic from a node until it has joined the Raft group and, once joined, until it is a member of the current Raft configuration and has caught up to within `arcadedb.server.readinessHAMaxLag` log entries (default `100`, env `ARCADEDB_SERVER_READINESSHAMAXLAG`) of the commit index. This keeps a restarted follower with a wiped or lagging log out of Service endpoints during a rolling update, so the write quorum is not dropped. The check also fails closed if the node's internal Raft engine is unhealthy, even while the server process itself is still running and responding.

This setting decides how strict the probe is about a node that is merely **behind**. It does not apply to a node whose replication has **stopped** - one that cannot write its replication log, or that hit an error it cannot apply. Such a node answers `503` whatever this setting says, because waiting for it would never help. See When a node has stopped for good for the two fields that name the cause.

Autoscaling

The chart supports Kubernetes `HorizontalPodAutoscaler` (HPA) to scale the number of ArcadeDB pods based on CPU utilization. Autoscaling is disabled by default; enable it only after verifying that your storage class supports `ReadWriteOnce` volumes on all potential target nodes, and that your HA configuration can tolerate dynamic replica counts.

When autoscaling is enabled in an HA configuration, ensure `minReplicas >= floor(maxReplicas / 2) + 1` so the cluster always retains a Raft majority. Falling

> below the quorum threshold causes writes to stall until enough peers rejoin. For example, with `maxReplicas: 5` the minimum safe `minReplicas` is `3`. See High Availability on Kubernetes for further guidance on Raft quorum sizing.
>
> The chart also pre-sizes the internal server list to `maxReplicas` for auto-join purposes; set `maxReplicas` conservatively to avoid advertising pod addresses that may never exist. Since v26.11.1 a pod that joins beyond the server list is routed to by Bolt and gRPC clients like the listed ones and reported by `GET /api/v1/cluster` all the same, so the pre-sized list is no longer what makes a scaled-up pod reachable for clients (see High Availability on Kubernetes).

Key	Default	Description
`autoscaling.enabled`	`false`	Set to `true` to create an `HorizontalPodAutoscaler` that scales the `StatefulSet` based on CPU utilization.
`autoscaling.minReplicas`	`1`	Minimum number of replicas the HPA will scale down to. In an HA cluster this must be at least `floor(maxReplicas / 2) + 1` to preserve Raft quorum.
`autoscaling.maxReplicas`	`5`	Maximum number of replicas the HPA will scale up to. Set conservatively — the chart pre-sizes the peer list to this value.
`autoscaling.targetCPUUtilizationPercentage`	`80`	Average CPU utilization target (as a percentage of the pod's CPU request) that triggers scale-out. Lower values scale out earlier; higher values pack more load per pod.

Example override (autoscaling with Raft-safe quorum bounds):

```
autoscaling:
  enabled: true
  minReplicas: 3
  maxReplicas: 5
  targetCPUUtilizationPercentage: 70
```

Security context

The chart sets a restrictive security posture by default. At the pod level, `podSecurityContext` ensures all processes run as a non-root user and that the filesystem group matches the `arcadedb` user (`1000`) created in the Docker image, so mounted volumes are writable without `chmod`. At the container level, `securityContext` pins the UID/GID to `1000`, blocks privilege escalation, and drops all Linux capabilities — the server needs none.

Key	Default	Description
`podSecurityContext.runAsNonRoot`	`true`	Require that the pod's entry-point process runs as a non-root UID. Kubernetes rejects the pod if the image's `USER` is `0`.

Key	Default	Description
`podSecurityContext.fsGroup`	`1000`	Supplemental GID applied to mounted volumes. Set to match the `arcadedb` user so PVC data directories are writable by the container process.
`securityContext.runAsUser`	`1000`	UID used by the container process. Matches the `arcadedb` user in the image.
`securityContext.runAsGroup`	`1000`	Primary GID used by the container process.
`securityContext.allowPrivilegeEscalation`	`false`	Prevents the container process from gaining more privileges than its parent (blocks `setuid` binaries, no `CAP_SYS_ADMIN`, etc.).
`securityContext.capabilities.drop`	`[ALL]`	Drops every Linux capability from the container. ArcadeDB requires none of the default capabilities.

Example override (adding `readOnlyRootFilesystem` for maximum hardening — ArcadeDB writes only to its data directory, which is a mounted volume):

```
podSecurityContext:
  runAsNonRoot: true
  fsGroup: 1000
securityContext:
  runAsUser: 1000
  runAsGroup: 1000
  allowPrivilegeEscalation: false
  capabilities:
    drop: [ALL]
  readOnlyRootFilesystem: true
```

`readOnlyRootFilesystem: true` is not set by the chart's default but is safe to add when persistence is enabled — ArcadeDB writes database files to the PVC mount path, not to the container root filesystem. If you run without persistence (`persistence.enabled: false`), omit this flag because the container writes databases to the root filesystem.

With `readOnlyRootFilesystem: true` you must also relocate the **log directory** to a writable mount, otherwise the server fails at startup with `java.nio.file.FileSystemException: ./log/arcadedb.log.0.lck: Read-only file system`. By default the server writes logs to `./log` (relative to the working directory), which is read-only under this hardening. Mount a writable volume (for example an `emptyDir`) and point the log directory at it by forwarding the `ARCADEDB_LOG_DIR` environment variable (it maps to the `arcadedb.server.logsDirectory` setting):

```
extraEnvironment:
  - name: ARCADEDB_LOG_DIR
    value: /writable/log
volumes:
```

```
  - name: arcadedb-log
    emptyDir: {}
volumeMounts:
  - name: arcadedb-log
    mountPath: /writable/log
```

When running an HA cluster with `readOnlyRootFilesystem: true`, also relocate the **Raft storage directory**. By default each node creates a dynamically-named `raft-storage-<nodeName>` folder under the server root path, which is both read-only under this hardening and awkward to mount in a `StatefulSet` because the name varies per pod. Set `arcadedb.ha.raftStorageDirectory` to a fixed, writable path backed by a per-pod PVC (an extra `volumeClaimTemplates` entry), so the `raft-storage-*` folder is always created under the same configured path:

```
extraCommands:
  - -Darcadedb.ha.raftStorageDirectory=/var/lib/arcadedb/raft
```

Service account

The chart creates a dedicated `ServiceAccount` for ArcadeDB pods by default. Disable creation and supply an existing `ServiceAccount` name when you manage service accounts outside Helm — for example, to attach an IRSA annotation (AWS EKS) or a Workload Identity annotation (GKE) that grants the pod cloud-provider permissions such as reading secrets from AWS Secrets Manager or Google Secret Manager.

Key	Default	Description
`serviceAccount.create`	`true`	Create a `ServiceAccount` as part of the release. Set to `false` to reuse a pre-existing account (supply its name in `serviceAccount.name`).
`serviceAccount.automount`	`false`	Mount the service account token into pods. ArcadeDB does not call the Kubernetes API, so the token is unnecessary; keeping it unmounted reduces the attack surface.
`serviceAccount.annotations`	`{}`	Annotations applied to the `ServiceAccount` object. Used for cloud-provider workload identity (e.g. `eks.amazonaws.com/role-arn` for IRSA, `iam.gke.io/gcp-service-account` for GKE Workload Identity).
`serviceAccount.name`	`""`	Name of the `ServiceAccount` to use. When empty and `serviceAccount.create` is `true`, the chart generates a name from the release.

Example override (reusing an existing service account with a GKE Workload Identity annotation):

```
serviceAccount:
  create: false
```

```
  name: arcadedb-wi-sa
  annotations:
    iam.gke.io/gcp-service-account: arcadedb@my-gcp-project.iam.gserviceaccount.com
```

Scheduling

`nodeSelector`, `tolerations`, and `affinity` control which nodes the ArcadeDB pods are placed on. The chart ships with a default `podAntiAffinity` rule (weight 100) that prefers scheduling each pod on a different node — the most useful pattern for Raft-based HA clusters, since it distributes quorum peers across failure domains and keeps the cluster alive when a single node goes down.

Key	Default	Description
`nodeSelector`	`{}`	Map of node labels that pods must match. Use to restrict ArcadeDB to a specific node pool (e.g. `cloud.google.com/gke-nodepool: db-pool`).
`tolerations`	`[]`	List of Kubernetes `Toleration` objects. Needed when ArcadeDB should run on tainted nodes (e.g. dedicated database nodes marked with `NoSchedule`).
`affinity`	(pod anti-affinity — see below)	Full `affinity` stanza. The default contains a `preferredDuringSchedulingIgnoredDuringExecution` pod anti-affinity rule that spreads pods across distinct hostnames. Override the entire key to replace it.

The chart's built-in `affinity` default:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: app.kubernetes.io/name
                operator: In
                values:
                  - arcadedb
          topologyKey: kubernetes.io/hostname
```

Example override (required anti-affinity across availability zones — stricter than the default preference-based rule):

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
```

```
            - key: app.kubernetes.io/instance
              operator: In
              values:
                - my-arcadedb
        topologyKey: topology.kubernetes.io/zone
```

Switching from `preferredDuringScheduling` to `requiredDuringScheduling` means Kubernetes will leave pods `Pending` rather than co-locate them when there are fewer zones than replicas. Verify that your cluster has at least as many zones (or nodes) as `replicaCount` before using `required` rules.

Pod metadata

`podAnnotations` and `podLabels` attach metadata directly to the pod objects created by the `StatefulSet`. Annotations are the standard integration point for cluster tooling: Prometheus scrape configuration, secret injection sidecars (Vault Agent, External Secrets), and service mesh proxy injection all rely on pod annotations. Labels supplement the chart-generated selectors and are useful for observability grouping, cost allocation, or policy enforcement tools that filter by label.

Key	Default	Description
`podAnnotations`	`{}`	Annotations applied to every pod. Common uses: Prometheus scrape hints, Vault Agent injection, Datadog APM tracing, Istio sidecar policy.
`podLabels`	`{}`	Extra labels applied to every pod. Merged with the chart-generated labels (`app.kubernetes.io/*`). Useful for cost-allocation labels, Cilium network policies that match on custom labels, or PodDisruptionBudget selectors.

Example override (Prometheus scrape annotations):

```
podAnnotations:
  prometheus.io/scrape: "true"
  prometheus.io/port: "2480"
  prometheus.io/path: "/metrics"
```

The Prometheus annotations shown above work with the community `prometheus/prometheus` scrape-config style. If you use the Prometheus Operator (`ServiceMonitor`/`PodMonitor` CRDs), configure a `PodMonitor` resource instead and omit these annotations. Enable the Prometheus plugin (`arcadedb.plugins.prometheus.enabled: true`) to expose the `/metrics` endpoint.

Network policy

Setting `networkPolicy.enabled: true` creates a Kubernetes `NetworkPolicy` that restricts traffic to and from ArcadeDB pods: the HTTP port (2480) accepts connections from any pod in the cluster, while

the Raft gRPC port (2434) is locked down to ArcadeDB pods only. This requires a CNI plugin that enforces `NetworkPolicy` resources — Calico, Cilium, Weave Net, and most managed Kubernetes offerings (GKE, EKS with VPC CNI + policy, ASK Azure CNI) support this; vanilla `kubenet` does not.

Key	Default	Description
`networkPolicy.enabled`	`false`	Create `NetworkPolicy` resources for the release. When `true`, inbound HTTP (2480) is permitted from all pods in the cluster; Raft gRPC (2434) is restricted to pods belonging to the same ArcadeDB release.

Example override:

```
networkPolicy:
  enabled: true
```

Enabling `networkPolicy` has no effect if your cluster's CNI does not enforce `NetworkPolicy` objects. Verify enforcement with `kubectl auth can-i create networkpolicies` and test the policy with a known-blocked source after enabling it.

Extras

The escape-hatch keys let you attach resources that the named chart values do not cover: extra Kubernetes manifests rendered alongside the chart, extra volumes and volume mounts injected into the container spec, and additional `volumeClaimTemplates` for the `StatefulSet`. These are advanced options intended for cases where a named value does not exist — for example, mounting a `ConfigMap` as a configuration file, adding a sidecar's shared `emptyDir`, or provisioning a separate PVC for WAL logs.

Key	Default	Description
`extraManifests`	`{}`	Map of arbitrary Kubernetes manifests rendered as part of the Helm release. Use to co-deploy `ConfigMap`, `ExternalSecret`, or other objects that must live in the same namespace and lifecycle as the chart.
`volumes`	`[]`	List of extra `volume` entries added to the pod spec. Pair with `volumeMounts` to expose the volume inside the container.
`volumeMounts`	`[]`	List of extra `volumeMount` entries added to the ArcadeDB container spec. The `mountPath` must not overlap with the persistence mount (`/home/arcadedb/databases`).
`volumeClaimTemplates`	`[]`	Extra `volumeClaimTemplates` for the `StatefulSet` (e.g. a dedicated PVC for WAL files, replication logs, or backups). The primary database PVC is controlled by `persistence.enabled` — see Persistence.

Example override (mounting an extra `ConfigMap` as a config file):

```
volumes:
  - name: custom-config
    configMap:
      name: arcadedb-extra-config
volumeMounts:
  - name: custom-config
    mountPath: /home/arcadedb/config/custom.json
    subPath: custom.json
    readOnly: true
```

The corresponding `ConfigMap` can be co-deployed via `extraManifests`:

```
extraManifests:
  arcadedb-extra-config:
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: arcadedb-extra-config
    data:
      custom.json: |
        {
          "key": "value"
        }
```

8.4.5. Operating the cluster

Inspecting cluster status and testing failover

The `/api/v1/server` endpoint reports the live HA state — current leader, replicas, and roles. Port-forward the HTTP service and read the `ha` object:

```
kubectl -n arcadedb port-forward svc/my-arcadedb-http 2480:2480 &
curl -u root:<password> http://localhost:2480/api/v1/server | jq .ha
```

To confirm automatic failover works, delete the leader pod and watch the cluster re-elect a new leader and restart the deleted pod:

```
kubectl -n arcadedb delete pod my-arcadedb-0
kubectl -n arcadedb get pods -w
```

The cluster re-elects a leader within seconds; the `StatefulSet` then recreates the deleted pod, which rejoins as a replica and catches up via the Raft log. See High Availability for the underlying election and catch-up semantics.

Scaling

Use `helm upgrade` to change the replica count persistently:

```
helm upgrade my-arcadedb arcadedb/arcadedb \
  --namespace arcadedb \
  --reuse-values \
  --set replicaCount=5
```

`kubectl scale statefulset` also works for an immediate, one-off change:

```
kubectl -n arcadedb scale statefulset my-arcadedb --replicas=5
```

A `kubectl scale` change is reverted on the next `helm upgrade` because Helm reapplies the `replicaCount` from the chart values. For persistent changes, prefer `helm upgrade --set replicaCount=N` (or update the values file).

Scaling does not pause the workload. New pods auto-join the Raft cluster, and pods removed during scale-down auto-leave via the `preStop` hook (see High Availability on Kubernetes).

For cluster sizing, three to five nodes covers most workloads; seven is the practical upper end before Raft replication overhead outweighs the added fault tolerance. When scaling down, remove one pod at a time and never drop below the Raft majority — losing quorum stalls writes until enough peers rejoin.

Upgrading the chart

```
helm repo update
helm upgrade my-arcadedb arcadedb/arcadedb \
  --namespace arcadedb \
  --reuse-values
```

Pin `image.tag` to a specific ArcadeDB version for reproducible upgrades.

Backups

Direct the backup output to a volume **outside** the database PVC — for example a dedicated `volumeClaimTemplates` entry (see Extras) or an object-store target — so a backup never competes with live data for space and survives deletion of the database volume. In an HA cluster, take the backup at the **leader** for a consistent snapshot of committed state.

Observability

Enable the Prometheus plugin (`arcadedb.plugins.prometheus.enabled: true`) to expose ArcadeDB metrics on the HTTP port, then scrape them with Prometheus (see Pod metadata for scrape annotations or a `PodMonitor`). Beyond the usual host and JVM metrics, alert on:

- **Raft leader changes** — frequent re-elections signal node instability or network problems.
- **replication lag** — a follower falling persistently behind risks a full snapshot transfer and weakens read freshness.
- **PVC capacity** — ArcadeDB stops accepting writes when the database volume fills; alert well before the PVC is full.

Uninstalling

```
helm uninstall my-arcadedb --namespace arcadedb
```

Persistent volume claims **survive uninstall** by design. To remove them too:

```
kubectl -n arcadedb delete pvc -l app.kubernetes.io/name=arcadedb
```

The `Secret` holding the root password is not managed by Helm and is also retained. Delete it explicitly if no longer needed.

8.4.6. High Availability on Kubernetes

ArcadeDB on Kubernetes uses the standard `StatefulSet` + headless `Service` pattern (the same pattern used by `etcd`, CockroachDB, and Apache Ozone). The chart pre-computes the full peer list from `replicaCount` and injects it into each pod via environment variables, so users do not configure `arcadedb.ha.serverList` by hand.

The chart sets the equivalent of:

```
arcadedb.ha.enabled=true
arcadedb.ha.k8s=true
arcadedb.ha.k8sSuffix=.my-arcadedb.arcadedb.svc.cluster.local
arcadedb.ha.serverList=my-arcadedb-0.my-
arcadedb.arcadedb.svc.cluster.local:2434:2480,my-arcadedb-1.my-
arcadedb.arcadedb.svc.cluster.local:2434:2480,my-arcadedb-2.my-
arcadedb.arcadedb.svc.cluster.local:2434:2480
```

(Substitute your release name and namespace.)

Auto-join on scale-up

When `arcadedb.ha.k8s=true` and a new pod starts without an existing Raft storage directory, the server automatically joins the existing cluster via the Raft `SetConfiguration(ADD)` admin API. This enables zero-downtime scale-up: `helm upgrade --set replicaCount=N` (or `kubectl scale statefulset`) adds new pods that join the existing cluster without restarting any existing peer. The same probe runs on every pod start as a membership self-check: a pod the cluster dropped while it was down re-adds itself. Membership is read from the Raft **leader's** configuration only. A peer that answers as a follower points the probe at the leader it follows, and while no peer answers as the leader (the cluster is still electing) the probe keeps retrying instead of trusting a

follower, whose configuration can predate the pod's removal. *(Since v26.11.1: the first peer to answer decided, so a pod removed from the cluster could be told it was still a member by a follower that was behind, and then stayed out of the cluster until restarted.)* A pod that joined beyond the peers of `arcadedb.ha.serverList` is routed to like the listed ones: the Bolt routing table offers it as a reader, and as the writer while it leads, a gRPC call refused on a follower names it when it leads, and `GET /api/v1/cluster` reports it in `peers[]` with its `httpAddress`. *(Since v26.11.1: such a pod was replicated to but never routed to, and the workaround was to pre-size `arcadedb.ha.serverList` to the maximum pod count.)*

Auto-leave on scale-down

The chart installs a `preStop` hook that calls `POST /api/v1/cluster/leave` so the terminating pod cleanly transfers leadership (if it holds it) and removes itself from the Raft group before shutdown.

Securing the Raft port with mTLS

The Raft gRPC port carries AppendEntries, RequestVote and the Ratis admin/client calls in plaintext by default, guarded only by an IP-based peer allowlist. On a shared cluster, enable mTLS with `arcadedb.ha.tls.*` and pair it with a `NetworkPolicy` restricting the port to the StatefulSet's own pods. cert-manager is the natural issuer here: a `Certificate` writes a `Secret` holding exactly the three files the settings need (`tls.crt`, `tls.key`, `ca.crt`). Two details are easy to get wrong on Kubernetes — the certificate's SAN must be the pod's stable headless-service DNS name (`my-arcadedb-0.my-arcadedb.arcadedb.svc.cluster.local`) rather than the pod IP, and `privateKey.encoding` must be set to `PKCS8`, because cert-manager defaults to `PKCS1`, which the Raft transport does not parse. Mount the `Secret` read-only through the chart's `volumes` / `volumeMounts` (see Extras) with `defaultMode: 0400`, and point the settings at it via `arcadedb.extraCommands`. Auto-join needs no extra configuration: the join probe inherits the server's transport settings, so it speaks TLS exactly when the cluster does. For the manifests, the openssl equivalent, and the certificate-rotation caveat, see mTLS for the Raft gRPC transport.

Users and API tokens on a restarted pod

The server users, groups and API tokens live in the server's `config/` directory, not in a database, so a pod whose data volume is persistent but whose `config/` directory is not comes back without them. Because it is still a member of the Raft group, it is not re-admitted. Such a pod asks the leader for the current documents shortly after it first sees one, so no action is needed. With `arcadedb.server.readinessRequiresHA=true`, a pod added by auto-join on scale-up, or one caught up by a snapshot, also stays out of the Service endpoints until those documents arrive, for at most `arcadedb.ha.securityConvergenceReadinessTimeout` (30 s by default); see security-convergence readiness. Persisting `config/` is still worth doing if you want a pod to come back with them already in place; see Cluster security documents. You can mount a persistent volume through the chart's `volumes` / `volumeMounts` (see Extras) and point `arcadedb.server.configDirectory` at it with an entry in `arcadedb.extraCommands`; see Relocating the configuration directory for what moves and what does not.

For the rest of the Raft semantics — quorum, named peers, snapshot threshold, election timeouts, cluster token, replication failure handling — see High Availability.

Pre-staging the database on every pod

When you want to scale a single-pod ArcadeDB deployment up to N pods after importing a large (1+ GB) database, pre-stage identical database files on every pod's PVC and let the cluster recognise the matching state at first formation, rather than letting the leader stream the full database to every new replica over HTTP.

The chart leaves the bootstrap feature enabled by default (`arcadedb.ha.bootstrapFromLocalDatabase=true`). Operator workflow:

1. Install the chart with `replicaCount: 1` (no HA) and import the dataset.
2. After import is complete, tar the database directory or take a regular full ArcadeDB backup.
3. Distribute the archive into every pod's PVC. Common patterns: an init container that pulls from S3/GCS, an `nfs:` volume mount with the archive, baking the archive into the container image, or `kubectl cp` for one-off operator workflows. Each pod's PVC must contain the same database files under `arcadedb.databaseDirectory`.
4. `helm upgrade` the release with `replicaCount: 3` (or more). The new pods come up with non-empty databases and the same fingerprint; the cluster elects the bootstrap source automatically and forms in seconds.

For the underlying protocol and edge cases (different-age peers, late newer joiner, cluster restart with stale data on a node), see Offline Cluster Bootstrap.

8.4.7. Troubleshooting

The most common deployment failure is a missing or incorrectly named `rootPassword Secret` — pods crash-loop with an authentication-related error in the log. Verify the secret exists and that the chart's `arcadedb.credentials.rootPassword.secret.name` and `.key` match it.

Inspect a pod's events:

```
kubectl -n arcadedb describe pod my-arcadedb-0
```

Read the server log:

```
kubectl -n arcadedb logs my-arcadedb-0
```

(Replace `my-arcadedb-0` with the pod name you want to inspect.)

To tear down a stuck cluster and start over:

```
helm uninstall my-arcadedb --namespace arcadedb
kubectl -n arcadedb delete pvc -l app.kubernetes.io/name=arcadedb
```

Then re-run `helm install` with corrected values.

8.4.8. Reference deployment

The `ArcadeData/arcadedb-deployments` repository hosts a working, scripted reference deployment under `kubernetes/`. It pairs a kind cluster with the official chart and provides:

- `start.sh` — creates the kind cluster, runs `helm dependency update` + `helm install --wait`, and starts a port-forward to `localhost:2480`.
- `stop.sh` — terminates the port-forward, runs `helm uninstall`, and destroys the kind cluster.
- `test.sh` — validates that the cluster is functional.
- `values.yaml` — the values overrides used by `start.sh`. Note that this file wraps overrides under a top-level `arcadedb:` key because it consumes the chart as a **subchart**; when installing the chart directly (the path documented above), use the same overrides at the top level.

Use this repository as a starting point for your own setup and as a reproducible bug-report environment.

8.5. Installation on AWS (Marketplace AMI)

ArcadeDB publishes an official Amazon Machine Image (AMI) on AWS Marketplace, sold by ArcadeDB. It installs the open source (Apache 2.0) ArcadeDB server on Amazon EC2 with nothing left to set up:

- CIS-hardened Ubuntu 22.04 LTS with OpenJDK 25
- ArcadeDB installed as the `arcadedb` systemd service, started on boot
- A root password generated on each instance's first boot, with no default credentials
- Built for x86_64 and AWS Graviton (arm64) instances

There are two listings, one per CPU architecture:

Architecture	Listing
x86_64 (t3, m6i/m7i, c6i/c7i, r6i/r7i)	ArcadeDB Community Edition (x86_64)
arm64 (t4g, m6g/m7g, c6g/c7g, r6g/r7g)	ArcadeDB Community Edition (arm64)

The AMI carries an hourly software fee, set by instance size, on top of the EC2, EBS and data transfer charges. New subscribers get a 7-day free trial of the software fee. The current rates are on each listing's pricing tab. The fee appears on your AWS bill, so there is no separate contract or vendor to onboard. If you prefer to install ArcadeDB yourself on EC2 at no software cost, use the Docker image or the binary distribution; the database engine is the same.

The Marketplace image includes community support through GitHub issues and this documentation, with no SLA. For a support contract, see ArcadeDB support plans.

8.5.1. Launch an instance

1. Open the listing for your architecture and choose **View purchase options**, then **Subscribe**.
2. Choose **Launch your software**, pick a region and an instance type. `m6i.large` (or `m7g.large` on Graviton) is a good starting point.
3. Select your EC2 key pair.
4. In the security group, allow inbound TCP **22** (SSH) and **2480** (HTTP API and Studio) **only from your own IP range**. Do not open these ports to `0.0.0.0/0`. If you enable more protocols later, such as PostgreSQL (5432) or Bolt (7687), open their ports the same way.
5. Launch the instance and wait for it to pass its status checks.

8.5.2. Read the root password

Every instance generates its own `root` password on first boot. SSH in as the `ubuntu` user and read it:

```
ssh -i my-key.pem ubuntu@<instance-public-ip>
cat ~/arcadedb-root-password.txt
```

Store the password in your secrets manager, then delete the file from the instance.

8.5.3. Connect

Open Studio at `http://<instance-public-ip>:2480` and log in as `root`. To keep port 2480 closed to the internet altogether, reach it through an SSH tunnel instead and open `http://localhost:2480`:

```
ssh -i my-key.pem -L 2480:localhost:2480 ubuntu@<instance-public-ip>
```

Check the server from the command line:

```
curl -s -u root:<password> http://<instance-public-ip>:2480/api/v1/server?mode=basic
```

Applications connect over the HTTP API on port 2480 with any of the drivers. The plain HTTP port is meant for private networks: for traffic that crosses the internet, put an Application Load Balancer with a TLS certificate in front of the instance, or connect over a VPN or SSH tunnel.

8.5.4. Operate the server

Task	How
Service status	`sudo systemctl status arcadedb`
Restart	`sudo systemctl restart arcadedb`
Database files	`/var/lib/arcadedb/databases`

Keep the databases on the instance's EBS volume and back them up with ArcadeDB backups or EBS

snapshots. The software fee stops while the instance is stopped; EBS storage is still billed.

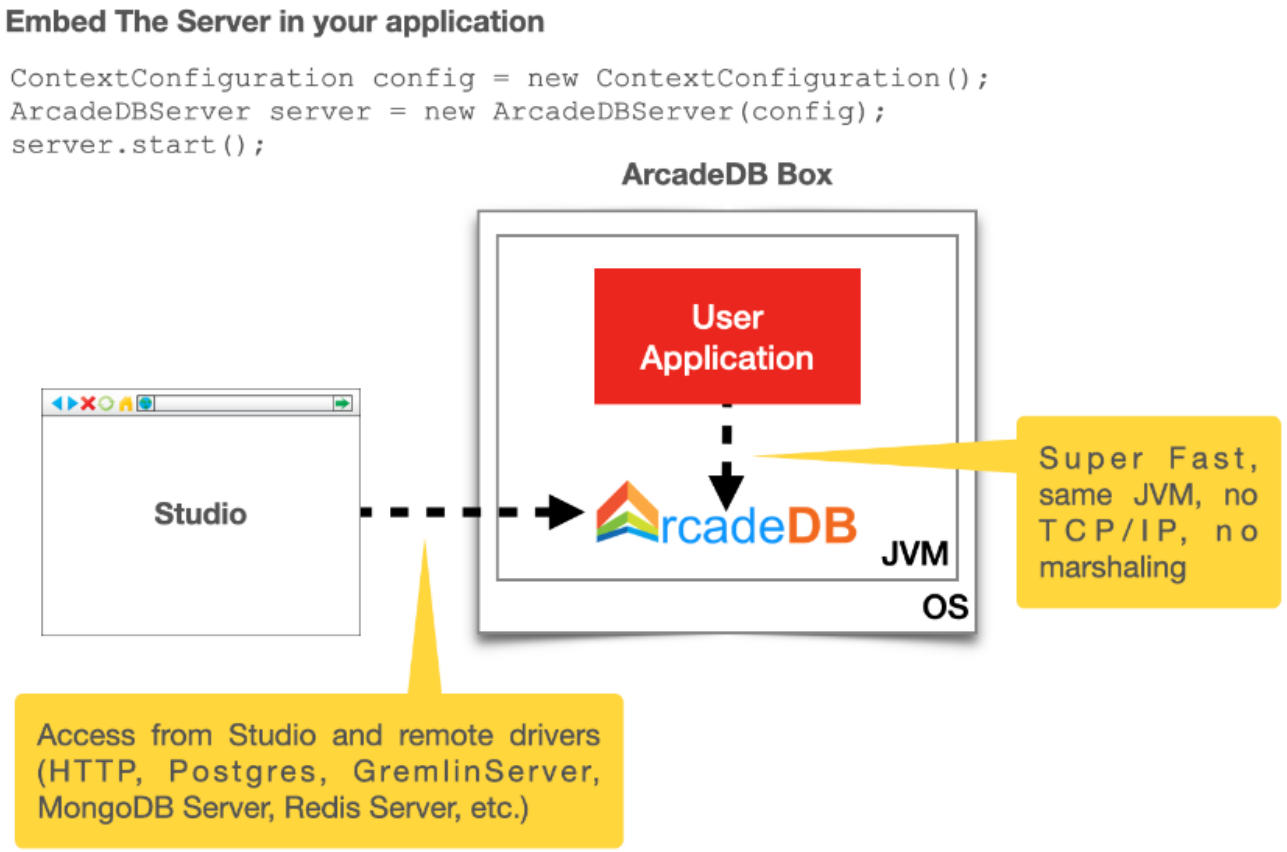
8.5.5. Upgrade

A new ArcadeDB release is published as a new version of the Marketplace listing. To move an existing instance to it, take a backup, launch a new instance from the new version, and restore the backup there, following the upgrade checklist of the release.

8.6. Embedded Server

Embedding the server in your JVM allows to have all the benefits of working in embedded mode with ArcadeDB (zero cost for network transport and marshalling) and still having the database accessible from the outside, such as Studio, remote API, Postgres, REDIS and MongoDB drivers.

We call this configuration an "ArcadeDB Box".



First, add the server library in your classpath. If you're using Maven include this dependency in your `pom.xml` file.

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-server</artifactId>
    <version>26.11.1</version>
</dependency>
```

This library depends on `arcadedb-network-<version>.jar`. If you're using Maven or Gradle, it will be

imported automatically as a dependency, otherwise please add also the `arcadedb-network` library to your classpath.

The `arcadedb-server` dependency will only start the ArcadeDB server. You will see the HTTP URL for the server along with the port number displayed, for example, `http://localhost:2480`. However, if you try to access this URL to see the ArcadeDB studio, you will receive a `"Not Found"` message. This is because the arcadedb-server dependency only adds the embedded server. If you need to access the ArcadeDB studio to execute graph database queries, then you will need to add the following dependency:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-studio</artifactId>
    <version>26.11.1</version>
</dependency>
```

8.6.1. Java 17 notes

Java 17 packages are available through GitHub package. To use them, you need to add the GitHub package repository to your `pom.xml` file:

```
<repositories>
  <repository>
    <id>github</id>
    <url>https://maven.pkg.github.com/ArcadeData/arcadedb</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```

Then, you can add the `arcadedb-server` dependency with the `-java17` suffix to use the Java 17 compatible version:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-server</artifactId>
    <version>26.11.1-java17</version>
</dependency>
```

8.6.2. Start the server in the JVM

To start a server as embedded, create it with an empty configuration, so all the setting will be the default ones:

```
ContextConfiguration config = new ContextConfiguration();
ArcadeDBServer server = new ArcadeDBServer(config);
server.start();
```

To start a server in distributed configuration (with replicas), you can set your settings in the `ContextConfiguration`:

```
config.setValue(GlobalConfiguration.HA_SERVER_LIST,
"192.168.10.1,192.168.10.2,192.168.10.3");
config.setValue(GlobalConfiguration.HA_REPLICATION_INCOMING_HOST, "0.0.0.0");
config.setValue(GlobalConfiguration.HA_ENABLED, true);
```

High availability is no longer part of `arcadedb-server`. Since the migration to Apache Ratis, the HA implementation lives in a separate module, `arcadedb-ha-raft`, and is discovered at startup via the Java `ServiceLoader`. If you embed only `arcadedb-server`, setting `HA_ENABLED=true` (or a non-blank `HA_SERVER_LIST`) is **not** enough: with no HA plugin on the classpath the node starts STANDALONE with no replication. Add the dependency explicitly:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-ha-raft</artifactId>
    <version>26.11.1</version>
</dependency>
```

The official server distribution and Docker image already bundle this module, so this step only applies to embedded deployments that manage their own dependencies. When HA is requested but no plugin is found, the server logs a `WARNING` at startup and records a server event under the `HA` component.

When you embed the server, you should always get the database instance from the server itself. This assures the database instance is just one in the entire JVM. If you try to create or open another database instance from the `DatabaseFactory`, you will receive an error that the underlying database is locked by another process.

```
Database database = server.getDatabaseForOperator(<NAME>);
```

`getDatabaseForOperator` is the operator's entry point: it opens the database even when an operator closed it with `close database`, or while a restore is replacing it. Code that resolves a database on behalf of a client (a protocol, a request handler, a plugin serving users) calls `getDatabaseForClient(<NAME>)` instead, which refuses such a database with `DatabaseNotAvailableException`. Before v26.11.1 `getDatabaseForOperator` was named `getDatabase(String)`: that name still works and is deprecated, to be removed in a later release.

Or this if you want to create a new database if not exists:

```
Database database = server.getOrCreateDatabase(<URL>);
```

8.6.3. Create custom HTTP commands

You can easily add custom HTTP commands on ArcadeDB's Undertow HTTP Server by creating a Server Plugin (look at MongoDBProtocolPlugin plugin implementation for a real example) and implementing the registerAPI method. Example for the HTTP POST API /myapi/test:

```
package com.yourpackage;

public class MyTest implements ServerPlugin {
  // ...

  @Override
  public void registerAPI(HttpServer httpServer, final PathHandler routes) {
    routes.addPrefixPath("/myapi",//
      Handlers.routing()//
        .get("/account/{id}", new RetieveeAccount(this))// YOU CAN ADD YOUR HANDLERS
UNDER THE SAME PREFIX PATH
        .post("/test/{name}", new MyTestAPI(this))//
    );
  }
}
```

You can use GET, POST or any HTTP methods when you register your handler. Note that multiple handlers are defined under the same prefix /myapi. Below you can find the implementation of a "Test" handler that can be called by using the HTTP POST method against the URL /myapi/test/{name} where {name} is the name passed as an argument. Note that the MyTestAPI class is inheriting DatabaseAbstractHandler to have the database instance as a parameter. If the user is not authenticated, the execute() method is not called at all, but an authentication error is returned. If you don't need to access to the database, then you can extend the AbstractHandler class instead.

```
public class MyTestAPI extends DatabaseAbstractHandler {

  public MyTestAPI(final HttpServer httpServer) {
    super(httpServer);
  }

  @Override
  public void execute(final HttpServerExchange exchange, ServerSecurityUser user,
final Database database) throws IOException {
    final Deque<String> namePar = exchange.getQueryParameters().get("name");
    if (namePar == null || namePar.isEmpty()) {
      exchange.setStatusCode(400);
      exchange.getResponseSender().send("{ \"error\" : \"name is null\"}");
```

```
      return;
    }

    final String name = namePar.getFirst();

    // DO SOMETHING MEANINGFUL HERE
    // ...

    exchange.setStatusCode(204);
    exchange.getResponseSender().send("");
  }
}
```

At startup, ArcadeDB server will initiate your plugin and register your API. To start the server with your plugin, register the full class in `arcadedb.server.plugins` setting:

Example:

```
$ java ... -Darcadedb.server.plugins=MyPlugin:com.yourpackage.MyPlugin ...
```

8.6.4. HTTPS connection

In order to enable HTTPS on ArcadeDB server, you have to set the following configuration before the server starts:

```
configuration.setValue(GlobalConfiguration.NETWORK_USE_SSL, true);
configuration.setValue(GlobalConfiguration.NETWORK_SSL_KEYSTORE,
"src/test/resources/master.jks");
configuration.setValue(GlobalConfiguration.NETWORK_SSL_KEYSTORE_PASSWORD,
"keypassword");
configuration.setValue(GlobalConfiguration.NETWORK_SSL_TRUSTSTORE,
"src/test/resources/master.jks");
configuration.setValue(GlobalConfiguration.NETWORK_SSL_TRUSTSTORE_PASSWORD,
"storepassword");
```

Where:

- `NETWORK_USE_SSL` enable the SSL support for the HTTP Server
- `NETWORK_SSL_KEYSTORE` is the path where is located the keystore file
- `NETWORK_SSL_KEYSTORE_PASSWORD` is the keystore password
- `NETWORK_SSL_TRUSTSTORE` is the path where is located the truststore file
- `NETWORK_SSL_TRUSTSTORE_PASSWORD` is the truststore password

Note that the default port for HTTPs is configured via the global setting:

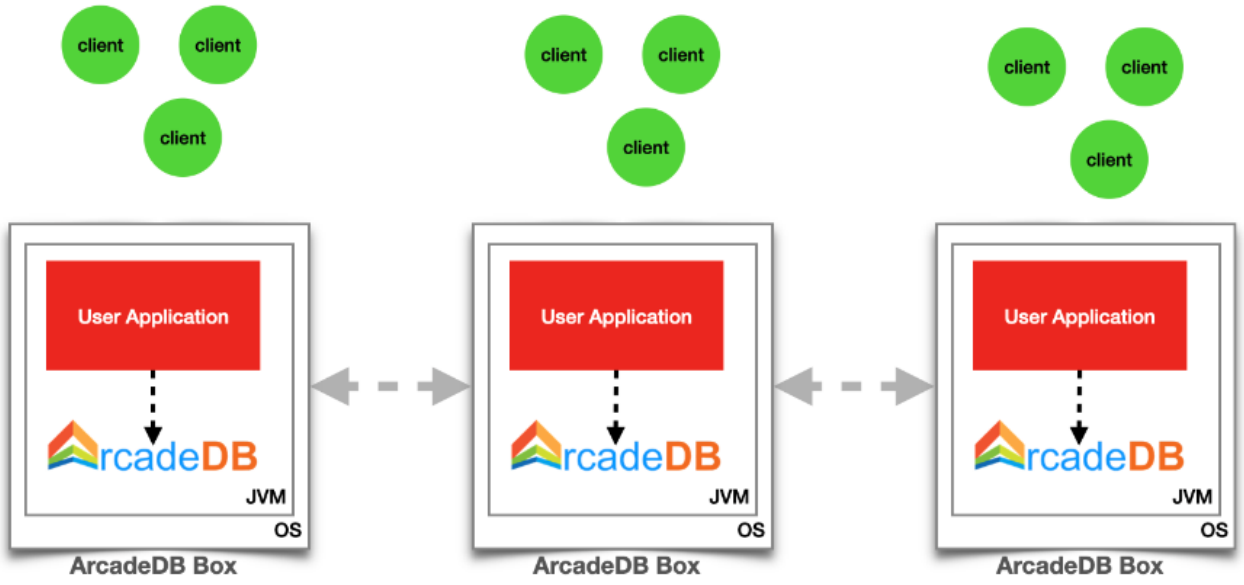
```
GlobalConfiguration.SERVER_HTTPS_INCOMING_PORT
```

And by default starts from 2490 to 2499 (increases the port if it's already occupied).

if HTTP or HTTPS port are already used, the next ports are used. With the default range of 2480-2489 for HTTP and 2490-2499 for HTTPS, if the port 2480 is not available, then the next port for both HTTP and HTTPS will be used, namely 2481 for HTTP and 2491 for HTTPS

8.6.5. Replication between boxes

You can replicate databases across multiple boxes to have true high availability:



Best of latency between User Application and Database Server + Replication for High Availability

8.7. Server

To start ArcadeDB as a server run the script `server.sh` under the `bin` directory of ArcadeDB distribution. If you're using MS Windows OS, replace `server.sh` with `server.bat`.

```
$ bin/server.sh

  ▯▯▯▯▯▯╗  ▯▯▯▯▯▯▯╗   ▯▯▯▯▯▯▯╗  ▯▯▯▯▯▯╗  ▯▯▯▯▯▯▯╗  ▯▯▯▯▯▯▯▯╗ ▯▯▯▯▯▯▯╗  ▯▯▯▯▯▯▯╗
▯▯ ╔═══▯▯╗ ▯▯ ╔═══▯▯╗ ▯▯ ╔═══════╝ ▯▯ ╔═══▯▯╗ ▯▯ ╔═══▯▯╗ ▯▯ ╔═══════╝ ▯▯ ╔═══▯▯╗ ▯▯ ╔═
═══▯▯╗
▯▯▯▯▯▯▯ || ▯▯▯▯▯▯ ╔═╝ ▯▯ ||        ▯▯▯▯▯▯▯ || ▯▯ ||   ▯▯ || ▯▯▯▯▯╗    ▯▯ ||    ▯▯ || ▯▯▯▯▯▯ ╔═╝
▯▯ ╔═══▯▯ || ▯▯ ╔═══▯▯╗ ▯▯ ||        ▯▯ ╔═══▯▯ || ▯▯ ||   ▯▯ || ▯▯ ╔═══╝    ▯▯ ||    ▯▯ || ▯▯ ╔═══▯▯╗
▯▯ ||   ▯▯ || ▯▯ ||   ▯▯ || ╚▯▯▯▯▯▯╗ ▯▯ ||   ▯▯ || ▯▯▯▯▯▯ ╔═╝ ▯▯▯▯▯▯▯╗ ▯▯▯▯▯▯ ╔═╝ ▯▯▯▯▯▯ ╔═╝
```

```
 ╚═══╝   ╚═══╝ ╚═══╝   ╚═══╝ ╚═══════════╝ ╚═══╝   ╚═══╝ ╚═══════════╝
 ╚═══════════╝ ╚═══════════╝ ╚═══════════╝
PLAY WITH DATA                                    arcadedb.com

2025-12-09 22:22:22.839 INFO  [ArcadeDBServer] <ArcadeDB_0> ArcadeDB Server v26.11.1
is starting up...
2025-12-09 22:22:22.841 INFO  [ArcadeDBServer] <ArcadeDB_0> Running on Mac OS X 15.6 -
OpenJDK 64-Bit Server VM 25.0.1 (Homebrew)
2025-12-09 22:22:22.843 INFO  [ArcadeDBServer] <ArcadeDB_0> Starting ArcadeDB Server
in development mode with plugins [] ...
2025-12-09 22:22:22.878 INFO  [ArcadeDBServer] <ArcadeDB_0> - Metrics Collection
Started...
2025-12-09 22:22:22.891 INFO  [ArcadeDBServer] <ArcadeDB_0> Server root path: .
2025-12-09 22:22:22.891 INFO  [ArcadeDBServer] <ArcadeDB_0> Databases directory:
./databases
2025-12-09 22:22:22.891 INFO  [ArcadeDBServer] <ArcadeDB_0> Backups directory:
./backups

+--------------------------------------------------------------------+
|                WARNING: FIRST RUN CONFIGURATION                    |
+--------------------------------------------------------------------+
| This is the first time the server is running. Please type a        |
| password of your choice for the 'root' user or leave it blank      |
| to auto-generate it.                                               |
|                                                                    |
| To avoid this message set the environment variable or JVM          |
| setting `arcadedb.server.rootPassword` to the root password to use.|
+--------------------------------------------------------------------+

Root password [BLANK=auto generate it]: *
```

The first time the server is running, the root password must be inserted and confirmed. The hash (+salt) of the inserted password will be stored in the file `config/server-users.json`. The password length must be between 8 and 256 characters. To know more about this topic, look at Security. Delete this file and restart the server to reinsert the password for server's root user.

The default rules of security are pretty basic. You can implement your own security policy. Check the Security Policy.

You can skip the request for the password by passing it as a setting. Example:

```
-Darcadedb.server.rootPassword=this_is_a_password
```

Alternatively the password can be passed file-based. Example:

```
-Darcadedb.server.rootPasswordPath=/run/secrets/root
```

which is particularly useful for container-based deployments.

The password file is a plain-text file and should not contain any line breaks / new lines.

Once inserted the password for the root user, you should see this output.

```
Root password [BLANK=auto generate it]: *********
*Please type the root password for confirmation (copy and paste will not work):
*********

2025-12-09 22:23:33.719 INFO  [HttpServer] <ArcadeDB_0> - Starting HTTP Server (host
=0.0.0.0 port=[2480, 2489] httpsPort=[2490, 2499])...
2025-12-09 22:23:33.738 INFO  [undertow] starting server: Undertow - 2.3.20.Final
2025-12-09 22:23:33.741 INFO  [xnio] XNIO version 3.8.16.Final
2025-12-09 22:23:33.744 INFO  [nio] XNIO NIO Implementation Version 3.8.16.Final
2025-12-09 22:23:33.803 INFO  [HttpServer] <ArcadeDB_0> - HTTP Server started (host
=0.0.0.0 port=2480 httpsPort=2490)
2025-12-09 22:23:33.908 INFO  [ArcadeDBServer] <ArcadeDB_0> Available query languages:
[sqlscript, mongo, gremlin, java, cypher, js, graphql, sql]
2025-12-09 22:23:33.909 INFO  [ArcadeDBServer] <ArcadeDB_0> ArcadeDB Server started in
'development' mode (CPUs=8 MAXRAM=4,00GB)
2025-12-09 22:23:33.910 INFO  [ArcadeDBServer] <ArcadeDB_0> Studio web tool available
at http://192.168.1.102:2480
```

A warning `WARNING: Using incubator modules: jdk.incubator.vector` in the log can be safely ignored.

By default, the following components start with the server:

- HTTP Server, that listens on port 2480 by default. If port 2480 is already occupied, then the next is taken up to 2489.

Remote JMX. *(Since v26.12.1)* `bin/server.sh`, `bin/server.bat` and the Docker image no longer open a remote JMX port. Earlier versions started every server with a JMX agent on ports 9999 and 9998, reachable from any host, with authentication and SSL disabled. A JMX client can invoke any MBean of the JVM, so that agent gave whoever reached the port code execution inside the server process. Local tools such as `jconsole`, VisualVM and `jcmd` still attach to a running server without any setting, as the same operating system user, and the server metrics are served over HTTP (Prometheus) when `arcadedb.serverMetrics` is enabled.

To open remote JMX, pass the JVM flags in the `ARCADEDB_JMX` environment variable with authentication and SSL on, and publish the ports yourself when running in Docker:

```
$ export ARCADEDB_JMX="-Dcom.sun.management.jmxremote.port=9999 \
    -Dcom.sun.management.jmxremote.rmi.port=9998 \
    -Dcom.sun.management.jmxremote.authenticate=true \
    -Dcom.sun.management.jmxremote.password.file=/path/to/jmxremote.password \
    -Dcom.sun.management.jmxremote.access.file=/path/to/jmxremote.access \
    -Dcom.sun.management.jmxremote.ssl=true \
```

```
    -Djavax.net.ssl.keyStore=/path/to/keystore -Djavax.net.ssl.keyStorePassword=..."
$ bin/server.sh
```

When the server finds a remote JMX agent with authentication disabled, however it was configured (`ARCADEDB_JMX`, `JAVA_OPTS` or a `com.sun.management.config.file`), it logs a `SEVERE` message at startup and records it in the server event log.

In the output above, the name `ArcadeDB_0` is the server name. By default, `ArcadeDB_0` is used. To specify a different name define it with the setting `server.name`, example:

```
$ bin/server.sh -Darcadedb.server.name=ArcadeDB_Europe_0
```

In a high availability (HA) configuration, it's mandatory that all the servers in an cluster have different names.

8.7.1. Start server hint

To start the server from a location different than the ArcadeDB folder, for example, if starting the server as a service, set the environment variable `ARCADEDB_HOME` to the ArcadeDB folder:

```
$ export ARCADEDB_HOME=/path/to/arcadedb
```

8.7.2. Server modes

The server can be started in one of three modes, which affect the studio, logging, and security defaults:

Mode	Studio	Logging	WAL Flush	LOAD CSV file access
`development`	Yes	Detailed	No flush (`0`)	Enabled
`test`	Yes	Brief	No flush (`0`)	Enabled
`production`	No (can be force-enabled)	Brief	Auto-set to `1` if not configured	Auto-disabled if not configured

The mode is controlled by the `server.mode` setting with a default mode `development`.

In `production` mode the Studio web tool is not served by default, to reduce the attack surface. If you need it (for example to inspect a database running in production), force-enable it with the `studio.enabled` setting:

```
$ bin/server.sh -Darcadedb.server.mode=production -Darcadedb.studio.enabled=true
```

When enabling Studio in production, restrict network access to the HTTP port (for example with a firewall, a reverse proxy, or by enabling SSL), since it exposes a web UI on the public-facing server.

Since v26.10.2 Studio's static files carry an `ETag` (a digest of their bytes) and answer `304 Not Modified` to a matching `If-None-Match`. Studio pages and the API docs page reference their local scripts and stylesheets as `<path>?v=<digest>`, and only those versioned URLs are sent `Cache-Control: public, max-age=31536000, immutable`; pages and unversioned asset URLs are sent `no-cache`, so the browser revalidates them. Before, assets were cached for one day (`max-age=86400`) and stylesheets not at all, so after an upgrade a browser could run a fresh page with stale scripts. Now a new version is picked up at once, and a library that did not change stays cached across upgrades. A reverse proxy or CDN in front of the server should honor these headers and keep the query string in its cache key. `.png` files are now served as `image/png`.

Production mode defaults

When the server starts in `production` mode, ArcadeDB automatically applies safe defaults for settings that have not been explicitly configured:

- **WAL flush** (`arcadedb.txWalFlush`): set to `1` (flush without metadata) for transaction durability. If you explicitly set it to `0`, a warning is logged. See concepts/transactions.pdf for details.
- **LOAD CSV file access** (`arcadedb.opencypher.loadCsv.allowFileUrls`): disabled to prevent Cypher queries from reading local files on the server. Set to `true` explicitly if needed.
- **Studio web tool** (`arcadedb.studio.enabled`): the Studio web UI is not served. Set it to `true` to force-enable Studio in production (see Server modes).

These defaults are only applied when the setting has not been explicitly configured by the user (via system property, environment variable, or API call). Explicit settings are always respected.

At startup, a production checklist is logged summarizing the security-relevant configuration:

```
INFO  Production checklist:
INFO  - WAL flush: 1 (flush without metadata) [OK]
INFO  - SSL: disabled. Consider enabling for encrypted connections
INFO  - High availability: disabled. Consider enabling for fault tolerance
INFO  - LOAD CSV file access: disabled [OK]
INFO  - Backup support: enabled [OK]
INFO  - Studio web tool: disabled [OK]
```

Error details in production mode

In `production` mode no wire surface sends the engine's own error text to clients. That text can carry stored data (the values of a duplicated key) or details of the server (file paths, URLs), so it is replaced by one generic message, the same on every surface and for every failure: "The request failed. Check the server log for the details". The full error is written to the server log instead. This applies to:

- **HTTP**: the `detail` field is left out of the error response.
- **gRPC**: the status description carries the generic message.

- **PostgreSQL**: the error message (the `M` field) carries the generic message. The SQLSTATE code is kept, so drivers still tell a duplicated key (`23505`) or a syntax error (`42601`) apart.
- **MongoDB**: the protocol words its own error messages and keeps the engine message in the server log.
- **Gremlin Server**: the default WebSocket channelizer answers with the generic message. Since v26.10.2 a non-default channelizer, which cannot conceal the text, stops the server from starting in production mode, see Gremlin security.
- **Bolt**: the `FAILURE` message carries the generic message. The Neo4j status code is kept, so drivers still classify and retry the failure. A Cypher syntax error is concealed too, and keeps its `Neo.ClientError.Statement.*` code.
- **Redis**: the text of an error reply after its `ERR`, `TRYAGAIN` or `NOPERM` prefix is the generic message. Errors of the RESP protocol itself, such as `wrong number of arguments` or `value is not an integer or out of range`, and the refusal of a credential or a permission stay visible.
- **`/ws` insert session**: the `error` frames and the `message` of each entry of a `batchAck errors` array carry the generic message. The `code` and the exception class are kept. Text the protocol writes about the request itself, such as a missing `keyColumns`, an invalid option or an expired session, stays visible, as it is what the client needs to fix the request.
- **`/ws` change-event subscriptions**: the `error` frames of a failed action carry the generic message.
- **MCP**: a tool error, and the `Internal error` of a JSON-RPC error, carry the generic message. The validation of the tool arguments (a missing `database`, an unknown tool, a limit out of range) stays visible.

The structured part a client branches on (an error code, a Neo4j status, a RESP prefix, an exception class name) is never concealed, and neither is text the protocol layer writes itself about the request, which carries no stored data. One addition is made in every mode: when an HA replica refuses work that only the leader can run, the answer names the leader, for example `The request failed. Check the server log for the details. The leader is server 'arcadedb-0:2480'`, on every protocol, MCP and the Gremlin Server included. It is a cluster address, not stored data, and it is what tells the client where to retry. The Gremlin Server names the leader's HTTP address: connect to the Gremlin port of that server. In the server log, a failure the client did not cause is logged at `SEVERE` with its stack trace, and a failure the client caused (a duplicated key, a syntax error, a validation refusal) is logged with its message only: at `INFO`, or at `FINE` for the per-row errors of a `/ws` insert session, where a bulk load can refuse thousands of rows by design.

The values of a duplicated key are stored data, so they never reach the client in production mode. When a unique index refuses a write, a remote client still receives the exception type, the index name and the existing record's RID, but `DuplicatedKeyException.getKeys()` returns `[concealed]` instead of the key values. HTTP (`exceptionArgs`), gRPC (the `arcadedb-dup-keys` trailer), PostgreSQL and MongoDB behave the same way; on Bolt, Redis, the `/ws` insert session and MCP the whole message is concealed. In `development` and `test` mode the key values and the full error messages are returned as before.

Before v26.10.2 the `/ws` insert session, `/ws` subscriptions, Bolt, Redis and MCP returned the engine's error message as is in every mode, duplicated key values included. A client of these protocols that parses the error text receives the generic

message in production mode now: branch on the error code, the Neo4j status or the RESP prefix instead, and read the detail from the server log.

8.7.3. User changes on open connections

(Since v26.10.2.) A long-lived connection keeps the user it authenticated as, but the server checks that user again against the current security configuration on every request, as it does for every HTTP call. A user deleted, given a new password, or stripped of a database grant after the connection logged in loses access on its next request instead of keeping it for the life of the connection:

- **PostgreSQL**: a deleted or re-passworded user, or one that lost the grant on the connected database, gets a `FATAL` error with SQLSTATE `28P01` ("Credentials not valid") and the connection is closed.
- **Bolt**: a deleted or re-passworded user gets `Neo.ClientError.Security.Unauthorized` ("The credentials of this connection are no longer valid") and its open transaction is rolled back. A user that lost the grant on the database cannot commit the explicit transaction it opened: it is rolled back.
- **MongoDB**: a deleted or re-passworded user is logged out and the command is refused with an `Unauthorized` error; a user that lost the grant on the database is refused too.
- **Redis**: a deleted or re-passworded user is logged out: the command, and every later one, is refused with `NOAUTH Authentication required.` until the client sends `AUTH` again.
- `/ws`: every frame re-checks the user, for change-event subscriptions and insert sessions alike. A deleted or re-passworded user gets an `error` frame and the connection is closed with code `1008` ("Principal is no longer valid"); see Streaming Inserts over WebSocket for what happens to an open insert session.

8.7.4. Create default database(s)

Instead of starting a server and then connect to it, to create the default databases, ArcadeDB Server takes an initial default databases list by using the setting `server.defaultDatabases`.

```
$ bin/server.sh "-Darcadedb.server.defaultDatabases=Universe[albert:einstein]"
```

With the example above the database "Universe" will be created if doesn't exist, with user "albert", password "einstein".

Due to the use of `[]`, the command line argument needs to be wrapped in quotes.

A default database without users still needs to include empty brackets, ie: `-Darcadedb.server.defaultDatabases=Multiverse[]`

`:` separates the fields of a credential, `,` the credentials, `;` the databases and `[/]` the credentials block, so a user name, password or group that contains one of them writes it percent-encoded, as in the user part of a URI: `%3A` for `:`, `%2C` for `,`, `%3B` for `;`, `%5B` for `[`, `%5D` for `]` and `%25` for `%`. Only a `%` followed by

two hexadecimal digits is decoded, so a password that merely contains a `%` or a `+` reads as written. For example `-Darcadedb.server.defaultDatabases=Universe[albert:e%3Dmc%3A2:admin]` gives "albert" the password `e=mc:2` in the `admin` group.

Since 26.12.1 (issue #11269) a credential with more than three `:`-separated fields is refused with a warning, and neither creates the user nor grants it the database: before, a password written with a raw `:` was cut at it, the text after it became the group (`mypass:admin` meant the password `mypass` in the `admin` group) and the rest was dropped. A credential with an empty user name or password is refused the same way, and so is an entry with text after its `]` that does not open a `{…}` command list, which is what a raw `]` inside a password leaves behind. A password containing a sequence like `%41` now decodes to a different password: an existing user is then reported with the `already exists with a different password` warning and keeps its password, so write such a `%` as `%25`.

The users are written to `config/server-users.jsonl` before they take effect. Since 26.11.1 (issue #10535) a start that has to add a user, or to give an existing user access to the database or one more group, and cannot write that file (a read-only configuration directory, such as a Kubernetes ConfigMap mount) stops with an error naming the file, and nothing is granted in memory that is not on disk. A user that already holds the database under the group the entry names is left alone and nothing is written, so once the first start has made the grants the same entry starts on a read-only directory. Before, the grant of an existing user stayed in force in memory and the failed write was logged as `already exists with a different password`.

An entry the server cannot parse (no `[…]` block, a `[` or a `{` never closed) stops the list there: that entry and the ones after it are not loaded, and the server logs a WARNING naming the malformed entry. Since 26.10.2 the WARNING quotes only that entry, with its passwords masked the same way the settings reports mask them (`Universe[albert:*]`, and from the first `:` on for an entry written without brackets), and an unclosed `[` is reported this way instead of aborting the server start.

A database entry may be followed by `{restore:<URL>}`, which restores that database from a backup archive at start-up, replacing any copy already on disk (e.g. `-Darcadedb.server.defaultDatabases=Universe[albert:einstein]{restore:/tmp/universe.zip}`).

In a cluster, `restore:` acts on the node that is starting and on that node only: the other nodes keep their current copy of the database and are not told anything. This is intentional - it is per-node boot configuration, so a rolling restart does not wipe the database cluster-wide - but it does mean a node can come back with different content than its peers. The server logs a WARNING naming the database whenever this replaces a replicated one, so the divergence is visible in the log.

A client can reach the server while it is still booting and start a backup, restore or import of the same database. The `restore:` command waits for that operation to finish instead of refusing at once, for up to `arcadedb.server.startupRestoreSlotWaitMs` (60000 ms by default). While it waits the server logs at INFO:

```
The startup 'restore:' command for database 'Universe' is waiting up to 60000ms for
<operation> that was running on it to finish
```

If the other operation is still running when the wait runs out, the command refuses, the database

on disk is left untouched, and the server boot fails. Set the value to `0` (or a negative number) to refuse immediately, as earlier versions did.

Once the server is started, multiple clients can be connected to the server by using one of the supported protocols:

- HTTP/JSON
- Postgres
- MongoDB
- Redis

8.7.5. Closing idle databases

(Since v26.11.1.) A server that hosts one database per tenant can hold thousands of them, and every open database keeps its data files, its caches and, once written, a WAL file. The server can close the databases nobody is using and reopen each one on its next request, so the open files and the heap follow the active tenants only. Closing is off by default: set `arcadedb.server.databaseIdleTimeout`, `arcadedb.server.maxOpenDatabases`, or both.

```
$ bin/server.sh -Darcadedb.server.databaseLoadAtStartup=false \
                -Darcadedb.server.databaseIdleTimeout=1800000 \
                -Darcadedb.server.maxOpenDatabases=500 \
                -Darcadedb.server.databaseIdleExclude=catalog,audit
```

With the example above the server opens no database at start, closes a database nobody has used for 30 minutes, keeps at most 500 open at once, and never closes `catalog` and `audit`.

Setting	Meaning	Default
`arcadedb.server.databaseIdleTimeout`	Closes a database no client has used for this long, in milliseconds. `0` means never	`0`
`arcadedb.server.databaseIdleCheckInterval`	How often, in milliseconds, the server looks for databases to close. Idleness is measured at this resolution	`60000`
`arcadedb.server.databaseIdleMaxClosesPerCheck`	The most databases one check closes, idle and cap closes together, so a burst of databases going idle at once is closed over several checks. `0` means no bound	`10`
`arcadedb.server.databaseIdleExclude`	Comma-separated names of the databases never closed	empty
`arcadedb.server.maxOpenDatabases`	The most databases kept open at once. Beyond it, each check closes the least recently used databases that are not in use, idle timeout or not. `0` means no cap	`0`

The settings are read at each check, so a change made with `SET SERVER SETTING` or the `set server`

`setting` server command applies at the next check, without a restart.

What counts as used. Any query, command, transaction, record operation, scan or count, from any protocol. The server samples the engine's own counters of each database, so no protocol has to report anything. A database is never closed, however long it has been quiet, while:

- a transaction is open on it, on any thread: a Postgres or Bolt connection, a gRPC client transaction, an HTTP session opened with `/begin` or a `/ws` insert session;
- a statement is running on it, or asynchronous work is queued;
- a backup, restore, import or export of it is in progress;
- it is replicated (HA), or listed in `arcadedb.server.databaseIdleExclude`.

An open connection without a transaction does not keep a database open: a connection pool would otherwise keep every tenant open for good. The next request on that connection reopens the database. A database in use is never closed for the cap either, so the cap can be exceeded while more databases than that are busy.

Reopening. A database closed for idleness or for the cap keeps its files, and is closed without the mark that the `close database` server command sets. Unlike a database closed with that command, which stays closed until `open database`, it is reopened transparently by the next request of any protocol: HTTP, gRPC, PostgreSQL, Bolt, Redis, MongoDB, Gremlin and MCP. A request that resolved the database a moment before it was closed fails with a retryable error (`DatabaseIsReopeningException`, a retryable `503` over HTTP), and its retry finds the database reopened. The database listings include it while it is closed, see Databases on disk that are not open.

Metrics. Two counters are exported with the other server meters (see Cloud Observability): `arcadedb.server.databases.idle_closed`, tagged `reason idle` or `cap`, counts the databases closed, and `arcadedb.server.databases.reopened` counts the databases a request opened again after such a close. A reopen rate close to the close rate means the timeout is shorter than the gap between a tenant's requests. `arcadedb.server.databases.idle_close_failed` counts the closes that failed (the server log has the cause): such a database is deregistered anyway, and the next request opens it from its files again.

Automatic backups. The automatic backup schedule of a database follows the databases the server has open, so a database closed for idleness or for the cap is not backed up while it stays closed. It does not change while closed, but the writes made between its last backup and the close are covered only by the first backup after it is reopened. When every write must reach a backup within a bounded time, keep the idle timeout longer than the backup period, or list those databases in `arcadedb.server.databaseIdleExclude`.

Databases on disk that are not open

(Since v26.11.1.) The database listings and existence checks include the databases that are on disk but not open, those closed for idleness and those not opened at startup because `arcadedb.server.databaseLoadAtStartup` is `false`, because every protocol opens them on first use:

- HTTP `GET /api/v1/databases`, `GET /api/v1/exists/{database}` and the `list databases` server

command;

- the gRPC admin RPCs `ListDatabases`, `ExistsDatabase` and `GetDatabaseInfo`, and `CreateDatabase` with `if_not_exists` and `DropDatabase` with `if_exists`;
- Bolt `SHOW DATABASES`, and the database a Bolt connection falls back on when it names none and `arcadedb.bolt.defaultDatabase` is not set;
- the MCP `list_databases` tool and the schema resources;
- Gremlin traversal sources: the traversal opens a database that is not open, and the `g` alias keeps naming the same database whatever is open;
- the `drop database` server command, which drops a database that is not open.

A database closed with `close database`, being restored or being dropped is still not listed.

Before v26.11.1, with `arcadedb.server.databaseLoadAtStartup=false`, these surfaces answered as if a database not opened yet did not exist: it was missing from the listings and an existence check answered `false` until a request opened it.

8.7.6. Logging

The log files are created in the folder `./log` with the filenames `arcadedb.log.X`, where `X` is a number between 0 to 9, set up for log rotate. The current log file has the number 0, and is rotated based on server starts or file size.

By default ArcadeDB does not log debug messages into the console and file. You can change this settings by editing the file `config/arcadedb-log.properties`. The file is a standard logging configuration file.

The default configuration is the following.

```
1  handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler
2  .level = INFO
3  com.arcadedb.level = INFO
4  java.util.logging.ConsoleHandler.level = INFO
5  java.util.logging.ConsoleHandler.formatter = com.arcadedb.utility.AnsiLogFormatter
6  java.util.logging.FileHandler.level = INFO
7  java.util.logging.FileHandler.pattern=./log/arcadedb.log
8  java.util.logging.FileHandler.formatter = com.arcadedb.log.LogFormatter
9  java.util.logging.FileHandler.limit=100000000
10 java.util.logging.FileHandler.count=10
```

Where:

- Line 1 contains 2 loggers, the console and the file. This means logs will be written in both console (process output) and configured file (see line 7)
- Line 2 sets INFO (information) as the default logging level for all the Java classes between `FINER`, `FINE`, `INFO`, `WARNING`, `SEVERE`

- Line 3 is as (line 2) but sets the level for ArcadeDB package only `SEVERE`
- Line 4 sets the minimum level the console logger filters the log file (below `INFO` level will be discarded)
- Line 5 sets the formatter used for the console. The `AnsiLogFormatter` supports ANSI color codes
- Line 6 sets the minimum level the file logger filters the log file (below `INFO` level will be discarded)
- Line 7 sets the path where to write the log file (the file will have a counter suffix, see line 10)
- Line 8 sets the formatter used for the file
- Line 9 sets the maximum file size for the log, before creating a new file. By default it is 100MB
- Line 10 sets the number of files to keep in the directory. By default it is 10. This means that after the 10th file, the oldest file will be removed

If you're running ArcadeDB in embedded mode, make sure you're using the logging setting by specifying the `arcadedb-log.properties` file at JVM startup:

```
$ java ... -Djava.util.logging.config.file=$ARCADEDB_HOME/config/arcadedb-
log.properties ...
```

You can also use your own configuration for logging. In this case replace the path above with your own file.

8.7.7. Server Plugins (Extend The Server)

You can extend ArcadeDB server by creating custom plugins. A plugin is a Java class that implements the interface `com.arcadedb.server.ServerPlugin`:

```
public interface ServerPlugin {
  void startService();

  default void stopService() {
  }

  default void configure(ArcadeDBServer arcadeDBServer, ContextConfiguration
configuration) {
  }

  default void registerAPI(final HttpServer httpServer, final PathHandler routes) {
  }
}
```

(Since v26.11.1) A plugin should declare its HTTP routes through the overload `registerAPI(HttpServer, PluginRoutes)`, which takes each route together with its HTTP methods: a request with any other method answers `405` with an `Allow` header, before authentication. The `registerAPI(HttpServer, PathHandler)` shown above is deprecated for removal: a plugin that still

implements it keeps working and answers every method on its routes, and the server logs a WARNING at startup that names the plugin. Implement only one of the two overloads: a plugin that implements both gets both called, with a separate WARNING, and a path registered in both is registered twice, the later raw registration winning.

Once registered, the plugin (see below), ArcadeDB Server will instantiate your plugin class and will call the method `configure()` passing the server configuration. At startup of the server, the `startService()` method will be invoked. When the server is shut down, the `stopService()` will be invoked where you can free any resources used by the plugin. The method `registerAPI()`, if implemented, will be invoked when the HTTP server is initializing where one's own HTTP commands can be registered. For more information about how to create custom HTTP commands, look at Custom HTTP commands.

Example:

```
package com.yourpackage;

public class MyPlugin implements ServerPlugin {
  @Override
  public void startService() {
    System.out.println( "Plugin started" );
  }

  @Override
  public void stopService() {
    System.out.println( "Plugin halted" );
  }

  @Override
  default void configure(ArcadeDBServer arcadeDBServer, ContextConfiguration
configuration) {
    System.out.println( "Plugin configured" );
  }

  @Override
  default void registerAPI(final HttpServer httpServer, final PathHandler routes) {
    System.out.println( "Registering HTTP commands" );
  }
}
```

To register your plugin, register the name and add your class (with full package name) in `arcadedb.server.plugins` setting:

Example:

```
$ java ... -Darcadedb.server.plugins=MyPlugin:com.yourpackage.MyPlugin ...
```

In case of multiple plugins, use a comma (`,`) to separate them.

A common use of a Server Plugin is to register **custom Java functions** on a standalone server. Since there is no auto-discovery, drop your JAR in `lib/` (or `lib/plugins/`), call `database.getSchema().registerFunctionLibrary(…)` from the plugin's `startService()` method, and enable the plugin via `arcadedb.server.plugins`. See User Functions in Java for a complete example.

8.7.8. Metrics

The ArcadeDB server can collect, log and publish metrics. To activate the collection of metrics use the setting:

```
$ ... -Darcadedb.serverMetrics=true
```

To log the metrics to the standard output use the setting:

```
$ ... -Darcadedb.serverMetrics.logging=true
```

To publish the metrics as Prometheus via HTTP, add the plugin:

```
$ ... -Darcadedb.server.plugins
="Prometheus:com.arcadedb.metrics.prometheus.PrometheusMetricsPlugin"
```

Then, under `http://localhost:2480/prometheus` (or the respective ArcadeDB host) the metrics can be requested given server credentials. For details about the response format see the Prometheus docs.

8.8. Changing Settings

To change the default value of a setting, always put `arcadedb.` as a prefix. Example:

```
$ java -Darcadedb.dumpConfigAtStartup=true ...
```

To change the same setting via Java code:

```
GlobalConfiguration.findByKey("arcadedb.dumpConfigAtStartup").setValue(true);
```

Check the Appendix for all the available settings.

8.8.1. Environment Variables

The server script parses a set of environment variables which are summarized below:

`JAVA_HOME`	JVM location

JAVA_OPTS	JVM options
ARCADEDB_SETTINGS	ArcadeDB settings, as -Darcadedb.* arguments
ARCADEDB_HOME	ArcadeDB location
ARCADEDB_PID	arcadedb.pid file location
ARCADEDB_OPTS_MEMORY	JVM memory options
ARCADEDB_OPTS_GC	JVM garbage collector options
ARCADEDB_OPTS_HEADERS	JVM object header options (server.sh only)

server.sh enables compact object headers (JEP 519, -XX:+UseCompactObjectHeaders), which shrink every object header and cut allocation, when the JVM accepts the flag. It asks the JVM first (java -XX:+UseCompactObjectHeaders -version) and adds the flag only when that succeeds, so Java 25 and later get it while Java 21 still starts without it. The flag lives in ARCADEDB_OPTS_HEADERS, so setting JAVA_OPTS does not lose it. Setting ARCADEDB_OPTS_HEADERS skips the probe and uses your value instead, and an empty value opts out:

```
$ export ARCADEDB_OPTS_HEADERS=""
$ bin/server.sh
```

Use ARCADEDB_SETTINGS for ArcadeDB's own settings and leave JAVA_OPTS for JVM flags. They are separate variables so that setting one never discards the other: this matters in containers, where the image supplies its own values for ARCADEDB_OPTS_GC and ARCADEDB_OPTS_MEMORY and an environment variable is replaced rather than appended to.

```
$ export ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata"
$ bin/server.sh
```

For default values see the server.sh and server.bat scripts.

8.8.2. RAM Configuration

The ArcadeDB server, by default, uses a dynamic allocation for the used RAM. Sometimes you want to limit this to a specific amount. You can define the environment variable ARCADEDB_OPTS_MEMORY to tune the JVM settings for the usage of the RAM.

Example to use 800M fixed RAM for ArcadeDB server:

```
$ export ARCADEDB_OPTS_MEMORY="-Xms800M -Xmx800M"
$ bin/server.sh
```

ArcadeDB can run with as little as 16M for RAM. In case you're running ArcadeDB with less than 800M of RAM, you should set the "low-ram" as profile:

```
$ export ARCADEDB_OPTS_MEMORY="-Xms128M -Xmx128M"
$ bin/server.sh -Darcadedb.profile=low-ram
```

Setting a profile is like executing a macro that changes multiple settings at once. You can tune them individually, check Settings.

The profile applies all of its settings when it is given as a `-D` system property or as an environment variable, the page cache (`arcadedb.maxPageRAM`, 16MB for `low-ram`) included. A setting given explicitly next to the profile wins over the profile's value: `-Darcadedb.profile=low-ram -Darcadedb.maxPageRAM=64` keeps a page cache of 64MB and the rest of the `low-ram` profile *(since v26.11.1)*.

In case of memory latency problems under Linux systems, the following JVM setting can improve performance:

```
$ export ARCADEDB_OPTS_MEMORY="-XX:+PerfDisableSharedMem"
```

for more information, see https://www.evanjones.ca/jvm-mmap-pause.html

The Java heap memory is by default configured for desktop use; for custom containers, memory configuration can be adapted by:

```
$ export ARCADEDB_OPTS_MEMORY="-XX:InitialRAMPercentage=50.0
-XX:MaxRAMPercentage=75.0"
```

More information about Java memory configuration see this article.

8.9. High Availability

ArcadeDB is designed for continuous operation. Its high availability (HA) architecture ensures that your database remains accessible even when individual servers fail, providing both fault tolerance and horizontal read scalability.

This page explains the concepts behind ArcadeDB's HA architecture. For step-by-step setup instructions, see HA Configuration.

ArcadeDB's HA is powered by Apache Ratis, a battle-tested implementation of the Raft consensus protocol also used by Apache Ozone, Apache IoTDB, and Alluxio. The change is transparent to clients: the HTTP API, drivers, and query languages are unchanged.

8.9.1. Leader-Replica Replication Model

ArcadeDB uses a leader-replica replication model. At any point in time, a cluster has exactly one **leader** server and zero or more **replica** servers.

- The **leader** coordinates all write operations and distributes changes to replicas through the Raft log.
- **Replicas** serve read requests (queries) independently, allowing the cluster to scale read throughput horizontally.

Clients can connect to any server in the cluster. If a client sends a write request to a replica, the replica transparently forwards it to the leader over HTTP. Applications do not need to distinguish between leader and replica servers — the cluster handles routing internally.

Inside a transaction the client opened itself (a PostgreSQL `BEGIN`, an HTTP `/begin` session, a Bolt `BEGIN`, a gRPC `BeginTransaction`), a replica runs the writes in that transaction instead of forwarding them one by one: `ROLLBACK` undoes them and `COMMIT` replicates the whole block atomically through the Raft log. If the replica was behind the leader, the commit fails with a retryable conflict. Schema changes are always executed by the leader, since they are not transactional, and a `sqlscript` that mixes schema changes and writes is refused inside a transaction on a replica. A write that names a label or a relationship type with no type yet (an openCypher `CREATE` or `MERGE`) works in such a transaction too: the replica asks the leader to create the type, waits until it has applied it and carries on, so the data stays in the transaction while the type, as every schema change, outlives a `ROLLBACK`. If another schema change was applied while the transaction was open, that statement fails with a retryable conflict and the retry goes through.

8.9.2. Server Roles and Election

Every server in the cluster operates in one of two roles:

LEADER

The single server responsible for accepting and coordinating all write operations. Only one leader exists at any time.

REPLICA

A server that maintains a copy of the data and serves read requests. Replicas receive changes from the leader and can be promoted to leader if the current leader fails.

When a server starts, it joins the cluster and participates in a Raft election. Raft guarantees that the cluster agrees on exactly one leader at all times. Apache Ratis implements the full Raft protocol, including **pre-vote** (prevents disrupted elections from partitioned nodes), **leader lease**, and **parallel voting** for fast convergence.

If the cluster already has a healthy leader, the new server simply joins as a replica and catches up via the Raft log or, if it has fallen too far behind, a full snapshot download. If no leader exists — for example, during initial cluster formation or after a leader failure — an election determines which server becomes the new leader.

8.9.3. Replication and the Raft Log

The leader replicates changes to replicas through the Raft log, which combines a write-ahead log with distributed consensus. Each server maintains its own local copy of the Raft log.

When the leader processes a write operation, ArcadeDB uses a **replicate-first, commit-after** three-phase commit:

1. **Phase 1** — the leader captures the write-ahead log (WAL) pages produced by the transaction.
2. **Phase 2** — the captured payload is appended to the Raft log and replicated to followers. The commit is acknowledged only after a quorum of nodes has persisted the entry.
3. **Phase 3** — the leader applies the pages locally and returns success to the client.

If replication fails at Phase 2, no local writes are applied: the transaction is rolled back and no divergence can occur between the leader and the followers.

Multiple concurrent transactions are automatically batched into a single Raft round-trip via **group commit**, dramatically increasing throughput under concurrent load.

Schema DDL takes a different path: each schema change is replicated as its own entry type, carrying the schema document rather than WAL pages. That entry can carry just a delta against the schema the followers already hold, and a schema-only SQL script is replicated as a single entry. See Schema replication.

8.9.4. Write Quorum

ArcadeDB uses a configurable **write quorum** to control the trade-off between durability and availability. The quorum defines how many servers must acknowledge a write before it is considered committed.

The default quorum is `MAJORITY` — more than half of the servers in the cluster must confirm the write. This is the standard Raft quorum and provides a strong balance between durability and availability: data is stored on a majority of nodes while tolerating the failure of any minority.

Two quorum modes are supported:

`majority` **(default)**

A standard Raft quorum. Writes are committed as soon as a majority of peers acknowledge, so the cluster keeps serving writes as long as a majority is alive.

`all`

Every configured peer must acknowledge the write. Strongest durability, but a single unavailable peer stops writes.

The legacy values `none`, `one`, `two`, `three` are **not supported** and will cause a startup error. They were part of the old custom HA protocol; with Raft only `majority` and `all` are meaningful. If you were using a granular quorum value, update your configuration before upgrading.

If the quorum cannot be met — for example, because too many peers are unavailable — the transaction is rolled back and an error is returned to the client. This prevents partially committed writes.

For more on how transactions interact with replication, see Transactions. For quorum and other cluster settings, see Server Settings.

8.9.5. Read Consistency

When a client sends a read request to a replica, ArcadeDB supports three consistency levels that trade latency for freshness:

`eventual`

The replica reads locally without waiting. Fastest, but may return slightly stale data.

`read_your_writes` **(default)**

The replica waits until the Raft log index corresponding to the client's most recent write has been applied locally before serving the read. This ensures the client always observes its own writes, even when those writes went to the leader and the read goes to a replica.

`linearizable`

The replica issues a Ratis `ReadIndex` request to the leader and waits until its local applied index reaches the leader's committed index at the time of the request. This is the strongest guarantee and survives leader changes without serving stale data, at the cost of higher latency.

The default is set via `arcadedb.ha.readConsistency`. Clients can also override it per request with the `X-ArcadeDB-Read-Consistency` HTTP header. For read-your-writes, the client takes the commit index the server returned in the `X-ArcadeDB-Commit-Index` response header and sends it back on the next read as `X-ArcadeDB-Read-After`, so the replica waits only for that index. Every response carries the bookmark, including streamed query responses and error responses.

8.9.6. Automatic Failover

ArcadeDB handles leader failure automatically. When replicas stop receiving heartbeats from the leader, they initiate a new election using the Raft protocol. A replica with an up-to-date log is elected as the new leader, and the cluster resumes normal operation.

This failover process is transparent to clients: after a brief interruption during the election, clients reconnect (or are proxied) to the new leader and continue operating without manual intervention.

Common causes of leader unavailability include process termination, server shutdown or reboot, and network partitions that isolate the leader from the rest of the cluster.

8.9.7. Split-Brain Protection

A **split-brain** occurs when a network partition divides a cluster into two or more groups, each believing it should operate independently. This can lead to conflicting writes and data divergence.

ArcadeDB prevents split-brain scenarios through the Raft majority quorum requirement. Because a

leader must receive acknowledgment from a majority of servers to commit writes, at most one partition can ever hold a majority. Any minority partition cannot elect a leader or commit writes, so it effectively becomes read-only until the network heals. Pre-vote further prevents a partitioned node from forcing a disruptive election when it regains connectivity.

This design guarantees that the cluster never produces conflicting committed data, even during network failures.

8.9.8. Snapshots and Catch-Up

As the Raft log grows, ArcadeDB periodically takes a snapshot of the database state. The snapshot allows old log segments to be compacted, preventing unbounded disk growth.

When a replica falls too far behind the leader (for example, after a long network partition), ArcadeDB automatically transfers a fresh snapshot: the follower downloads a ZIP of the database from the leader over HTTP and installs it atomically. Once the snapshot is installed, the follower resumes normal log-based replication from the snapshot index.

8.9.9. Cluster Bootstrap from Pre-Staged Data

When a multi-node cluster forms for the first time on top of a pre-imported database, ArcadeDB skips the leader-streamed snapshot phase entirely. Every peer reports the fingerprint and `lastTxId` of its local database; the peer with the highest `lastTxId` becomes the bootstrap source via Raft leadership transfer; peers whose fingerprint matches bootstrap locally with zero bytes transferred. This makes "import the dataset once, then scale to N pods" a seconds-not-hours operation, and the cluster forms with everyone already at the same state.

The bootstrap path is gated on every peer's Raft log being empty, so subsequent cluster restarts continue to use the regular log replay and snapshot paths. For the operator-facing workflow and configuration flags, see Offline Cluster Bootstrap.

8.9.10. Dynamic Membership

Cluster membership can be changed at runtime without restarting the cluster. Operators can add peers, remove peers, transfer leadership, or have a node gracefully leave the cluster through the cluster REST API. On Kubernetes, scaling a StatefulSet up or down automatically adds and removes peers.

8.9.11. Cluster Discovery and Configuration

Servers discover each other through a configured list of peers (`arcadedb.ha.serverList`). Each entry can use the readable object form `[name@]host:{raft:2434,http:2480,https:2490,priority:10}` (recommended) or the positional form `[name@]host:raftPort:httpPort[:priority[:httpsPort]]`, where the Raft gRPC port (default `2434`) is used for consensus, the HTTP port for command forwarding, and the optional HTTPS port for encrypted peer-to-peer transfers when TLS is enabled. The optional `name@` prefix lets operators give each peer a stable, human-readable name (for example, `frankfurt`, `london`, `nyc`) shown in logs and Studio. See Server List Entry Format for details.

You can run multiple independent clusters on the same network by assigning each cluster a unique

name via `arcadedb.ha.clusterName`. Inter-node communication is authenticated with a shared **cluster token** automatically derived from the cluster name and the root password.

For detailed configuration options and deployment instructions, see HA Configuration.

8.9.12. Continuous Correctness Testing with Jepsen

Distributed systems are easy to claim correct and hard to prove correct. To know how ArcadeDB actually behaves under partitions, process kills, clock skew, and simulated power loss — rather than how we hope it behaves — the team maintains a public Jepsen test suite at ArcadeData/arcadedb-jepsen.

Jepsen is the de-facto framework for testing distributed systems under faults. It pairs aggressive failure injection with two model checkers — **Knossos** for linearizability and **Elle** for transactional isolation — that build a history of every operation the cluster acknowledged and then look for any execution that no correct system could have produced. Both checkers are model-driven, so a system can't pass them by being merely fast or quiet under load.

ArcadeDB is **not** officially Jepsen-certified — that requires an independent engagement with Jepsen LLC. The tests in this section are a self-hosted suite using the open-source Jepsen framework. We publish them in full so the community can review, reproduce, and extend them.

What gets tested

The suite combines **six workloads** with **eight fault models** (nemeses), so the cluster is exercised the way users actually break it in production:

Workload	What it verifies
`bank`	ACID conservation across concurrent transfers between accounts; the total balance must stay constant under any failure schedule.
`set`	No acknowledged write is ever lost; every successfully-inserted element survives partitions, kills, and replays.
`elle`	Multi-key transaction isolation. The Elle checker rejects G0, G1a, G1b, G2, and lost-update anomalies via dependency-graph cycle detection.
`register`	Single-key linearizability when reading from the leader.
`register-follower`	Linearizability for follower reads via Raft `ReadIndex`.
`register-bookmark`	Read-your-writes consistency through commit-index bookmarks (the same mechanism documented in Read Consistency).

Nemesis	What it injects
`partition`	Network partitions that isolate one or more nodes from the rest of the cluster.
`kill`	Hard process termination of randomly chosen servers.

Nemesis	What it injects
`pause`	`SIGSTOP` / `SIGCONT` pairs that freeze a server's process for seconds at a time.
`clock`	Bidirectional clock skew on individual nodes.
`lazyfs`	LazyFS-backed simulated power loss — drops un-fsynced writes to test recovery from crashes that lose the OS page cache.
`all`	Random combinations of `partition` + `kill` + `pause`.
`all+clock`	`all` plus simultaneous clock skew.
`all+lazyfs`	`all` plus simulated power loss.

The full matrix runs in Docker against a five-node cluster (`n1`–`n5`) and is reproducible end-to-end from the repository's `run-all-tests.sh`.

Current status

At the time of writing, the suite reports **all tests passing** — leader linearizability, follower linearizability via `ReadIndex`, ACID conservation, replication completeness, and read-your-writes — across every nemesis combination, including the LazyFS power-loss workloads added most recently. See the launch announcement for the methodology and original results, and the repository for the up-to-date matrix.

Concretely, this means that under every failure pattern we exercise:

- every write the cluster acknowledged remains durable after recovery,
- concurrent transactions remain ACID — Elle finds no isolation anomaly,
- single-key operations are linearizable on the leader and on followers using `ReadIndex`,
- clients reading their own writes never observe regressed state, even when their reads land on a different replica than the write.

What's not (yet) tested

Honesty matters more than green checkmarks. The suite does **not** yet cover:

- multi-hour or multi-day soak runs that surface slow leaks and pathological GC interactions,
- Byzantine-fsync scenarios (a filesystem that lies about durability beyond what LazyFS simulates),
- geo-distributed clusters with sustained tens-of-milliseconds latency between nodes,
- every imaginable compounded worst case — for example, an expired leader lease **plus** clock skew **plus** an active partition all happening at once.

These are explicitly tracked as future work. Pull requests for additional workloads, tighter checkers, or longer-duration runs are very welcome.

Mission

The reason the project exists is straightforward: we want to know what ArcadeDB actually does

under failure before we ask anyone else to trust it. Jepsen is the highest-leverage way the industry has found to answer that question, and we run it in public so the answer is verifiable rather than assertive. If you find a workload that breaks, please open an issue at arcadedb-jepsen/issues — that is exactly what the suite is for.

8.10. Configuring High Availability

ArcadeDB supports a High Availability (HA) mode where multiple servers share the same databases via Raft-based replication. All servers of a cluster serve the same set of databases.

HA is powered by Apache Ratis, a production-grade implementation of the Raft consensus protocol. The old custom replication protocol has been removed. For the underlying concepts, see High Availability Concepts.

8.10.1. Quick Start

To start an ArcadeDB server with HA enabled, at minimum you need to:

1. Set `arcadedb.ha.enabled` to `true`.
2. Define the list of peers in the cluster via `arcadedb.ha.serverList` (if you are deploying on Kubernetes, see Kubernetes). The value is a comma-separated list of peer entries. Each entry can use the readable **object form** `host:{raft:2434,http:2480,https:2490,priority:10}` (recommended) or the **positional form** `host:raftPort:httpPort[:priority[:httpsPort]]`; see Server List Entry Format. The optional `name@` prefix is described in Named Peers.
3. Optionally set the local server name via `arcadedb.server.name`. Each node must have a unique name. If not specified, the default is `ArcadeDB_0`.

Example starting a 3-node cluster on a single host (different ports):

```
$ bin/server.sh -Darcadedb.ha.enabled=true \
                -Darcadedb.server.name=node1 \
                -Darcadedb.ha.serverList
=localhost:2434:2480,localhost:2435:2481,localhost:2436:2482
```

The Raft gRPC port is `2434` by default and is configurable via `arcadedb.ha.raftPort`. The HTTP port in each entry is required so that replicas can forward non-idempotent requests to the leader over HTTP.

The cluster name is `arcadedb` by default; set `arcadedb.ha.clusterName=<name>` to run multiple independent clusters on the same network.

8.10.2. Server List Entry Format

Each comma-separated entry in `arcadedb.ha.serverList` can be written in either of two interchangeable forms. The two forms may be freely mixed in the same list, and both accept the optional `name@` prefix (see Named Peers).

Object form (recommended)

A brace-delimited set of named fields, which avoids the positional ambiguity of long colon-separated entries:

```
[name@]host:{raft:2434,http:2480,https:2490,priority:10}
```

Fields are unordered and all optional, with these defaults:

Field	Meaning
`raft`	Raft gRPC consensus port. Defaults to `arcadedb.ha.raftPort` (`2434`) when omitted.
`http`	HTTP port, used by replicas to forward non-idempotent requests to the leader. Omit if not needed.
`https`	HTTPS port, used for **encrypted** peer-to-peer transfers (e.g. snapshot download) when `arcadedb.ssl.enabled=true`. See TLS/SSL.
`bolt`	Client-reachable BOLT port, advertised in the BOLT routing table so `neo4j://` drivers find the leader and the followers. See Client Routing Ports.
`grpc`	Client-reachable gRPC port, advertised when a gRPC call refused on a follower names the leader to retry against. See Client Routing Ports.
`priority`	Leader-election preference (integer, default `0`). The node with the highest priority is preferred as leader.

`bolt` and `grpc` have no positional equivalent - an entry declaring either must use the object form.

Positional form

A colon-separated entry where each field is identified by position:

```
[name@]host:raftPort:httpPort:priority:httpsPort
```

Trailing fields are optional, so `host`, `host:raftPort`, `host:raftPort:httpPort`, and `host:raftPort:httpPort:priority` are all valid. The `httpsPort` is the fifth field and therefore requires `priority` to be present.

Prefer the object form when an entry specifies a priority or an HTTPS port - it is far easier to read `db1:{raft:2434,http:2480,https:2490,priority:10}` than `db1:2434:2480:10:2490`.

IPv6 hosts

In both forms an IPv6 host is written in brackets, and keeps them in every address derived from it: `[fd00::1]:2434:2480` or `[fd00::1]:{raft:2434,http:2480}`. An unbracketed IPv6 literal (`fd00::1:2434`) is refused, because its port cannot be told apart from the address. The Raft peer and the inbound gRPC peer allowlist read the host of an entry the same way, so both resolve

`fd00::1` from either form. *(Since v26.11.1: the positional form used to split a bracketed host on its colons, giving the Raft peer a broken address while the allowlist resolved the right host.)*

Port validation

Every port of an entry (`raft`, `http`, `https`, `bolt`, `grpc`) must be a number between 1 and 65535, and a node refuses to start on an entry that breaks this, naming the field and the entry: for example `Invalid raft port value 'abc' in peer address 'db1:abc'`. An empty field, such as a trailing `:`, is refused too. *(Since v26.11.1: a non-numeric or out-of-range port in the positional form was accepted, and failed later, when the peer was dialled, with a message naming the host instead of the port.)*

The following two entries are equivalent:

```
# Object form (recommended)
-Darcadedb.ha.serverList=db1:{raft:2434,http:2480,https:2490,priority:10}

# Positional form
-Darcadedb.ha.serverList=db1:2434:2480:10:2490
```

8.10.3. Client Routing Ports (`bolt`, `grpc`)

A client protocol's routing view tells a driver which node to send writes to and which nodes it may read from, so it deals in the ports **clients** dial - never the Raft port the cluster uses to talk to itself, and not necessarily the HTTP port either.

Declare `bolt` and `grpc` whenever your nodes do not all listen on the same port for that protocol. With the field omitted, a peer's address for that protocol is derived as *that peer's host* plus *this node's own port*. That is correct only for a homogeneous deployment where every node listens on the same port - a Kubernetes StatefulSet, for example - and a one-time WARNING per protocol is logged whenever the fallback is used.

```
# Three nodes on one host: same host, different ports - what the fallback cannot
express
-Darcadedb.ha.serverList=n0@localhost:{raft:2434,http:2480,bolt:7687,grpc:50051
},n1@localhost:{raft:2435,http:2481,bolt:7688,grpc:50052},n2@localhost:{raft:2436,http
:2482,bolt:7689,grpc:50053}
```

On a cluster like that one, leaving the ports undeclared makes every peer derive to the **same** address - and an address two peers both claim identifies neither of them, since two listening sockets cannot share one `host:port`. Rather than advertise a leader address that is really the address of the node answering (which would send a self-redirecting client straight back to the node that just refused it), ArcadeDB advertises nothing for that protocol: BOLT then offers this node as READ and ROUTE but never as writer, and a refused gRPC call falls back to naming the leader's HTTP address. A WARNING naming the field to declare is logged once per protocol.

A declared address always wins over a derived one. Declaring the ports of the nodes that differ is enough; the rest may keep deriving.

A peer that joined the Raft group at runtime (a Kubernetes scale-up beyond `arcadedb.ha.serverList`, `connect cluster`, `POST /api/v1/cluster/peer`) is part of the routing view like a listed one: the Bolt `ROUTE` table offers it as a reader, and as the writer while it leads, a gRPC call refused on a follower names it when it leads, and its address is derived the same way from its Raft host. *(Since v26.11.1: such a peer was replicated to but never routed to, and the workaround was to pre-size `arcadedb.ha.serverList` to the largest node count.)*

8.10.4. Peer-to-Peer Endpoints (`http`, `https`)

The same derive-from-this-node's-port fallback applies to the endpoints the **nodes** use to reach each other: a follower downloading a snapshot from the leader, a cluster verify comparing checksums with a peer, and the database presence matrix (`GET /api/v1/cluster?presence=true`). When two peers resolve to one `host:port`, that address identifies at most one of them and nothing can say which, so ArcadeDB **withholds** it rather than guessing: the operation is refused and reported as refused - a peer comes back `ERROR` / "not verified" from a verify, and unreachable from a presence matrix - instead of quietly answering for the wrong node.

The presence matrix lists every server the node knows: the server list plus any server that joined at runtime (a Kubernetes scale-up, `connect cluster`). Each one is either a column of the matrix or listed in `unreachable`, and a declared server the cluster no longer counts as a member is also listed in `notInConfiguration`, since a database it lacks is not a replication gap. *(Since v26.11.1: before, a pod that joined at runtime was missing from the matrix.)*

The refusal is visible in three places:

- a one-time WARNING per protocol, naming the peers that could not be told apart, the address they share, and the field to declare;
- `httpAddressAmbiguous: true` next to that peer's `httpAddress` in `GET /api/v1/cluster`, and a warning line on the node's card in the Studio Cluster dashboard;
- the error text of whichever operation was refused.

`http` and `https` are checked independently, because they are read from independent fields with independent fallbacks. A cluster that declares distinct `http` ports and omits the `https` ones passes the HTTP check with every peer's HTTPS endpoint still collapsed onto this node's own; the encrypted transfer then falls back to the (guarded) plain-HTTP endpoint.

One request is exempt: the **capability check**, in which the leader asks each peer which replication formats its build understands so it can use a compact one only when every node can read it. That call changes nothing on the peer and its reply says who answered, so ArcadeDB dials the shared address anyway and credits the answer to the node that identified itself. A cluster that has never declared its `http` ports therefore still gets the compact formats, instead of falling back to the verbose ones forever with nothing to explain why.

On the common single-host layout where every node shares an address and the Raft and HTTP ports move in step (`2434/2480`, `2435/2481`, ...), the capability check derives each peer's HTTP port from that same offset, so it reaches the peers the shared-address fallback cannot name. The derived port is used for the capability check only - never for a snapshot download, a cluster verify or a

forwarded write, which keep refusing a shared address.

A peer that still did not answer carries `capabilitiesUnknownReason` next to its entry in `GET /api/v1/cluster`, saying whether it is unreachable, running an older build, or not identified by any address of its own - the last being the one you fix by declaring its `http` port.

Every node also **learns** the HTTP address of a peer it cannot reach where it resolves it, the same way on every node. *(Since v26.10.2)* Each node states its own HTTP port and address in the capability check it sends to the others, and each answer carries the addresses the answering node holds for the other members. Neither is trusted on its own: an address is used only after the peer it names answers there under its own id. From then on that address serves every peer-to-peer call (snapshot downloads, verifies, forwarded writes) and is passed on to the other nodes. In practice, a peer admitted without an `http` port (`connect cluster host:raftPort`, `POST /api/v1/cluster/peer`), or declared on one node only, becomes reachable from every node within a capability round or two, whatever the port layout. An address this node's own `arcadedb.ha.serverList` declares is never replaced. Only the **port** is learnt: an offered address is accepted only on the peer's own Raft host. On a cluster with `arcadedb.ssl.enabled`, offered addresses are not probed at all, because the probe carries the cluster token and would send it over plain HTTP.

Before v26.10.2, the node that served an admission without an `http` port wrote a `raftPort + offset` guess for the new peer. Every other node derived the peer's Raft host plus its **own** HTTP port, so two nodes could dial the same member on two different addresses. A declared address reached only the node it was typed on and the leader.

Ask **any** node when you want to compare what the nodes understand: every node asks the others, so every node can report each peer's `capabilities`. This is what lets you check "is the rolling upgrade finished" through a Kubernetes Service or a load balancer, neither of which routes you to the leader.

Because every node asks, the probe traffic grows with the square of the cluster size. At normal Raft cluster sizes that is a few small HTTP calls every five seconds; it is worth knowing only if you run an unusually large voter set.

8.10.5. Named Peers

By default, each node identifies its slot in `arcadedb.ha.serverList` from the numeric suffix of `arcadedb.server.name` (for example, `ArcadeDB_0` maps to the first entry, `ArcadeDB_1` to the second, and so on). Display names shown in logs and Studio are then synthesized from the local node's prefix. This positional convention works well for Kubernetes StatefulSets but is awkward when nodes have human-readable names such as `frankfurt`, `london`, `nyc`.

You can add an optional `name@` prefix to any entry in `arcadedb.ha.serverList` to give each peer a stable, human-readable name:

```
$ bin/server.sh -Darcadedb.ha.enabled=true \
                -Darcadedb.server.name=frankfurt \
```

```
                -Darcadedb.ha.serverList
=frankfurt@10.0.0.1:2434:2480,london@10.0.0.2:2434:2480,nyc@10.0.0.3:2434:2480
```

When peer names are configured, each node identifies its slot by matching `arcadedb.server.name` against the configured peer names first; if no match is found, it falls back to the legacy `prefix_N` / `prefix-N` suffix resolution. This means:

- Server names no longer require a numeric suffix when peer names are used.
- Display names shown in logs and Studio reflect the configured peer name (e.g. `frankfurt (10.0.0.1:2480)` instead of `ArcadeDB_0 (10.0.0.1:2480)`).
- Mixed clusters work: entries with `name@` get their explicit name; entries without it fall back to the existing positional synthesis.
- Peer names must be unique within the cluster.

The Raft peer ID is still derived from the address (e.g. `10.0.0.1_2434`), so changing or omitting peer names does not affect Raft identity stability.

8.10.6. Architecture

ArcadeDB uses a leader/replica model with Raft consensus. At any time one server holds leadership and accepts writes; the others are replicas that serve reads and stand by for failover.

Cluster of Servers

Each server persists its own Raft log segments under `<rootPath>/raft-storage/`. The log is used for recovery after restart and to replicate state to peers that fall behind.

Size the Raft storage volume for the log, not only for the databases. If the volume fills, Ratis marks the log as failed at the first entry it could not write and the node rejects every following append while still reporting itself `RUNNING`. The node recovers from that state on its own: the health monitor (`arcadedb.ha.healthCheckInterval`) restarts the Raft server in place as soon as the volume has room for two log segments again - which the periodic log compaction (`arcadedb.ha.snapshotInterval`, escalated below `arcadedb.ha.raftStorageMinFreeSpacePerc`) normally provides - and logs a throttled SEVERE while it is waiting. The in-place restarts are bounded by `arcadedb.ha.ratisRestartMaxRetries` per incident, so a failure that is not about space (a failing disk, a permission change) is left to the operator instead of restarting the node on every tick. For the same reason, a follower that reinstalls a database from the leader's snapshot first purges its own Raft log when that volume is under pressure, and refuses the download up front when the volume cannot hold the archive, instead of failing halfway through the extraction.

Any read (query) can execute on any server in the cluster. All writes must go through the leader: a replica that receives a write transparently forwards it to the current leader via HTTP.

Write request executed on a replica (proxied to leader)

Internally, every committed write follows a **replicate-first, commit-after** three-phase commit:

1. Phase 1 (read lock): capture the WAL pages produced by the transaction, validated against the page versions the node holds.
2. Phase 2 (no lock): append the payload to the Raft log and wait for quorum acknowledgment.
3. Phase 3: apply the pages to the local database and return to the client.

If Phase 2 times out or fails, Phase 3 never runs: no local writes, no divergence. Multiple concurrent transactions are batched into fewer Raft round-trips via group commit (`HA_RAFT_GROUP_COMMIT_BATCH_SIZE`).

Phase 3 runs on the Raft apply thread of every node, the leader included, when the entry reaches its position in the log: the leader publishes the pages its transaction prepared, a replica applies the entry's WAL bytes. Every node therefore writes its pages in log order, and the leader's own commit can never overtake or trail the apply of a neighbouring entry.

Two transactions validated on different nodes against the same version of a page (for example two documents that share a page, each written by a different node) are both shipped to the leader with the same next version. The leader validates every entry, page by page, against the versions the log has already assigned, at the point that decides the log order, and refuses the second one **before appending it**: the originator receives a retryable `ConcurrentModificationException`, exactly as it would for a conflict on a single node, re-reads the page and retries. An entry is never merged onto another one region by region, so a node can neither lose an acknowledged write nor drift from its peers. While such an entry is in flight, a transaction on the leader that touches one of its pages fails its own Phase 1 with the same retryable error, without a round trip.

Since v26.12.1 the same rule holds for the work the leader starts on its own and replicates afterwards: TimeSeries compaction, retention, downsampling and `DELETE` of samples, LSM index compaction and vector index persistence write their pages on the leader first, and their entry reaches the log later. When an entry of another node touches a page such a commit writes, one of the two is refused with the same retryable error, and validations of other nodes' entries on that database wait while one of those commits writes its pages. Before, both could be accepted with the same page version, and a compaction could then remove a sample acknowledged by a follower from every node (issue #11921). A TimeSeries compaction or `DELETE` of samples that meets such an entry on the shard's header page (every append writes it) waits up to one second for the entry to be applied before it commits, and fails as before only if the entry is still in flight after that. Before, it retried at once and gave up, so under a steady stream of appends through followers a compaction kept failing and the shard's mutable pages kept growing (issue #12470). An in-place restart of the Raft server waits for such a commit to finish, up to `arcadedb.ha.ratisCloseTimeoutMs`, before it replaces the node's replication state.

8.10.7. Schema Replication

Schema DDL does not travel through the WAL pages described above. A schema change is recorded in a **schema recording session**, and when the outermost session on the thread closes, the leader replicates it as its own Raft entry type (`SCHEMA_ENTRY`), carrying the schema document and any bucket or index files the change created. One session is one entry.

Normally each DDL statement is one session. A SQL script made only of schema definition statements runs as a single session when `arcadedb.schemaBulkDDLScript` is on (the default), and so

does a Java `Schema.bulkChange()` block: the whole batch is one Raft entry (see reference/sql/sql-script.pdf). The trade-off is that the database write lock is held for the whole script.

That matters on the leader. Before appending an ordinary transaction, a commit waits for an active schema session to finish, so the transaction cannot reach the log ahead of the schema entry that creates the files it writes to. The wait lasts at most `arcadedb.ha.quorumTimeout` (10 s by default); after that the commit goes ahead anyway. A DDL script that runs longer than that leaves concurrent writers racing the schema entry. A long index build already opens the same window, so this is not new, but batching makes long sessions more common. Keep DDL scripts to schema definition, run data loads separately, or set `arcadedb.schemaBulkDDLScript=false` to go back to one entry per statement.

On the follower, a DDL entry is applied incrementally: only the components for the files the entry created are instantiated, instead of every component in the database being rebuilt (`arcadedb.ha.schemaIncrementalApply`, on by default). Building a large schema used to cost time quadratic in the number of types.

Since v26.10.2 the schema changes that used to rewrite `schema.json` on the leader alone travel in a `SCHEMA_ENTRY` too, so every follower, and the node that becomes leader after a failover, enforces the same rules:

- property constraints and attributes, set with `ALTER PROPERTY` or the Java `Property` setters: mandatory, not null, read only, min, max, regexp, custom values, type, hidden, compression and external;
- the database time zone (`Schema.setZoneId()`) and the date and date-time formats (`arcadedb.dateFormat` and `arcadedb.dateTimeFormat`, changed with `ALTER DATABASE`), and schema extensions;
- `ALTER TYPE … lightweight` and `ALTER TYPE … unique` on an edge type;
- the downsampling policy of a TimeSeries type (`ALTER TIMESERIES TYPE … DOWNSAMPLING POLICY`);
- the bucket selection strategy of a type (`ALTER TYPE … BucketSelectionStrategy`).

Calling one of these setters through the Java API on a follower's database now fails, as every other DDL does, with a `ServerIsNotTheLeaderException` (`Changes to the schema must be executed on the leader server`) instead of changing that follower alone. Send the statement through SQL instead, which a follower forwards to the leader. `Schema.setZoneId()` now also requires the `updateDatabaseSettings` permission, like the two format setters.

8.10.8. When a follower falls back to a full schema load (from v26.10.2)

An entry the incremental path cannot express makes the follower rebuild every component of the database on its apply thread: an entry that retires a file, adds a dictionary, a compacted index or a bloom filter, writes pages into a compacted index or a bloom filter, or renames component files, and any entry while `arcadedb.ha.schemaIncrementalApply` is `false`. On a large schema that is what shows up as an apply-lag spike after a DDL statement, and it re-creates every index instance cold (see the note on vector indexes below).

It is no longer silent. Each fallback is logged at `INFO` with the database, the entry's Raft index, the reason, what triggered it and how long the full load took:

```
Applied schema entry 42 of database 'mydb' with a FULL schema load instead of the
incremental refresh, which was declined
because the entry renames component files [file_renamed (4 file(s) renamed)]
(filesToAdd=0, filesToRemove=0,
filesToRename=4): every component of the database was re-instantiated on the apply
thread in 1830 ms
```

and published as two metrics, tagged by `reason`, that are present from the first scrape and restart from 0 with the process:

Metric	Meaning
`arcadedb.ha.schema.full_load.fallbacks`	Full loads started, so one that threw is counted too. Alert on `increase()`.
`arcadedb.ha.schema.full_load`	A timer of the full loads that finished: its count and its total time give the average cost of a fallback.

The `reason` tag is one of `incremental_disabled`, `file_renamed`, `not_loaded`, `file_retired`, `non_incremental_new_file`, `non_incremental_touched_file` and `touched_file_missing`.

A full load also re-creates every vector index with nothing resident, so the first search of each used to read all of its pages and load or rebuild the graph on the query thread. Since v26.10.2 the vector indexes that had a graph resident are warmed in the background, one after the other, after `arcadedb.vectorIndex.warmUpAfterSchemaReloadDelayMs` (1 s by default; `-1` disables it). Indexes nobody searched are not warmed.

8.10.9. Renaming a type on a cluster (from v26.10.2)

`ALTER TYPE <name> NAME <new-name>` (and the Java `DocumentType.rename()`) renames the type's files as well as the type: its buckets, the files of its indexes and, for a vertex type, its edge buckets. Since v26.10.2 the rename replicates with the schema change: the `SCHEMA_ENTRY` names every file that was renamed, and each follower renames the same bucket, index and edge files before it loads the new schema. That load is a full one (reason `file_renamed`, see When a follower falls back to a full schema load (from v26.10.2)). Sent to a follower, the statement is forwarded to the leader like any other DDL.

A node running an older build cannot rename the files, and would drop the type's buckets on its next schema reload. So while any peer of the Raft configuration runs an older build, or cannot be reached to say which build it runs, the rename is refused up front with HTTP `409 Conflict` (`Cluster is not ready for this operation`, with the missing peers in `exceptionArgs`), and nothing is renamed on any node. Finish the rolling upgrade, or restore contact with the peer, and run the statement again.

No setting overrides this refusal: `arcadedb.ha.securityEntryCapabilityGate` does not apply to it. To rename a type while a peer is gone for good, remove that peer from the Raft configuration first (`DELETE /api/v1/cluster/peer/{peerId}`, see Cluster Management REST API).

Once a rename has been replicated, the cluster cannot be downgraded below v26.10.2 on its retained Raft log: an older build replaying the entry cannot apply the rename. Rebuild a node you need to roll back from a snapshot, as described for schema deltas below.

8.10.10. Schema Delta Replication

By default each `SCHEMA_ENTRY` carries the whole schema document, so on a schema of several megabytes a one-property `CREATE PROPERTY` costs megabytes on the leader's CPU, on the wire, in every node's Raft log and in each follower's rewrite of `schema.json`. With `arcadedb.ha.schemaDelta` (on by default) the leader ships a delta against the schema the followers already hold. It falls back to the whole document whenever a delta cannot be trusted: the first schema entry after it becomes leader, an entry shipped in instalments, a compaction entry, or a change too large for a delta to be worth it.

Upgrades need no ordering. A node running a version that predates the delta cannot see it: it would apply an empty schema change and its schema would diverge from the leader's without an error. To prevent that, the leader asks every peer in the Raft configuration, through `POST /api/v1/cluster/capabilities`, whether it can decode a delta, and ships whole documents unless every peer has said yes. A peer that is unknown, unreachable, too old to have the endpoint, or whose answer has gone stale counts as no. During a rolling upgrade, deltas start on their own once the last node runs a version that can decode them.

A rolling downgrade to a version that predates the delta is not protected. The negotiation controls what the leader writes from now on, not the delta entries already committed to the Raft log. A node restarted on an older build replays those entries with a decoder that cannot see the delta, applies empty schema changes and diverges silently. To roll a node back that far, rebuild it from a snapshot (reinstall its databases from a current leader and discard its Raft storage) instead of restarting it on its retained log.

While the setting is on, the leader keeps one parsed copy of the schema document per replicated database in heap to diff the next change against: tens of MB per database on a multi-MB schema. Turning the setting off releases it and forces whole documents, which is also useful when diagnosing a schema divergence. Turning it off is safe at any time; turning it back on costs one whole-document entry per database to rebuild the diff base.

Whether deltas are actually being shipped is not exposed through metrics yet. The counters are available only from Java, through `RaftReplicatedDatabase.getSchemaDeltasShipped()` and `getSchemaDocumentsShipped()`, and with `arcadedb.ha.logVerbose=2` a follower logs `Applying a <n>-char schema delta to database '<name>' …` for each one.

8.10.11. Database settings (from v26.10.2)

`ALTER DATABASE` and the `set database setting` server command change settings that the database keeps in its `configuration.json`.

Since v26.10.2 a changed setting is replicated as its own Raft entry, and every node, the one that ran

the command included, applies exactly the keys the entry names. A setting the command did not change keeps each node's own value. The command returns once its own node has applied the change.

- **Rolling upgrades.** A node running an older build cannot decode the entry. While any peer has not confirmed it can, because it runs an older build or cannot be reached, the change is refused before anything is submitted, with HTTP `409 Conflict` or gRPC `FAILED_PRECONDITION` naming the peers, and the setting is put back on the node that ran the command. No setting overrides this refusal: `arcadedb.ha.securityEntryCapabilityGate=false` only applies to the two security entries (groups and API tokens), and the refusal of a database configuration change says there is no override. *(Since v26.11.1: before, turning the gate off also let a database configuration change through while a peer could not decode it.)*
- **Path-valued settings.** `arcadedb.externalPropertyBucketPath`, `arcadedb.pageSnapshotSpillPath` and `arcadedb.opencypher.loadCsv.importDirectory` set this way now apply on every node, so the directory must exist, and be writable where the setting needs it, on all of them.
- **Unknown outcome.** When the command fails with a replication timeout and it is not known whether the entry committed, the node that ran it settles the setting on what its own `configuration.json` holds: the new value if the entry was already applied there, otherwise the previous one, which the entry replaces on every node if it commits later. Run the command again to be sure which value is in force.
- **Refusal.** When the change is refused outright, the node that ran it puts the previous value back, but only while the setting still holds the value the command stored. If a change to the same setting made on another node was applied in the meantime, that value stands.
- **Overlapping commands.** The entry carries the value its own command set, not the value the node holds when the entry is submitted. When two commands change the same setting at the same time, every node applies them in commit order and ends on the value of the one that committed last.

8.10.12. Fixed-width float arrays (from v26.11.1)

Since v26.11.1 an `ARRAY_OF_FLOATS` or `ARRAY_OF_DOUBLES` value is stored fixed-width (see Data Types), a form older builds cannot read. It is not a Raft entry but part of the record bytes in the pages a transaction replicates, so a node on an older build would apply them and then fail every read of the record. No node writes the new form until every peer of the Raft configuration advertises the `fixed-width-float-arrays` capability; until then the old form is written, which every build reads. During a rolling upgrade the new form starts on its own once the last node runs v26.11.1. Both forms are always read, so a cluster holding a mix of them needs no action. Since v26.12.1 an upgraded node labels the database dictionary with format version 2 (see Upgrade), so a node still on an older build that has to install a full snapshot from an upgraded node refuses the dictionary until it is upgraded too. The cluster checksums compare the dictionary under its version 1 name, so nodes holding the same content under the two labels do not report a mismatch.

8.10.13. Write Quorum

A **quorum** is the number of peers that must acknowledge a write before it commits. Configure it via `arcadedb.ha.quorum`:

- `majority` (default) — a majority of peers must acknowledge (standard Raft).
- `all` — every configured peer must acknowledge.

If the configured quorum is not met within `arcadedb.ha.quorumTimeout` milliseconds, the transaction is rolled back and an error is returned to the client.

The legacy quorum values `none`, `one`, `two`, `three` are **not supported** and will cause a startup error. If you were using any of those values, update your configuration to `majority` or `all` before upgrading.

8.10.14. Read Consistency

When a read runs on a replica, ArcadeDB offers three consistency levels, configurable per-server via `arcadedb.ha.readConsistency`:

Level	Behavior
`eventual`	Read locally without waiting. Fastest, but may return data that was committed on the leader after the replica's last apply.
`read_your_writes` (default)	The replica waits until the Raft log index corresponding to the client's most recent write has been applied locally before serving the read.
`linearizable`	The replica issues a Raft `ReadIndex` request to the leader and waits until its local applied index reaches the leader's committed index for that request. Strongest guarantee, highest latency.

A node that is no longer a member of the Raft configuration (it left with `leaveCluster` or was removed with `DELETE /api/v1/cluster/peer/{id}`) receives no more entries. On such a node, `linearizable` reads and `read_your_writes` reads whose bookmark it has not applied are refused with a retryable error naming the reason, so the client can retry on a member. This includes reads that were already waiting when the node was removed: they are refused within a second instead of after the quorum timeout. `eventual` reads, and `read_your_writes` reads whose bookmark the node already applied, are still served. *(Since v26.10.2: such reads waited the whole quorum timeout, and a* `read_your_writes` *read then served data missing the write its bookmark named.)*

The same node refuses at once, with a retryable `NeedRetryException`, everything that would submit an entry to the cluster: a commit that carries writes (a read-only transaction still commits), a database drop, a database create or install, an `ALTER DATABASE` configuration change, and the replication of a security user, group or API token. The message names the reason and the way out, for a database operation `Database '<name>' cannot <operation> on this server: it is not a member of the Raft cluster's configuration any more (it left or was removed), so it receives no entries and could not apply this one. Send the request to a server of the cluster, or add this server back to it` and for a security document `Cannot replicate the security <document> from this server: it is not a member of the Raft cluster's configuration any more (it left or was removed), so it receives no entries and the change would apply on every other server but not here. Send the request to a server of the cluster, or add this server back to it`. *(Since v26.10.2: the leader still accepted the entry and committed it on every member, while the removed node waited* `arcadedb.ha.quorumTimeout` *for an apply it never received and then released the transaction unapplied; a drop dropped the database on every member and then failed on the removed*

node, and a security change was in force everywhere except on the node that answered the request.)

Clients communicate the consistency contract through three HTTP headers:

Header	Direction	Purpose
`X-ArcadeDB-Read-Consistency`	request	Overrides the server default for this request. Accepts `eventual`, `read_your_writes`, or `linearizable`.
`X-ArcadeDB-Read-After`	request	Client-supplied bookmark: the Raft commit index the replica must have applied before serving the read. Used to implement read-your-writes across connections.
`X-ArcadeDB-Commit-Index`	response	Current last-applied commit index, echoed on every response, including error responses and streamed (`Accept: application/x-ndjson`) query responses. Clients capture this value and send it back as `X-ArcadeDB-Read-After` on subsequent reads.

Older clients that sent `X-ArcadeDB-Commit-Index` on requests are still accepted as a backward-compatible fallback.

The bookmark is a lower bound: it says the server had applied **at least** that index when it answered. That holds on a refused request too, so a client can keep its read-your-writes barrier after an error instead of falling back to a stale one.

The one endpoint where it is not a header is the streaming bulk load, `POST /api/v1/batch/{database}` with `Accept: application/x-ndjson`. There the response has already started by the time the commit index is known, and a header set at that point would be dropped silently, so the value travels in the body instead, as `commitIndex` on the terminal `summary` or `error` line. The Java driver handles this transparently: `RemoteDatabase.getLastCommitIndex()` advances the same way whichever encoding was used.

Since v26.10.2 a replica relays such a streamed load to the leader in both directions at once, so a client that sends it to a replica receives the leader's `progress` lines while it is still uploading, as it does on the leader. A body that grows past `arcadedb.server.httpBodyContentMaxSize` after the leader started answering ends the stream with an `error` line carrying status `413` during the upload too.

8.10.15. Load Shedding and Backpressure

The Raft group-commit queue is bounded (`arcadedb.ha.groupCommitQueueSize`, default 10000 pending transactions). When a write arrives but the queue is full, the server waits up to `arcadedb.ha.groupCommitOfferTimeout` ms (default 100) for a slot, and if still full throws `ReplicationQueueFullException` — a `NeedRetryException` that clients automatically retry with backoff. This protects the server from OOM under sustained write overload and lets clients back off gracefully.

8.10.16. Witness / Read-Scale Nodes

Set `arcadedb.ha.serverRole=replica` to pin a node as a permanent replica. The node's Raft priority is

set to 0 at startup so it is never elected leader, while still receiving all writes and serving reads. Useful for read-scale deployments or for cross-DC witnesses that should not take over as leader.

8.10.17. Automatic Failover

If the leader becomes unreachable, replicas start a new Raft election. A replica with an up-to-date log is elected as the new leader and the cluster resumes serving writes. Pre-vote prevents partitioned nodes from triggering disruptive elections.

Common causes of leader unavailability include:

- The ArcadeDB server process has been terminated.
- The physical or virtual host has been shut down or rebooted.
- Network issues prevent the leader from reaching a majority of peers.

When a replica rejoins after being offline, it catches up via the Raft log automatically; if it has fallen behind past the log purge boundary, the leader streams a full database snapshot over HTTP and the replica installs it atomically. Since v26.10.2 the snapshot is compressed at DEFLATE level 1 by default (`arcadedb.ha.snapshotCompressionLevel`), and two followers are served at the same time; see Performance Tuning for when to change either.

A vector index's proximity graph is built by every node on its own, so it is not part of the data the nodes compare or share. Since v26.10.2 a snapshot ships the graph file (`.vecgraph`) empty, without its side files, and the follower builds and verifies a graph of its own on the first search of each vector index after the install, the way every node does after `CREATE INDEX`. That first search pays for one graph build. A follower that installed a snapshot on an earlier version holds the leader's graph pages without the manifest that vouches for them and reuses them unverified: each such reuse is counted in `unverifiedGraphReuses` in the index's `getStats()` and logged at `WARNING` (`Vector index <name> is reusing a persisted graph that carries no manifest …`). Such a follower keeps that graph until its next graph persist, a `REBUILD INDEX`, or its next snapshot install.

The leader also **auto-recovers a follower's replication channel** that is wedged on a stale peer address — for example after a Kubernetes pod is rescheduled onto a new IP: it re-resolves the peer and re-establishes the channel on its own, instead of leaving the follower silently disconnected and the cluster at bare quorum until an operator forces a leadership transfer. See Replication-Channel Self-Healing.

8.10.18. Replication-Channel Self-Healing

A follower can be up, healthy, and reachable by every other node, and still receive nothing from the leader. The leader keeps one long-lived gRPC channel per follower. If that channel is wedged on a peer address that is no longer valid (the classic case is a Kubernetes pod restarting onto a new IP inside the JVM's positive DNS cache window), the leader's appender for that follower keeps writing into a socket that goes nowhere. The follower's `matchIndex` stops advancing, or never advances at all (`matchIndex=-1`), while the rest of the cluster commits normally. Quorum is unaffected as long as a majority is still healthy, so the cluster keeps serving writes at reduced redundancy and nothing fails loudly.

ArcadeDB detects and repairs this on its own, in three escalating steps.

1. Detect: the follower is reported unreachable. The leader's cluster monitor tracks the time since the last successful RPC to each follower. Once that exceeds `arcadedb.ha.peerUnreachableThreshold` (default 10s), the follower is reported **unreachable**. This is a diagnostic signal only: it does not change Raft membership, quorum, or the write path. It is, however, the input every step below depends on, so setting it to `0` disables the whole self-healing sequence.

2. Reset: rebuild that one follower's channel. Once a follower has stayed continuously unreachable for `arcadedb.ha.peerChannelResetDuration` (default 60s), the leader closes that follower's replication channel so the next send re-resolves DNS and reconnects. Only the unreachable peer's channel is touched; the other followers, and leadership itself, are left alone, so there is no flapping risk. A follower that reconnects inside the interval clears the streak and is never reset, so an in-progress reconnection is not disrupted.

A single reset does not always stick, because the rebuilt channel may still resolve the stale address if the JVM DNS cache has not expired. The reset is therefore retried once per interval, **up to 5 attempts**. The counter re-arms as soon as the follower becomes reachable again. Set the duration to `0` to disable the automatic reset and keep only the manual leadership transfer.

3. Escalate: transfer leadership to rebuild the appender. The retry budget alone leaves a gap: the streak only re-arms when the follower becomes reachable, so a permanently wedged channel would sit in the given-up state until an operator restarted the leader process. `arcadedb.ha.peerChannelResetEscalation` (default `true`) closes it. When the budget is exhausted and the channel is still dead, the leader **transfers leadership to a healthy peer**, so the new leader builds a brand-new appender to the stuck follower.

The target is selected with the same rules as a manual step-down and is never the wedged follower itself. If no healthy peer is eligible, the leader logs at SEVERE and leaves the follower for the operator.

The escalation is bounded, not perpetual. Leadership transfer is exactly what makes a **new** leader inherit the problem, so each healthy peer escalates a given follower **at most once per 30-minute cooldown**. If the follower is unreachable for a reason a fresh appender cannot fix (it is genuinely down, or partitioned away), the cluster passes leadership around at most once per peer and then settles on the operator-intervention path instead of churning. Set the flag to `false` to keep the detection and logging but never move leadership automatically.

Log lines to look for

The whole sequence is visible in the leader's log. Searching for these strings tells you which step the cluster reached:

Log line	Meaning
`Replica '<peer>' has NEVER received a single append (matchIndex=-1) …`	SEVERE. The follower's replication path is dead: it has not received one entry since the leader took office. This is the symptom, logged before any repair is attempted.

Log line	Meaning
`Follower '<peer>' unreachable for <n>ms; triggering a replication-channel reset (attempt N/5) …`	WARNING. Step 2. `N` tells you how much of the retry budget is left. Repeated lines climbing to `5/5` mean the resets are not sticking.
`Follower '<peer>' still unreachable after 5 replication-channel resets over <n>ms; escalating to rebuild the appender for it.`	SEVERE. The budget is exhausted and escalation is enabled. The leadership transfer follows.
`Transferring leadership to '<target>' to rebuild the replication appender for wedged follower '<peer>'.`	WARNING. Step 3 firing. A leader change in the logs immediately after this one is intentional, not a failure.
`… giving up automatic channel recovery - operator intervention (e.g. a leadership transfer) is required.`	SEVERE. Escalation is disabled (`peerChannelResetEscalation=false`), no healthy peer was eligible, or the 30-minute cooldown already covered this follower. Nothing further is automatic: investigate the follower.

If you reach the last line, the follower is not suffering from a stale address, and rebuilding the channel will not help. Check that the follower process is alive, that its Raft port is reachable from the leader, and that its address resolves to an entry in `arcadedb.ha.serverList`. A peer that moved to an address outside the list is rejected by the peer allowlist described under Security.

8.10.19. Durable Raft Storage

`arcadedb.ha.raftPersistStorage` defaults to `true`. The per-node Raft log under `raft-storage-<nodeName>` is **persisted across restarts**, and `arcadedb.ha.raftStorageDirectory` must live on **durable** storage (a persistent volume, not an ephemeral or `tmpfs` mount).

With ephemeral Raft storage, wiping the log on restart could leave the cluster in a broken state:

- a lagging follower that came back after a **full-cluster cold restart** could permanently diverge and fail to rejoin, surfacing as a `WALVersionGapException`; or
- the cluster could silently re-form as a fresh single-node cluster, dropping committed history.

Durable storage lets a restarted node rejoin by **replaying its own persisted log** rather than always requiring a full snapshot resync from the leader.

Migration: the per-node directory is named `raft-storage-<nodeName>` dynamically, which container runtimes and Kubernetes cannot mount as a volume (a wildcard like `raft-storage-*` is created as a literal directory, not expanded). Set `arcadedb.ha.raftStorageDirectory` to a **static** path (e.g. `/home/arcadedb/raft-storage`) and mount **that** path on durable storage before upgrading. On Kubernetes, back it with a `PersistentVolumeClaim` — see Kubernetes. On a plain filesystem you can instead copy the existing `raft-storage-<nodeName>` directory across the upgrade. A throwaway or test cluster that intentionally wants ephemeral behavior can opt out with `arcadedb.ha.raftPersistStorage=false`.

8.10.20. Committed-remotely response contract

When a write is forwarded from a replica to the leader (or issued directly to the leader), it is possible for the transaction to be **committed cluster-wide by quorum** and then fail to apply on the node that is answering the client. This is a distinct outcome from an ordinary failure: the data is durably committed, so retrying would create a duplicate.

This case has its own contract. The server raises a `TransactionCommittedRemotelyException`, mapped over HTTP to **status `409 Conflict`**. A client that receives it must **not** retry the transaction — the write already succeeded cluster-wide; it should instead re-read to observe the committed state. This is different from a transient conflict (a genuine write-write conflict or `DeadlockException`), which remains safe to retry.

8.10.21. Outcome-unknown response contract

(Since v26.12.1.) A write can also end in an outcome nobody knows: the node answering the client handed it to the replication and quorum was not confirmed (`ReplicationDispatchedTimeoutException`), or a replica forwarded it to the leader and the connection to the leader was lost, or the leader did not answer in time, after the request had been sent. The write may have been committed or not.

The server raises a `TransactionOutcomeUnknownException`, mapped over HTTP to **status `504 Gateway Timeout`** with the label `Transaction outcome unknown: the write may have been applied, check before retrying it`, on the leader and on the replica alike. Before v26.12.1 these answered `500`, and a lost leader connection reported a raw `java.io.EOFException`. The answer carries no `Retry-After`, and the Java remote client does not retry it.

A client that receives it must neither give the write up nor resend it blindly: read the database to see whether the write landed, then resend it only if it did not. Sending the same `X-Request-Id` again does not settle it, because the leader keeps its answers in memory and a leader that died no longer has them. A failure to connect to the leader, with no request sent, is different: it stays a retryable `503` (`NeedRetryException`), since the write did not run.

8.10.22. Writes during a leadership transfer

(Since v26.12.1.) While a leadership transfer is pending (`POST /api/v1/cluster/leader`, a rolling restart, the cluster's own hand-off), the leader steps down and refuses every write it receives. The refusal happens before the write reaches the Raft log, so nothing was written. It is a retryable refusal, like a conflict on a record page:

- over HTTP it answers `503` with a `NeedRetryException` whose message says the leader is stepping down, and it is the same through a replica that forwarded the write. The server's own transaction retry (`arcadedb.txRetries`) retries it, so many of these writes succeed without the client noticing.
- over gRPC it answers `ABORTED`.

A write forwarded by a replica at the moment leadership moves off that replica answers `503` too, with no leader named. Before v26.12.1 these answers were `500` (`INTERNAL` over gRPC) with "outcome unknown", and the forwarded one could be a `400` that sent the operator to the server list. The `400`

stays for a really wrong server list: another peer's HTTP address that resolves to this node (see `arcadedb.ha.serverList`).

There is one exception. If the Raft client had already sent the same write again after an earlier attempt that may have reached a log (a timeout, a lost reply, a closed stream), the first copy may still commit, so the stepping-down refusal of the second copy is not proof that nothing was written. It stays an outcome-unknown answer, see Outcome-unknown response contract: check before retrying.

8.10.23. Writes through a replica during its resync

(Since v26.12.1.) While a replica's copy of a database is replaced from the leader (`POST /api/v1/cluster/resync/{database}`, a resync the leader starts, a bootstrap reinstall), a write sent to that replica over HTTP `POST /command` and forwarded to the leader is answered with the leader's `200` answer once the leader committed it, even when something on the replica fails afterwards because its copy is being replaced. Before v26.12.1 such a write was answered `503 Cannot execute command`, although the leader had committed it, so a client that retried it applied it twice.

The leader's answer is used only for a request outside a session that forwarded exactly one write. Its body is the leader's rendering of the statement: in the client's own `serializer`, `limit` and `typeHints` when the forward carried the whole request, otherwise in the default rendering (`record` serializer, default row limit, no type hints). A request that wrote or ran a command on the replica, or forwarded several writes, keeps the retryable `503`, and so does a write refused with `Server is installing a snapshot, please retry`, which is refused before anything is forwarded.

8.10.24. Creating a database in a cluster

(Since v26.12.1.) A `create database` server command creates the database only once a quorum has committed it: every server, the one that ran the command included, creates the database when it applies that change, and the command answers once the server that ran it has the database. If the quorum is not confirmed in time, the command fails with an unknown outcome (see Outcome-unknown response contract) and no server has the database yet: it appears on every server if the change commits later (this can take a few seconds more, when the leader steps down and a new one is elected), and on none if it never commits. Check with `GET /api/v1/exists/<dbname>` before sending the command again. Before v26.12.1 the server that ran the command created the database first, so a creation that never committed, for example because the leader was lost right after it, left the database on that server only: it listed the database and answered reads of it, while the other servers answered `Database not found` and the cluster still reported itself in sync (issue #11848).

8.10.25. Creating or dropping a database through a replica

(Since v26.12.1.) A `create database` or `drop database` server command sent to a replica is forwarded to the leader, and the replica answers the leader's `200` only once it has applied the change itself: after a creation the next request to that replica finds the database, after a drop `GET /api/v1/exists/<dbname>` on that replica answers `false`. The wait lasts at most `arcadedb.ha.quorumTimeout` (10 s by default), and usually a few milliseconds. If the replica cannot confirm in time it logs a warning and still answers `200`, since the leader made the change. Before v26.12.1 the replica answered at once, so a client that created a database and then its schema

through a load balancer could be answered `404 Database not found` when both requests reached the same replica, and a client that checked a drop could still see the database (issue #11739).

A drop applied on a replica where the database had been closed with `close database` now also ends that close, so the replica no longer reports the dropped database as existing.

8.10.26. Dropping and creating a database again under the same name

(Since v26.12.1.) A server that restarts replays its Raft log from its last checkpoint. When a database was dropped and created again under the same name after that checkpoint, the replay now skips every change that belongs to the dropped database, its drop included, and leaves the database created later in place. Before v26.12.1 the replay applied the old schema changes, transactions and drop to the new database, so until the replay reached the end of the log the restarted server could answer `404 Database not found`, `400 unknown timeseries type`, or `500`, and quarantined a database with a TimeSeries type (issue #11919). A database dropped for good is still dropped by the replay.

Each server records where its copy of each database was created in `.raft/created-index`, next to `.raft/applied-index`. The protection therefore applies to databases created, or received from the leader, by a server running v26.12.1 or later: a database created before the upgrade is protected from its next drop and create on. A database restored from a backup records no such position.

8.10.27. Long statements forwarded to the leader

(Since v26.12.1.) A replica forwards a statement to the leader over HTTP/1.1, so the replica's client gets the answer however long the statement runs on the leader, as when the statement is sent to the leader directly. Before v26.12.1 the forward used HTTP/2, and the leader closed the connection `arcadedb.network.socketTimeout` (30 s by default) into any longer statement, such as a `CHECK DATABASE` on a large database: the client got `500` with `java.io.EOFException: EOF reached while reading` while the leader kept running the statement, and a retry ran it again (issue #12442).

8.10.28. When a replica refuses a write and names the leader

A replica forwards the writes it receives to the leader (see High Availability Concepts for the writes it runs itself, inside a transaction the client opened). Some writes cannot be forwarded and are refused instead: a schema change sent over a protocol with no forward, or any write when the forward itself cannot be made. Such a refusal names the leader, and since v26.10.2 every wire protocol answers it as a redirect, in that protocol's own terms, with the leader's address in the text or in a field of its own:

Protocol	Answer
HTTP	`400` with the leader's address in `exceptionArgs`.
gRPC	`FAILED_PRECONDITION` with the leader in the `arcadedb-leader-*` trailers; on a streamed insert, the `NOT_LEADER` code with `leader_address`. See reference/grpc-api.pdf.
`/ws` insert session	The `NOT_LEADER` code with `leaderAddress`. See reference/http-api/http.pdf.

Protocol	Answer
Bolt	`Neo.ClientError.Cluster.NotALeader`. The text names the leader's Bolt address when the routing table knows one, otherwise its HTTP address. A routing (`neo4j://`) driver drops the node as writer and retries against the new one.
MongoDB	`10107 NotWritablePrimary`, with `not primary. The leader is server '<address>'` in `errmsg`.
Redis	`READONLY`, the answer Redis gives to a write sent to a replica, naming the leader.
PostgreSQL	`40001`: the protocol has no redirect code, so the state stays the retryable one and the message ends with `The leader is server '<address>'`.
Gremlin Server	The error's status message ends with `. The leader is server '<address>' (HTTP address; use its Gremlin Server port). Retry there`, following the production placeholder in production mode. The cluster knows its peers by their HTTP addresses, so connect to the Gremlin Server port of that host. See reference/gremlin/gremlin.pdf.
MCP	The tool error (`isError: true`, over HTTP `200`) ends with `. The leader is server '<address>'`, appended to the message, or to the production placeholder. See reference/mcp/mcp.pdf.

The address is added after the message is concealed in production mode, so it reaches the client in every mode. A refusal raised during an election, when no leader is known yet, names nobody and keeps its retryable answer (`503` over HTTP): retry shortly.

(Since v26.10.2.) The PostgreSQL extended query protocol, which most drivers use for prepared statements, forwards a replica's DDL, and its writes outside an explicit transaction block, to the leader like the simple query protocol does.

8.10.29. Running statements on a cluster

Each server keeps its own list of running statements: the HTTP `list queries` and `terminate query` server commands and Cypher `SHOW TRANSACTIONS` and `TERMINATE TRANSACTIONS` run on the server that receives them, a replica included, and are never forwarded to the leader. A statement id names a statement on the server that issued it.

Since v26.10.2 a write that a replica forwards to the leader can be followed and stopped from either side:

- On the replica it is listed with `forwardedTo`, the name of the leader running it.
- On the leader it is listed with `forwardedFrom`, the id it has on the replica, and with the tag the client sent (`X-ArcadeDB-Query-Tag`), so `list queries tag <label>` finds it on the leader too.
- `terminate query <replica id>` sent to the replica stops the work on the leader as well, and the client receives the termination error of its protocol.
- `terminate query <replica id>` sent to the leader finds the statement by the replica's id too.

The forward carries the statement's id under the cluster token, which an HA cluster always has configured (see Security). The usual visibility rule applies on both servers: **root** sees and stops every statement, any other user only their own.

8.10.30. Offline Cluster Bootstrap

By default, when an HA cluster forms for the **first time** and one or more peers already hold a non-empty database on local disk, the peers do **not** download the database over HTTP from a single source. Instead, every peer reports a `(fingerprint, lastTxId)` tuple per database, the cluster elects the peer with the highest `lastTxId` as the source via leadership transfer, and any peer whose fingerprint matches the source bootstraps locally in seconds with zero bytes transferred.

This is the recommended path for scaling a single-instance ArcadeDB deployment up to a multi-node HA cluster after a large dataset has already been imported. Operator workflow:

1. Import the dataset into a single ArcadeDB instance with HA disabled.
2. Tar the database directory, or take a regular full backup.
3. Distribute the archive out-of-band to every peer's filesystem (init container from S3, NFS read-only mount, baked image layer, `kubectl cp`).
4. Start every peer with HA enabled (`arcadedb.ha.enabled=true`) and the usual `arcadedb.ha.serverList`. No special flag is required; the bootstrap path is on by default via `arcadedb.ha.bootstrapFromLocalDatabase=true`.
5. The peers form the Raft group with everyone already at the same state.

Time-to-cluster goes from "minutes-to-hours of HTTP snapshot transfer per replica" to "seconds, because everyone already has the bytes". For the Kubernetes recipe, see Pre-staging the database on every pod.

The bootstrap protocol handles the following cases automatically:

- **Identical pre-staged databases**: every peer's fingerprint matches the elected source, all peers bootstrap locally with zero bytes transferred.
- **Different-age peers**: the peer with the highest `lastTxId` is elected as the source. Older peers reinstall the leader-shipped full snapshot; subsequent transactions are picked up by native Raft replication.
- **Late newer joiner**: a peer with a strictly newer `lastTxId` than the cluster's chosen baseline refuses to start and logs a SEVERE with the recovery procedure. This prevents a misconfigured rolling deploy from silently overwriting newer data with older data.
- **Cluster restart**: the bootstrap path is gated on every peer's Raft log being empty. After the cluster has committed at least one Raft entry, restarts use the regular Raft log replay or leader-shipped snapshot path; the bootstrap path does not engage, and stale local data on a single pod cannot cause divergence.
- **Databases created while the cluster forms**: an application that creates and seeds its databases right after the first leader election is not baselined against them. The peer that sourced the baseline is never refused for its own copy advancing past the sampled `lastTxId`, and a baseline committed for a database that already has Raft history on a node is ignored on every node alike,

because replication - not bootstrap - keeps those copies in step.

While the cluster is deciding which copy to keep, **every** node - the leader included - answers `503` on `/api/v1/ready` and refuses client requests on the databases being decided, then serves normally as soon as the decision is applied. This usually takes seconds; if a peer is unreachable it can last up to `arcadedb.ha.bootstrapTimeoutMs`.

The committed bootstrap baseline (`fingerprint` and `lastTxId` per database) is visible in the response to `GET /api/v1/cluster` and in the Studio cluster dashboard.

8.10.31. When a node has stopped for good

Most problems a node reports are temporary: it is catching up, or downloading a snapshot, and it rejoins on its own. Two are not, and `GET /api/v1/cluster` says so directly:

- `criticalHalt` - a replicated change could not be applied, so this node has stopped applying changes altogether. Its databases are frozen at that point and stay there until the node is restarted. If the reason mentions an unknown entry type, a newer node is writing a format this one cannot read: upgrade this node first, or it stops again at the same place.
- `raftLogFailure` - the node cannot write to its replication log, almost always a full or read-only disk. It recovers by itself once there is room again.

Both are `null` on a healthy node, and both raise a matching entry in `alerts`. A third field, `crashLoopEscalated`, is `true` once the node has given up restarting itself and needs to be restarted by hand.

`alerts` also distinguishes two situations that used to look the same. `bootstrap-diverged-databases` means this node **kept** a copy that the cluster never adopted - the data is here and intact. `bootstrap-database-missing` means the opposite: this node is **missing** a database the cluster has. Do not copy a node's database directory to the other nodes when it is the one missing the database; run `POST /api/v1/cluster/resync/{database}` on it instead, transferring leadership first if that node is the leader.

The full text of the error behind `criticalHalt` and `raftLogFailure` is shown to the `root` user only. Other users see the condition, the position and the time, which is what a monitoring rule needs.

A node in either state answers `503` on `/api/v1/ready`, so Kubernetes takes it out of the Service. This does not depend on `arcadedb.server.readinessRequiresHA`: a node whose replication has stopped is never ready, whatever that setting says.

Quarantined databases

A node **quarantines** a database when it cannot apply a committed Raft entry for it: an intermediate transaction never reached the node (a WAL version gap), the entry cannot be decoded from the node's own log, applying it failed, a snapshot install did not bring the database up to the snapshot's index, or a schema change ran on this node but was never published to the others. The entry is skipped, so the node's copy is known to be missing a committed change. While the quarantine stands the node:

- answers `503` on `/api/v1/ready`, and `GET /api/v1/cluster` raises the critical `local-resync-in-progress` alert, whose `details.divergedDatabases` and `details.divergenceCauses` name the database and the cause;
- refuses reads that need linearizability on that database instead of serving them stale, and does not checkpoint its Raft log;
- keeps the quarantine across restarts: it is recorded in `.raft/applied-index`.

Since v26.11.1 an index compaction on the leader is quarantined too when the cluster cannot say whether it accepted the compaction: the publication timed out and, after waiting up to the quorum timeout, the outcome is still unknown, or the entry was lost to a leadership change. The leader keeps its old index files and schema untouched until the compaction is accepted, so a leader that crashes in between restarts on the same files as every other server. Before, an unknown outcome was left alone and the old leader's index could silently miss entries written by the next leader.

After a power loss. *(Since v26.11.1)* In production mode, the crash recovery of a database can abort on a gap in its write-ahead log. Before, the database opened anyway and the node served wrong property values and missing index data. Now a database whose crash recovery did not complete is quarantined under the cause `LOCAL_RECOVERY_INCOMPLETE`, and a database that opens with an unavailable unique index under `UNAVAILABLE_UNIQUE_INDEX`. *(Since v26.12.1)* The same cause covers a unique index whose compacted part could not be loaded (the log says `Invalid sub-index … File with id N was not found`), which only enforced its constraint on part of the keys. The node refuses readiness, hands leadership off and resyncs the database from the leader, as for any other quarantine. A node that is the only voter has no leader to resync from and only logs a SEVERE message. Known limits, tracked upstream:

- a recovery that throws on a torn page (a `SerializationException` while replaying the WAL) stops the node from starting on every restart instead of quarantining the database (#11280);
- a quarantined database may still answer reads until its resync completes (#11281, not yet confirmed);
- a first or header page lost after fsync, and a torn page with no WAL history, are not covered yet (#11258); the second one needs a per-page checksum (#11157);
- an index that becomes unavailable after a later schema reload is not rechecked.

The way out is a resync, which reinstalls the database from a peer and lifts the quarantine. The node starts one itself; to force it, run `POST /api/v1/cluster/resync/{database}` on that node. A leader cannot resync from itself: transfer leadership first (`POST /api/v1/cluster/leader`).

A node that is the only voter of its cluster has no peer to resync from, so a quarantine restored from disk, or raised while the cluster still had peers, never lifts there. Neither does one that every voter holds on the same database (the critical `no-healthy-copy-on-any-voter` alert), since no node then has a copy to serve. For those cases only, v26.10.2 adds an operator override, `POST /api/v1/cluster/accept-diverged/{database}`:

- It requires the `root` user. It accepts the node's local copy as the truth, and lifts the quarantine and the read floor that goes with it, in one persisted write. The response carries `database`, `localServer`, `appliedIndex` and, when they stood, `divergenceCause` and `readFloor`.
- The entry the quarantine skipped is **not** replayed: if the copy is missing it, it stays missing. Run

`CHECK DATABASE` afterwards, and restore from a backup if the database is damaged.

- Every use is logged at `WARNING`, naming the user, the source address, the applied index and the cause: `Database '<name>': user 'root' (from <address>) lifted its quarantine WITHOUT a resync, accepting this node's copy as it is …`
- On a node with peers that can serve a resync it is refused with `409 Conflict` (gRPC `FAILED_PRECONDITION`), because the resync is the way out there. A database with no quarantine and no read floor answers `404`.

The same override is available over gRPC as `AcceptDivergedDatabase` (see reference/grpc-api.pdf) and from Java through `RemoteServer.acceptDivergedDatabase()`.

Entries held for the next restart (from v26.11.1)

Some committed entries belong to no single database: security users, groups and API tokens, and a database's `configuration.json`. When a node cannot write one of them (for example the volume is full or read-only), it cannot quarantine a database for it. Versions before v26.11.1 skipped the entry and went on checkpointing the Raft log, so a restart did not replay it and the node silently kept the old users or settings.

Since v26.11.1 the node **holds** the entry instead. It does not checkpoint the Raft log past it, so the log keeps growing, and it logs at `WARNING`, at most every five minutes:

`The Raft log is not checkpointed past index <n> because entries are held for the replay (…)`

A hold is lifted when:

- a later entry for the same user, group or token is written successfully, or a later configuration entry rewrites every setting the held one named;
- the database is dropped or resynced;
- the node installs a snapshot from the leader that covers the entry;
- the node restarts and applies the entry again.

If the warning keeps appearing, fix the cause (free the disk, restore write access) and restart the node.

A committed entry for a single database that fails to apply, whatever the cause, now quarantines that database and resyncs it (see Quarantined databases), instead of being skipped.

Lifting the stale-snapshot read floor. A node that restarts with a Raft snapshot marker ahead of the entries its databases applied (by more than `arcadedb.ha.snapshotGapTolerance`) is missing the entries in between, and the Raft log no longer holds them. It raises a node-wide **read floor**: `/api/v1/ready` answers `503` and every `linearizable` read is clamped until a full resync from a peer fills the gap. A leader cannot resync from itself, and the only voter of a cluster is always its leader, so there the floor never lifts. For that case only, v26.10.2 adds a second override, `POST /api/v1/cluster/accept-stale-snapshot`, which names no database because the floor is node-wide:

- It requires the `root` user, and the body is ignored. It accepts the node's databases as they are and records `max(marker index, applied index)` as the applied position, in one persisted write, so a

restart does not raise the floor again. The response carries `result`, `localServer`, `readFloor`, `snapshotIndex` and `appliedIndex`.

- The entries between the floor and the marker are **not** replayed: a database missing them stays missing them. Run `CHECK DATABASE` afterwards, and restore any damaged database from a backup.
- A database quarantined on its own keeps its quarantine and its read floor: lift those with `POST /api/v1/cluster/accept-diverged/{database}`, one database at a time.
- Every use is logged at `WARNING`, naming the user, the source address, the floor and the marker index: `<user> lifted the node-wide stale-snapshot read floor WITHOUT a resync (read floor <n>, snapshot marker <n>, applied position now <n>) …`
- It answers `404` when no floor stands, and `409 Conflict` on a node that is not the only voter (a follower resyncs on its own, and a leader hands the leadership to a peer first) or while a snapshot download is running on the node. A change that cannot be written to disk answers `500` (`Could not lift the stale-snapshot read floor: <reason>. The floor stays`) and lifts nothing.

The same override is available over gRPC as `AcceptStaleSnapshot` (see reference/grpc-api.pdf) and from Java through `RemoteServer.acceptStaleSnapshot()`.

Accepting the leader's unverified copy. A database closed on a node while it was a follower, which the last resync could not verify, carries a marker file (`.ha-unverified-closed-copy`). Before such a node, once leader, reopens the copy and makes it the cluster's copy, it asks every peer of the configuration about its own, and refuses while a peer does not answer or holds a copy that cannot be ordered against it (the critical `unverified-closed-copy-refused` alert). A peer that is gone for good but still in the configuration keeps the database closed on every node. The operator override is `POST /api/v1/cluster/accept-copy/{database}`:

- It requires the `root` user, runs on the leader only (a follower answers `400`), and the body is ignored. It removes the marker and accepts the leader's copy without the other servers' confirmation. It does not reopen the copy: the next request that names the database does, and every follower then installs it. The response carries `result`, `database`, `localServer`, `appliedIndex` and, when one stood, `overriddenRefusal`.
- Every use is logged at `WARNING`, naming the user, the source address, the applied index and the refusal it overrode: `Database '<name>': <user> accepted this leader's copy, which the last resync could not verify, as the cluster's copy WITHOUT the other servers' confirmation …`
- It answers `404` when this server holds no closed copy of the database marked unverified, and `500` when the marker cannot be removed, in which case the copy stays closed.

Raft layer restarted in place

A node can restart its Raft layer without restarting the process: for example to recover once a full log volume has room again (see the note on sizing the Raft storage volume under Architecture), or to reformat its Raft storage and reinstall a snapshot when it is stuck at a stale term. Since v26.10.2 `GET /api/v1/cluster` reports both, and what happened after:

- `localInPlaceRestarts` holds two counts since the process started: `recovered`, restarts that kept the Raft log, and `reformatted`, restarts that reformatted it. The same counts are published as metrics (see how-to/operations/monitoring.pdf).

- `localReplicationPathUnproven` is `true` while the node has been restarted in place and has taken no replicated entry since, nor seen a newer term with a known leader. On an idle cluster that is normal for a while.
- `localLeaderUnreachableSinceRestart` is `true` when that state outlasts the time a healthy path takes to deliver entries, while no leader has made itself known or the leader reports entries this node does not hold. It raises the `follower-leader-unreachable-since-restart` alert: `critical` when a leader is known, `warning` when none is, since a cluster that lost its quorum looks the same from every node. `raftState` reads `RUNNING` throughout, but the node does not count toward the quorum. If the alert outlasts the leader's channel reset and escalation (see Replication-Channel Self-Healing), restart the node by hand.

Since v26.10.2 a follower restarted in place is no longer reformatted automatically when it looks stuck at a stale term while its replication path is still unproven: that state also matches a node the leader's appends do not reach, and a reformat would turn it into a voter with an empty log. The reformat is held, a `WARNING` is logged every 5 minutes, and the `follower-stuck-at-stale-term` alert carries `details.replicationPathUnproven: true` with a recommendation to restart the node by hand.

8.10.32. Cluster Management REST API

Cluster membership and leadership can be changed at runtime through dedicated REST endpoints. All endpoints require authentication as the `root` user.

Method	Path	Description
GET	`/api/v1/cluster`	Return the current cluster status: current leader, peers and their roles, current term, commit and applied indices, per-follower replication lag, and the `alerts` array. Every peer carries `inConfiguration`: `true` for a member of the live Raft configuration, `false` for a peer that `arcadedb.ha.serverList` declares but the cluster no longer contains (removed with `DELETE /api/v1/cluster/peer/{peerId}`, or dropped by an earlier version while it was down). Such a peer has `role` `NOT_IN_CONFIGURATION` instead of `FOLLOWER`, and the divergence raises the `peers-not-in-configuration` alert (`critical` when the missing peer is the node answering, `warning` otherwise); a member added at runtime but absent from the server list raises the informational `peers-not-in-server-list` alert, and is listed in `peers[]` with its `httpAddress` like any other peer *(since v26.11.1: it was listed without one)*. Two peer ids on one Raft address raise the `peers-share-address` warning: only one process answers there but both ids vote, so remove the extra id with `DELETE /api/v1/cluster/peer/{peerId}`. Of two concurrent add-peer requests for one address under different ids, the second is refused. Since v26.10.2 the answer also carries `localInPlaceRestarts` (`{"recovered": <n>, "reformatted": <n>}`), `localReplicationPathUnproven` and `localLeaderUnreachableSinceRestart`, see Raft layer restarted in place.

Method	Path	Description
POST	`/api/v1/cluster/peer`	Add a peer to the running cluster. Body: `{"peerId":"<id>","address":"host:raftPort:httpPort","name":"<friendlyName>","priority":0}`. The optional `name` is the human-readable display name shown in logs and Studio. The optional `priority` (non-negative integer, default `0`) sets the new peer's leader-election priority the same way a `priority` in `arcadedb.ha.serverList` does - use `0` for a witness that must never become leader, a higher value for a preferred one. The new peer is seeded with the current user list so authentication works immediately. This call adds the named peer to **this** server's cluster; it does not make this server join another one, and an address that resolves to this server is refused with `400`. A peer that cannot join - it answers a connection but never catches up - is reported after `arcadedb.ha.membershipChangeTimeout` at the latest, and a peer that turns out to belong to a **different** cluster is refused at once, naming `arcadedb.ha.clusterName` as the thing to check.
DELETE	`/api/v1/cluster/peer/{peerId}`	Remove a peer from the cluster. The removed node also loses its access to the Raft port, including on Kubernetes, where the headless service keeps publishing a pod's address until the pod itself terminates.
POST	`/api/v1/cluster/leader`	Transfer leadership. Body: `{"peerId":"<target>","timeoutMs":30000}`. If `peerId` is omitted, Raft picks a new leader automatically. Must be sent to the leader: any other server answers `409` and names the current leader.
POST	`/api/v1/cluster/stepdown`	Make the current leader step down. The cluster elects a new leader. Must be sent to the leader: any other server answers `409` and names the current leader.

Method	Path	Description
POST	`/api/v1/cluster/leave`	Gracefully remove this server from the cluster. If the local server is the leader, leadership is transferred first. This endpoint is used by the Kubernetes `preStop` hook. Like `DELETE /api/v1/cluster/peer/{peerId}`, it is computed against the committed Raft configuration: while this server's Raft server is starting or restarting and cannot read it, both answer at once that the configuration cannot be read and to retry in a moment. *(Since v26.11.1: during that window both used the declared `arcadedb.ha.serverList` instead, and waited out `arcadedb.ha.membershipChangeTimeout` (90 s) before reporting a missing leader.)*
POST	`/api/v1/cluster/verify/{database}`	Compare component file checksums for the given database across all peers. Useful to confirm that all nodes have converged. See Verify Outcomes for the values `overallStatus` can take.
POST	`/api/v1/cluster/resync/{database}`	Discard this server's copy of the database and install a fresh snapshot from the leader. This lifts a quarantine (see Quarantined databases) and reinstalls a database the node is missing or whose snapshot install was interrupted. Refused on the leader itself: transfer leadership first. Answers `503` when no leader is reachable. The body is ignored.
POST	`/api/v1/cluster/accept-diverged/{database}`	*(Since v26.10.2.)* Accept this node's copy of the database as it is, lifting its quarantine and read floor without a resync. Only where no peer can serve a resync: on the only voter of the cluster, or when every voter holds the database quarantined; anywhere else it answers `409`. Answers `404` when nothing stands on the database. Every use is logged at `WARNING`. See Lifting a quarantine without a resync.
POST	`/api/v1/cluster/accept-stale-snapshot`	*(Since v26.10.2.)* Lift this node's node-wide stale-snapshot read floor without a resync, accepting its databases as they are; the entries between the floor and the snapshot marker are not replayed. Only on the only voter of the cluster, and not while a snapshot download is running; otherwise it answers `409`. Answers `404` when no floor stands, and `500` when the change cannot be persisted, leaving the floor in place. A database quarantined on its own keeps its quarantine. The body is ignored. Every use is logged at `WARNING`. See Lifting the stale-snapshot read floor.

Method	Path	Description
POST	`/api/v1/cluster/accept-copy/{database}`	Leader only: accept the leader's closed copy of the database, which the last resync could not verify, as the cluster's copy without the other servers' confirmation. The next request that names the database reopens it. A follower answers `400`; a server with no such copy answers `404`. The body is ignored. Every use is logged at `WARNING`. See Accepting the leader's unverified copy.
GET	`/api/v1/ha/snapshot/{database}`	Leader-only: stream a ZIP of the database for follower catch-up. Requires the cluster token and is used internally by snapshot recovery. `root□s Basic credentials are also accepted, without a cluster token: since v26.11.1 a request that carries a cluster token that does not match is answered `401` "Invalid cluster token" whatever credentials come with it, as every other cluster route does.
POST	`/api/v1/cluster/security-seed`	Leader-only: replicate the cluster security documents to the calling node. Requires the cluster token and is called by peers, not by operators. The caller may send digests of the copies it already holds, in which case a node already in step is answered without anything being written. Any other server answers `409` and names the current leader. See Cluster security documents.

Send leadership changes to the leader. `/api/v1/cluster/leader` and `/api/v1/cluster/stepdown` answer `409 Conflict` when the server receiving them is not the leader, and the message names the leader to retry against.

This matters if you call them through a Kubernetes Service or any load balancer, which spreads requests over every ready pod: previously such a request reached a follower, was routed on to the leader anyway, and caused an election you did not intend - while the follower replied `200 OK`.

Example: add a new peer at runtime.

```
$ curl -u root:<password> \
       -X POST http://leader:2480/api/v1/cluster/peer \
       -H 'Content-Type: application/json' \
       -d '{"peerId":"node4","address":"10.0.0.4:2434:2480"}'
```

Adding a peer with a human-readable name:

```
$ curl -u root:<password> \
       -X POST http://leader:2480/api/v1/cluster/peer \
```

```
       -H 'Content-Type: application/json' \
       -d
'{"peerId":"10.0.0.4_2434","address":"10.0.0.4:2434:2480","name":"frankfurt"}'
```

Adding a peer as a non-voting witness, which will never become leader:

```
$ curl -u root:<password> \
       -X POST http://leader:2480/api/v1/cluster/peer \
       -H 'Content-Type: application/json' \
       -d '{"peerId":"10.0.0.4_2434","address":"10.0.0.4:2434:2480","priority":0}'
```

8.10.33. Verify Outcomes

`POST /api/v1/cluster/verify/{database}` reports one `status` per peer and rolls them up into a single `result.overallStatus`:

`overallStatus`	**Meaning**
`ALL_CONSISTENT`	Every peer was contacted and every peer's checksums match this node's.
`INCONSISTENCY_DETECTED`	At least one peer was contacted and its checksums differ - a divergence somebody has actually observed.
`VERIFICATION_INCOMPLETE`	No divergence was observed, but at least one peer could not be verified: it was unreachable, or its address does not identify it (see Peer-to-Peer Endpoints). Such a peer is reported with `status: ERROR` and an `error` naming what to fix. It is also reported when the TimeSeries sealed stores could not be fully compared - see below.

`ALL_CONSISTENT` therefore means what it says: it is not reachable while any peer is unverified. An unverified peer is deliberately **not** rolled up as an observed divergence, which would send an operator hunting for something nobody saw.

Alerting that keys on `overallStatus != "ALL_CONSISTENT"` is unaffected by the third value. Alerting that keys on `overallStatus == "INCONSISTENCY_DETECTED"` must add `VERIFICATION_INCOMPLETE` to keep catching an unreachable or unidentifiable node.

Files left out of the comparison

Some files in a database directory belong to the node that holds the copy, not to the data: two healthy nodes at the same commit can hold different ones, or one can have none. Since v26.10.2 they are left out of the checksums each node reports (`GET /api/v1/ha/snapshot/{database}/checksums`), so the verify no longer reports them as a divergence on a healthy cluster:

- `statistics.json`, and `cached-count.json` from earlier releases: cached record counts and free-space hints, written when the node last closed the database;
- `database.lck`, the exclusive-open marker;

- a Graph Analytical View's persisted CSR (`gav-<view>.csr`) and its contraction-hierarchy orders (`.cchorder`), a cache written only on the node that was leader when it closed;
- a vector index's proximity graph and the files derived from it (`.vecgraph`, `.vecgraphfp`, `.vecordmap` and `.vecpq`), which every node builds on its own. The vectors themselves live in the index pages, which are compared;
- `last-tx-id.bin`, the highest transaction id the node assigned when it last rotated its WAL or closed.

TimeSeries Sealed Stores

The verify also compares the TimeSeries sealed stores (`.ts.sealed`), the files holding compacted historical samples. They were previously left out, which mattered because they are replicated by their own path rather than with the data pages: a node whose sealed store failed to install, or was repaired by hand, reported a perfect match while holding different history from the leader.

Two fields report how far that comparison got:

Field	Meaning
`peers[].uncomparedSealedStores`	The peer runs an older build that does not report these files, so they were left out of the comparison with it. Present only during a rolling upgrade; the value is how many were skipped.
`result.incompleteSealedStores`, `peers[].incompleteSealedStores`	A sealed store could not be read, so that side's answer is short of one. The peer is not rolled up as agreeing, and `overallStatus` becomes `VERIFICATION_INCOMPLETE`.

Both mean the same thing for alerting: a `CONSISTENT` that did not compare every sealed store is a weaker statement than one that did, so neither is reported as `ALL_CONSISTENT`.

8.10.34. Studio Cluster Dashboard

ArcadeDB Studio exposes a **Cluster** tab (visible when HA is enabled) that displays the current leader, peer roles, Raft term and commit index, per-follower replication lag, and provides buttons for leadership transfer, peer add/remove, and database verification. See Studio.

8.10.35. Security

Cluster token

Inter-node HTTP forwarding (replica → leader proxy, snapshot downloads) is authenticated with a shared **cluster token** sent as the `X-ArcadeDB-Cluster-Token` HTTP header. If `arcadedb.ha.clusterToken` is not set explicitly, the token is derived deterministically from `arcadedb.ha.clusterName` and the `root` password via PBKDF2-HMAC-SHA256 — it is **not** persisted on disk. Every node that starts with the same cluster name and the same root password ends up with the same token automatically. *(Since v26.11.1)* The root password can come from the inline `arcadedb.server.rootPassword` setting or from the file named by `arcadedb.server.rootPasswordPath`. The inline value is checked first, then the file, and both are read the same way: one trailing CR/LF is dropped, so a node given the password inline and a

node reading it from a file derive the same token. An explicit `arcadedb.ha.clusterToken` or `arcadedb.ha.clusterTokenPath` takes precedence. A node started with only `rootPasswordPath` forms a cluster.

Token fingerprint

A node whose token is derived from the root password records a fingerprint of it (a SHA-256 under a domain prefix, never the token or the password) in `<raft-storage>.cluster-token-fingerprint`, next to its Raft storage directory, with mode 0600. The file must be writable, otherwise startup fails with a `ConfigurationException`. If the password (file or inline setting) later derives a different token, for example because a Kubernetes Secret was rotated, the node refuses to start with a message that names the setting the password came from. Without this check, the nodes on the new token and those on the old one refuse each other's requests with `401` during a rolling restart. To rotate the root password on purpose, either restart every node together with `arcadedb.ha.clusterTokenAllowRotation=true` and then remove the setting, or move to an explicit `arcadedb.ha.clusterTokenPath`. A token that is not derived from the password clears the record. A node that does not persist its Raft storage has no fingerprint to compare against, so a token that differs from the group's is refused per request, as before. A node that already ran on the inline source has no record yet, so a rotation made before the upgrade cannot be detected at its first start. A refused cluster token answers `401` "Invalid cluster token", and the refusing node logs a WARNING, at most once every 60 seconds, that names the caller and where its own token comes from.

Set `arcadedb.ha.clusterToken` (or `HA_CLUSTER_TOKEN`) explicitly when you need to rotate the token without changing the root password, or when each peer has a different root password. To keep the secret off the command line (and out of `ps`//`proc/<pid>/cmdline`), set `arcadedb.ha.clusterTokenPath` to a file the server reads at startup, e.g. a Kubernetes Secret mounted on `tmpfs`. It is read only when `arcadedb.ha.clusterToken` is not set, and its content is trimmed of surrounding whitespace so a trailing newline does not break the token comparison between nodes.

Peer allowlist

Inbound Raft gRPC connections are filtered against the DNS-resolved hosts in `arcadedb.ha.serverList`; connections from other addresses are rejected. This closes the "any host that knows the port can inject log entries" vector. It is, however, IP-based, so a spoofed source address or a compromised peer defeats it. Treat it as a best-effort default rather than a substitute for TLS: on an untrusted network, enable mTLS on the Raft gRPC transport, which binds peer identity to a certificate instead of to an address.

On Kubernetes a pod's headless-service DNS record is only published once the pod is `Ready`, so peers routinely come up before each other's names resolve and the allowlist starts out incomplete. To avoid a self-inflicted partition during that window, the filter is hardened in three ways:

- **Fast convergence**: while the allowlist has never been complete, a connection from an unknown address re-resolves DNS on a short floor (~1s) instead of waiting the full `arcadedb.ha.grpcAllowlistRefreshMs` interval.
- **Sticky entries**: when a host that resolved before temporarily stops resolving (transient DNS outage, pod-IP churn mid-restart), its last-known-good IPs are retained for `arcadedb.ha.peerAllowlistStickyTtlMs` (default 5 min) rather than being evicted immediately.

- **Startup fail-open grace**: until every peer host has resolved at least once, and for at most `arcadedb.ha.peerAllowlistStartupGraceMs` (default 60s) from startup, an unmatched address is accepted with a warning instead of rejected. Once the peer set has fully resolved once, or the window elapses, the filter enforces strictly and does not fail open again.

  For a security-strict deployment that should reject from the very first connection, set `arcadedb.ha.peerAllowlistStartupGraceMs=0`. If you previously saw `Rejecting Raft gRPC connection from non-peer address` storms with peers failing to form a cluster on startup, this hardening removes that failure mode.

Cluster security documents

The server users, groups and API tokens are held in `server-users.jsonl`, `server-groups.json` and `server-api-tokens.json`, in the server's `config/` directory. They live outside the databases, so a database snapshot does not carry them: they are replicated to the other nodes as their own Raft entries.

A node is seeded with the current set when it is admitted to the cluster, and also re-seeds itself when it rejoins one it never left. That second case covers a Kubernetes pod whose data volume is persistent but whose `config/` directory is not: it comes back as an existing cluster member, so it is never re-admitted. A node asks the leader for the current documents shortly after it first sees one, and again after it receives a snapshot, sending digests of the copies it already holds so that a node already in step costs one HTTP request and no Raft entries.

With `arcadedb.server.readinessRequiresHA` on, a node that joined at runtime or was caught up by a snapshot is held NOT READY until its documents converge, for at most `arcadedb.ha.securityConvergenceReadinessTimeout` (30 s by default), so a load balancer does not route logins to it while it still enforces stale credentials. See security-convergence readiness.

A node that is already leading the cluster by the time its catch-up runs has no peer to ask, so its own copy becomes the cluster's reference. It says so once, at `INFO`, starting with `This node leads the cluster by the time its security catch-up ran`. If that node's `config/` directory was restored out of band, re-apply the security changes made while it was away.

8.10.36. TLS/SSL for peer-to-peer transfers

This section covers the **HTTP side channels** only — every node-to-node call that is not Raft itself. It does not encrypt the Raft gRPC port, which carries AppendEntries, RequestVote and the Ratis admin/client calls. That is a separate axis, configured with `arcadedb.ha.tls.*`; see mTLS for the Raft gRPC transport. Enabling one does nothing for the other.

When `arcadedb.ssl.enabled=true`, the server binds an HTTPS listener on `arcadedb.server.httpsIncomingPort` (default `2490`) **in addition to** the plain HTTP listener on `arcadedb.server.httpIncomingPort` (default `2480`). The two listeners use different ports.

The side channels covered here are the snapshot download, the cross-node verify, the follower-to-leader command forward, the peer capability and auth-session probes, the stalled-replica resync,

the bootstrap-state probe, the security-document seed and the remote shutdown command. Every one of them carries the cluster token, so on a cluster with `arcadedb.ssl.enabled` they should all travel encrypted. To send them encrypted, declare each peer's HTTPS port in `arcadedb.ha.serverList` using the `https` field (object form) or the fifth positional field:

```
$ bin/server.sh -Darcadedb.ha.enabled=true \
                -Darcadedb.ssl.enabled=true \
                -Darcadedb.ssl.keyStore=/etc/arcadedb/keystore.p12 \
                -Darcadedb.ssl.keyStorePassword=... \
                -Darcadedb.ssl.trustStore=/etc/arcadedb/truststore.jks \
                -Darcadedb.ssl.trustStorePassword=... \
                -Darcadedb.ha.serverList=db1:{raft:2434,http:2480,https:2490},db2:
{raft:2434,http:2480,https:2490},db3:{raft:2434,http:2480,https:2490}
```

The peer certificate is validated against the configured **trust store** (`arcadedb.ssl.trustStore`), and its name is verified against the name the peer was dialled by.

The HTTPS listener is configured with a single keystore (`arcadedb.ssl.keyStore`), so it serves **one** identity, and the name a dial advertises in SNI selects nothing: every name is answered with the same certificate. The name a peer is dialled by is the one in `arcadedb.ha.serverList`, or the peer's Raft host when the entry omits an explicit endpoint. So every name any node can be reached by has to be a subject alternative name of that one certificate. On Kubernetes that usually means listing the pod's DNS name, the headless-service name and, if you declare peers by address, the pod IP. A name that is missing fails the handshake at the first peer-to-peer call rather than at startup.

If the `https` port is omitted on a homogeneous cluster (every node listens on the same HTTPS port, e.g. a Kubernetes StatefulSet), each peer's HTTPS endpoint is derived automatically from the peer host plus this node's local HTTPS port, so no extra configuration is needed. Only when no HTTPS endpoint can be resolved does the transfer fall back to plain HTTP. The plain listener is the one that is always bound, so falling back keeps a cluster that never declared its `https` ports able to bootstrap, resync and be administered — but it does put the cluster token on the wire in clear text, so it is announced once per JVM:

```
WARNING: SSL is enabled but no HTTPS address is known for a peer; probing its
bootstrap-state over plain HTTP,
which puts the cluster token on the wire in clear text. Declare each node's 'https'
port in
arcadedb.ha.serverList to avoid it.
```

One such line is logged per JVM for the whole family of side channels, naming whichever happened to fall back first; the remedy is the same for all of them. Seeing it on a cluster you believe is encrypted end to end means at least one peer has no resolvable HTTPS endpoint — check that the `https` field is declared for every entry, and see Peer-to-Peer Endpoints for the case where it is declared but shared between two nodes.

Declaring an HTTPS port that points at the plain HTTP port (e.g. `https:2480`) causes a TLS handshake failure (`Unsupported or unrecognized SSL message`). Always point `https` at the HTTPS listener's port.

8.10.37. mTLS for the Raft gRPC transport

The previous section covers the HTTP side channels. This one covers the **Raft gRPC port** (`arcadedb.ha.raftPort`, default `2434`), which carries AppendEntries, RequestVote, InstallSnapshot and the Ratis admin/client calls.

The two are independent axes. `arcadedb.ssl.*` does nothing for the gRPC port, and `arcadedb.ha.tls.*` does nothing for the HTTP listeners; a hardened cluster wants both. Nor does gRPC TLS replace the `X-ArcadeDB-Cluster-Token` check described in Security — that check still authenticates the HTTP forwarding and snapshot-download traffic.

TLS on the Raft port is **off by default**, so a development or test cluster still starts with no configuration at all. Turn it on with four settings:

```
$ bin/server.sh -Darcadedb.ha.enabled=true \
                -Darcadedb.ha.tls.enabled=true \
                -Darcadedb.ha.tls.certChainFile=/etc/arcadedb/tls/tls.crt \
                -Darcadedb.ha.tls.privateKeyFile=/etc/arcadedb/tls/tls.key \
                -Darcadedb.ha.tls.trustCertCollectionFile=/etc/arcadedb/tls/ca.crt \
                -Darcadedb.ha.serverList=db1:2434:2480,db2:2434:2480,db3:2434:2480
```

One setting covers all three gRPC endpoints (admin, client, and server-to-server), so there is nothing further to switch on per message type. Mutual authentication is on by default; see Server-only TLS to relax it.

When `arcadedb.ha.tls.enabled=true`, all three file settings must name a readable PEM file or **the node refuses to start**, with a `ConfigurationException` naming the offending setting:

```
arcadedb.ha.tls.certChainFile must be set when arcadedb.ha.tls.enabled is true
arcadedb.ha.tls.privateKeyFile (/etc/arcadedb/tls/tls.key) is not a readable file:
Raft gRPC TLS cannot be initialized
```

The paths are validated at startup rather than left to the TLS layer, so a typo or an unmounted Secret volume fails once and clearly, instead of surfacing later as an opaque handshake error on every peer connection.

The private key must be **PKCS#8** — the PEM whose armor labels it simply `PRIVATE KEY`. A PKCS#1 key, which the armor labels `RSA PRIVATE KEY` instead and which several toolchains emit by default, is not accepted. Convert one with `openssl pkcs8 -topk8 -nocrypt`.

What the certificate has to look like

The startup checks confirm the three files are **readable**. They cannot confirm the certificate is the

right **kind** of certificate. Both mistakes below produce a TLS handshake failure at the first peer connection, not a startup error, so they are worth getting right up front.

Both the `serverAuth` and `clientAuth` extended key usages

Every node both accepts and initiates Raft connections. With `arcadedb.ha.tls.mutualAuth=true` (the default), the certificate in `certChainFile` is presented as the TLS **server** certificate to peers that dial in **and** as the **client** certificate when this node dials out — one certificate, two roles. A CA configured to issue single-purpose certificates therefore produces a file that is perfectly readable and simply the wrong kind. Ask for both EKUs explicitly; do not assume the issuer's default profile includes them.

A subject alternative name matching the address peers dial

The SAN must match the host as it appears in the other nodes' `arcadedb.ha.serverList`, not whatever the node calls itself locally. On Kubernetes that is the pod's stable headless-service DNS name (`my-arcadedb-0.my-arcadedb.arcadedb.svc.cluster.local`), never the pod IP, which changes on every restart.

`certChainFile` holds this node's certificate followed by any intermediate CA certificates, in that order. `trustCertCollectionFile` holds the cluster CA certificate(s) that sign every node certificate: a peer presenting a certificate that does not chain to this collection is rejected during the handshake, before any Raft message is read.

Server-only TLS

Setting `arcadedb.ha.tls.mutualAuth=false` encrypts the traffic but stops requiring a client certificate from peers that dial in. That leaves the Raft port open to any client that trusts the cluster CA, which is the back door mutual authentication exists to close, so the server logs a `WARNING` at startup:

```
Raft gRPC TLS is enabled with arcadedb.ha.tls.mutualAuth=false: the transport is
encrypted but the
dialling peer is NOT authenticated, so any client trusting the cluster CA can still
open a Raft stream
```

Turning mutual authentication off does **not** make `arcadedb.ha.tls.trustCertCollectionFile` optional. It only governs what this node demands of peers dialing **in**; when this node dials **out** it still needs the cluster CA to validate the certificate the other node presents. All three file settings stay mandatory whenever `arcadedb.ha.tls.enabled=true`.

Private key permissions

The server logs a `WARNING` when the private key file is readable by the group or by others, because any local account that can read it can present this node's identity to the cluster:

```
The Raft gRPC private key arcadedb.ha.tls.privateKeyFile (/etc/arcadedb/tls/tls.key)
is readable
beyond its owner: any local account that can read it can present this node's identity
```

```
to the
cluster. Restrict it to owner-only (chmod 600)
```

It is deliberately a warning and not a refusal: a container image or a mounted Secret may arrive with permissions the node cannot change, and refusing to start would be the worse outcome. The check is a no-op on filesystems with no POSIX permission view.

Issuing a cluster CA and node certificates with openssl

For a fixed-size cluster outside Kubernetes, one self-signed CA plus one certificate per node is enough. Create the CA once, on a host that is not a cluster node:

```
$ openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 -nodes \
      -keyout cluster-ca.pem -out ca.crt \
      -subj "/CN=arcadedb-cluster-ca"
```

Then, for each node (`db1` here), generate a key and a certificate signing request:

```
$ openssl req -newkey rsa:2048 -nodes \
      -keyout db1-pkcs1.pem -out db1.csr \
      -subj "/CN=db1"
```

Sign it with an extension file that carries **both** EKUs and the SAN peers will dial. This file is the part that is easy to get wrong:

db1.ext

```
basicConstraints = critical,CA:FALSE
keyUsage         = critical,digitalSignature,keyEncipherment
extendedKeyUsage = serverAuth,clientAuth
subjectAltName   = DNS:db1,DNS:db1.internal.example.com
```

```
$ openssl x509 -req -in db1.csr -CA ca.crt -CAkey cluster-ca.pem -CAcreateserial \
      -days 365 -sha256 -extfile db1.ext -out db1.crt
```

Finally, convert the key to PKCS#8 and lock it down:

```
$ openssl pkcs8 -topk8 -nocrypt -in db1-pkcs1.pem -out db1-key.pem
$ chmod 600 db1-key.pem
```

Node `db1` is then configured with `certChainFile=db1.crt`, `privateKeyFile=db1-key.pem` and `trustCertCollectionFile=ca.crt`. Every node in the cluster gets the same `ca.crt` and its own certificate and key.

Confirm a certificate before rolling it out — the two lines that matter are the SAN and the EKU:

```
$ openssl x509 -in db1.crt -noout -ext subjectAltName,extendedKeyUsage
$ openssl verify -CAfile ca.crt db1.crt
```

Issuing certificates with cert-manager

On Kubernetes, cert-manager is the natural issuer: it writes a `Secret` with exactly the three files these settings need (`tls.crt`, `tls.key`, `ca.crt`) and renews it automatically.

Start from a CA `Issuer` backed by a self-signed root, or by your existing PKI:

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: arcadedb-cluster-ca
  namespace: arcadedb
spec:
  ca:
    secretName: arcadedb-cluster-ca-tls
```

Then one `Certificate` per pod, named for the pod's headless-service DNS record:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: arcadedb-raft-0
  namespace: arcadedb
spec:
  secretName: arcadedb-raft-0-tls
  duration: 2160h    # 90 days
  renewBefore: 360h  # 15 days
  privateKey:
    algorithm: RSA
    size: 2048
    encoding: PKCS8           # cert-manager defaults to PKCS1, which is not accepted
  usages:
    - server auth             # both are required: every node dials and is dialled
    - client auth
    - digital signature
    - key encipherment
  dnsNames:
    - my-arcadedb-0.my-arcadedb.arcadedb.svc.cluster.local
  issuerRef:
    name: arcadedb-cluster-ca
    kind: Issuer
```

`privateKey.encoding` must be set to `PKCS8` explicitly. cert-manager's default is `PKCS1`, which the Raft transport does not parse.

Two `usages` entries do the work of the `extendedKeyUsage` line in the openssl recipe: `server auth` and `client auth`. Omitting either one is the single-purpose-certificate mistake, and it will only show up as a handshake failure between two running nodes.

Per-pod `Certificate` objects give each node its own identity, which is what makes a compromised node's certificate revocable on its own. The shortcut of one shared `Certificate` listing every pod's DNS name works, and is simpler for a fixed-size cluster, but every node then holds an identity valid for every other node.

With HashiCorp Vault as the issuer instead, the equivalent of the two EKUs is a PKI role created with both flags set:

```
$ vault write pki/roles/arcadedb-raft \
      allowed_domains="my-arcadedb.arcadedb.svc.cluster.local" \
      allow_subdomains=true \
      server_flag=true \
      client_flag=true \
      key_type=rsa key_bits=2048 \
      max_ttl=2160h
```

Mounting the Secret

Mount the certificate `Secret` read-only and point the three settings at the mount path. Using the Helm chart, append to `volumes` and `volumeMounts`, and set the values via `arcadedb.extraCommands`:

```
volumes:
  - name: raft-tls
    secret:
      secretName: arcadedb-raft-0-tls
      defaultMode: 0400
volumeMounts:
  - name: raft-tls
    mountPath: /etc/arcadedb/tls
    readOnly: true

arcadedb:
  extraCommands:
    - -Darcadedb.ha.tls.enabled=true
    - -Darcadedb.ha.tls.certChainFile=/etc/arcadedb/tls/tls.crt
    - -Darcadedb.ha.tls.privateKeyFile=/etc/arcadedb/tls/tls.key
    - -Darcadedb.ha.tls.trustCertCollectionFile=/etc/arcadedb/tls/ca.crt
```

`defaultMode: 0400` keeps the private-key permission warning quiet. This is the same pattern as `arcadedb.ha.clusterTokenPath` in Security: the secret stays off the command line and out of `ps`.

Because each pod needs the certificate issued for **its own** DNS name, a per-pod `Secret` cannot be mounted from a single chart-wide value. Either render one `StatefulSet`-per-replica, or use cert-manager's csi-driver, which issues a short-lived certificate into an ephemeral volume at pod start

and needs no `Secret` at all.

No extra configuration is needed for Kubernetes auto-join: the join probe is built from the same transport parameters as the local server, so it speaks TLS exactly when the cluster does. See High Availability on Kubernetes.

Known limitations

No certificate rotation without a restart

The PEM files are read once, when the Raft transport is built. Renewing a certificate — including a cert-manager renewal that rewrites the mounted `Secret` in place — takes effect only after the node is restarted. Roll the nodes one at a time and wait for each to rejoin, so the cluster keeps its quorum throughout. Choose a `duration` that makes the restart cadence acceptable.

No KMS integration

The private key is read from a file on disk. There is no HSM, PKCS#11 or cloud-KMS path; protecting the key is the deployment's responsibility (owner-only permissions, a `tmpfs`-backed mount, or a csi-driver volume that never touches durable storage).

8.10.38. Kubernetes

For Kubernetes deployments using the official Helm chart — including the `StatefulSet` + headless `Service` pattern, auto-join on scale-up, and the `preStop` auto-leave hook — see High Availability on Kubernetes in the Kubernetes guide.

8.10.39. Troubleshooting

Performance: insertion is slow

ArcadeDB uses an optimistic concurrency model: if two threads try to update the same page, the first wins, the second throws a `ConcurrentModificationException` and the client retries (configurable number of times). In HA mode the retry window is wider because file locks are held during the Raft round-trip, and the same rule applies across nodes: two nodes writing the same page concurrently are ordered by the leader, and the second one is refused with the same retryable error.

If you are inserting many records in parallel, allocate one bucket per thread to eliminate contention. Example for the vertex type `User`:

```
ALTER TYPE User BucketSelectionStrategy `thread`
```

With enough buckets, parallel insertions avoid page contention entirely and do not hit the retry path.

A database is missing after a restart, and the log names `.snapshot-pending`

A follower that is killed while it is installing a snapshot from the leader — a liveness-probe kill, an OOM kill, a node drain — leaves `databases/<name>/` holding a mix of the previous database and the incoming one. It is neither, and opening it would register and serve that mix. The installer marks

the directory with a `.snapshot-pending` file before it touches a single byte and removes the file only once the installed copy has been reopened successfully, so the marker's presence is the node's own record that the install did not finish.

(Since v26.11.1) A snapshot install is also verified before the previous copy is given up. The leader sends the name and size of every file, and the install compares the live files against that account right after the swap and before the reopen. On a mismatch, such as zero-length page files or a corrupt `schema.json`, it rolls back to the previous copy. The leader's file account is kept as `.snapshot-manifest` through the swap, so the crash-recovery roll-forward compares against it too and restores the backup instead of keeping a half-installed copy. An install without a manifest falls back to the check that the database reopens.

While that marker exists, the database is **not opened by anything**: not the startup scan, not the default-database configuration (`arcadedb.server.defaultDatabases`), and not a client request naming it. A request for it is answered with `Database '<name>' is not available: an interrupted HA snapshot install left '.snapshot-pending' in its directory`. The refusal lasts as long as the marker does — it is a statement about the directory, not about a phase of startup.

The node still starts. HA snapshot recovery runs during startup and either completes the interrupted swap (the download had finished) or rolls it back to the previous copy (it had not); either way it removes the marker, and the database is opened before the server reports itself online. In the normal case the only trace is one INFO line naming the deferred database.

This includes a swap interrupted by an earlier release that recorded no phase: recovery works out from the files whether the new copy was already going in (it finishes the install) or the old copy was still being moved aside (it restores the old database). In the rare case a file of the incoming copy has to be set aside to let the old database open, it is moved to `databases/<name>/.snapshot-orphans/` and a WARNING names it; it is safe to delete once the database is verified.

The marker survives startup in two cases:

- **The node is not running HA** — `arcadedb.ha.enabled=false`, or the `arcadedb-ha-raft` module is absent from an embedded deployment. Nothing runs the recovery pass, so nothing reconciles the directory.
- **Recovery ran and deliberately declined to act.** When there is neither a completed download nor a backup to restore from, and the directory does not hold a loadable schema, recovery leaves it and the marker untouched for inspection rather than blessing an incomplete directory as a healthy database.

Both are reported at SEVERE, naming the database:

```
Database '<name>' was NOT opened: snapshot recovery did not clear its '.snapshot-
pending' marker,
so its directory is still neither the previous database nor the installed snapshot.
```

To resolve it, let the cluster replace the copy: on a follower joined to a healthy cluster, `POST /api/v1/cluster/resync/{database}` reinstalls the database from the leader, reconciling the interrupted install before it downloads anything. It works on a deferred database — the resync

never needs to open the local copy first. If the node cannot reach a leader, or it is running standalone, the directory has to be dealt with on the volume: `databases/<name>/.snapshot-backup/` holds the previous copy when one was taken, and removing `databases/<name>/` entirely lets the node re-acquire the database from the leader on the next reconcile. Inspect before deleting — when no backup was taken, that directory is the only copy the node has.

Deleting `databases/<name>/` and restarting now reliably brings the database back on a node that had it before. Earlier versions could restart without it: the node kept a record, stored next to the database directories rather than inside them, saying the database had already been checked, so deleting the directory left that record behind and the node concluded there was nothing to do. It logged that the check had completed and carried on without the database.

A node that cannot complete the reinstall — no leader reachable yet, for example — now reports the database in `GET /api/v1/cluster` and retries periodically until it succeeds, instead of running short of it silently.

Writes on a replica fail with a retry error during a schema change

While the leader runs a schema change (for example `CREATE INDEX` on a large type), or drops or reinstalls a database, it refuses write transactions that other nodes send for that database with a retryable error, and accepts them again as soon as the operation ends. Retry the write; a long index build means retries for as long as it runs.

A write on a replica is also refused with a retryable error when its transaction began before the replica had applied a schema change the leader has already applied (for example a new index), so that it cannot leave a record out of that index. The retry waits for the schema change to be applied locally and then succeeds. Callers that do not retry see a `ConcurrentModificationException` during DDL bursts; set transaction retries accordingly. One peer that is older, not yet probed or offline turns this protection off for every writer, and it is best-effort across a leader restart. Set `arcadedb.ha.txSchemaCheck=false` on the writing nodes to switch it off.

A node whose Raft state machine is closed, or is being restarted in place, refuses every commit at once with a retryable `NeedRetryException`: `Database '<name>' cannot commit on this server: its Raft state machine is closed and has not been restarted yet, so a committed entry could not be applied here. Retry shortly, or send the request to another server of the cluster`. Retry on the same node once the restart is done, or on another node. *(Since v26.10.2: such a commit, a leader's included, waited for the whole quorum timeout before it failed, every commit for as long as the state machine stayed closed.)*

A node logs that it sent a Raft request "to itself", or refused one "addressed to" another peer

A node answers a Raft request only when it is the peer the request was addressed to, and refuses it otherwise. The refusal is logged once a minute per peer, in one of two forms:

```
This node sent a Raft appendEntriesAsync meant for peer 'arcade2_2434' to itself and
refused it (issue #10016):
the address of 'arcade2_2434' resolves to arcade2=[127.0.0.1], which leads back to
this node. ...
```

```
Refused a Raft requestVote from peer 'arcade1_2434' addressed to peer 'arcade2_2434'
(issue #10016):
the address 'arcade1_2434' holds for 'arcade2_2434' leads to this node instead. ...
```

Both mean the same thing: the host name of a peer in `arcadedb.ha.serverList` resolves to the wrong node. The usual cause is a container that was stopped or disconnected: on some container platforms its name stops resolving to its old address and resolves to the loopback address or to another container instead. The first form names the node that holds the wrong address (the one logging it). The second names it in the message, and that node is the one to inspect.

Nothing needs to be done for the cluster to keep working. The surviving majority keeps its leader, and a node that dialled itself rebuilds its connection to that peer every few seconds, so the peer is reached again as soon as its name resolves correctly. If the message persists while the peer is running, check what its name resolves to from inside the node that logs it (for example `getent hosts <name>` in the container), and the DNS configuration or the server list.

The same refusal applies when the nodes disagree about a peer's id, which is derived from its `host:raftPort` entry in the server list: a node whose own entry differs from the one the other nodes hold for it is unreachable to them, and the message names the id they used. Make every node's server list spell each peer the same way.

(Since v26.10.2.) Earlier versions answered such a request in the name of the peer it was meant for. A leader that received its own heartbeat stepped down, and a candidate that received its own vote request failed every election with `java.lang.IllegalStateException` at `RaftConfigurationImpl.hasMajority`. The cluster then had no leader until the surviving nodes were restarted.

Verbose HA logging

Enable detailed HA logging for diagnostics via `arcadedb.ha.logVerbose`:

- `0` — off (default).
- `1` — basic: elections, leader changes, peer membership.
- `2` — detailed: commands, WAL replication, schema.
- `3` — trace: every state machine apply.

Ratis warning: snapshot "has missing MD5 file"

Earlier versions log this WARNING on every checkpoint and every restart, more often the longer a node runs:

```
Snapshot file SingleFileSnapshotInfo(t:1, i:7293):[.../sm/snapshot.1_7293] has missing
MD5 file.
```

It never meant corruption. ArcadeDB's real snapshot is the set of database files the transaction manager has already flushed to disk. The file Ratis inspects is a zero-byte marker whose name

carries the `(term, index)` pair used to decide which Raft log segments can be purged, and it deliberately has no `.md5` companion. The warning is filtered out (see Per-package verbosity); other messages from the same Ratis logger are still logged.

8.10.40. HA Settings

The following settings control HA behavior. A complete list of all HA parameters is available in Server Settings.

Setting	Description	Default Value
`arcadedb.ha.enabled`	Enables HA for this server	false
`arcadedb.ha.clusterName`	Cluster name. Useful when running multiple clusters in the same network	arcadedb
`arcadedb.ha.serverList`	Comma-separated list of peers. Each entry uses the readable object form `[name@]host:{raft:2434,http:2480,https:2490,priority:10}` (recommended) or the positional form `[name@]host:raftPort:httpPort[:priority[:httpsPort]]`; see Server List Entry Format. The optional `name@` prefix gives the peer a human-readable name used in logs and Studio; see Named Peers. The optional priority (integer, default 0) prefers higher-valued nodes during elections. The optional `https` port encrypts peer-to-peer transfers when `arcadedb.ssl.enabled=true`; see TLS/SSL. Example: `frankfurt@db1:{raft:2434,http:2480,https:2490,priority:10},london@db2:{raft:2434,http:2480,https:2490}`	(empty)
`arcadedb.ha.txSchemaCheck`	Replicated transactions state the Raft index their node had applied when they began, so the leader refuses (retryably) a replica transaction prepared before a schema change it already applied. See the Troubleshooting section	true

Setting	Description	Default Value
`arcadedb.ha.raftPort`	Default Raft gRPC port used when an entry in `serverList` does not specify one	2434
`arcadedb.ha.quorum`	Write quorum: `majority` or `all`	majority
`arcadedb.ha.quorumTimeout`	Timeout in ms waiting for the quorum acknowledgment	10000
`arcadedb.ha.schemaDelta`	Ship schema changes as a delta instead of the whole schema document, once every peer has confirmed it can decode one. See Schema Delta Replication	true
`arcadedb.ha.schemaIncrementalApply`	Followers apply a DDL entry by creating only the components for the files it added, instead of rebuilding all of them. See Schema Replication	true
`arcadedb.ha.readConsistency`	Default read consistency for follower reads: `eventual`, `read_your_writes`, `linearizable`	read_your_writes
`arcadedb.ha.electionTimeoutMin` / `arcadedb.ha.electionTimeoutMax`	Minimum and maximum Raft election timeout in ms. Increase for WAN clusters	2000 / 5000
`arcadedb.ha.serverRole`	Node role: `any` (default) or `replica` (pinned follower, never elected leader)	any
`arcadedb.ha.snapshotThreshold`	Number of Raft log entries after which the leader takes a snapshot	100000
`arcadedb.ha.bootstrapFromLocalDatabase`	When `true` (default), at first cluster formation peers exchange `(fingerprint, lastTxId)` per database and the highest-`lastTxId` peer is elected as the bootstrap source. See Offline Cluster Bootstrap.	true
`arcadedb.ha.bootstrapTimeoutMs`	Maximum time in ms the bootstrap leader waits for every peer in `serverList` to report its bootstrap state before proceeding.	120000

Setting	Description	Default Value
`arcadedb.ha.groupCommitQueueSize` / `arcadedb.ha.groupCommitOfferTimeout`	Bounded queue size and offer timeout (ms) for Raft group-commit backpressure	10000 / 100
`arcadedb.ha.raftPersistStorage`	If true, the Raft storage directory is preserved across restarts, enabling rejoin by replaying the persisted log instead of a full snapshot resync. **Defaults to `true`**; ensure `raftStorageDirectory` lives on durable storage. See Durable Raft Storage	true
`arcadedb.ha.raftStorageDirectory`	Parent directory for the per-node `raft-storage-<nodeName>` sub-folder. Empty (default) uses the server root path. With durable storage enabled by default, point this at a persistent volume. See Durable Raft Storage	(empty)
`arcadedb.ha.peerUnreachableThreshold`	Time in ms since the last successful RPC to a follower before the leader reports it as unreachable. Diagnostic only — it does not change membership or quorum — but it is the signal the channel reset below depends on. 0 disables it, and with it the whole self-healing sequence. See Replication-Channel Self-Healing	10000
`arcadedb.ha.peerChannelResetDuration`	Time in ms a follower must stay continuously unreachable before the leader resets that one follower's replication channel, forcing a fresh DNS resolution and reconnect. Retried once per interval, up to 5 attempts. 0 disables. See Replication-Channel Self-Healing	60000

Setting	Description	Default Value
`arcadedb.ha.peerChannelResetEscalation`	When the reset retry budget is exhausted and the channel is still dead, transfer leadership to a healthy peer so the new leader builds a fresh appender to that follower. Bounded by a 30-minute per-follower cooldown on each peer. `false` logs only. See Replication-Channel Self-Healing	true
`arcadedb.ha.stalledReplicaResyncDurationMs`	How long in ms a replica must stay STALLED (its `matchIndex` not advancing while the leader commits) before the leader forces it to resync. A follower still at `matchIndex=-1` counts as STALLED regardless of the numeric lag. 0 disables leader-driven recovery; the condition is still logged	60000
`arcadedb.ha.stopServerOnReplicationFailure`	If true, the JVM exits after exhausting step-down retries on a phase-2 replication failure. Default false keeps the server up and logs CRITICAL	false
`arcadedb.ha.clusterToken`	Shared secret for inter-node authentication. If empty, the token is derived at every start from `arcadedb.ha.clusterName` and the root password, see Security	(empty)
`arcadedb.ha.clusterTokenAllowRotation`	Lets a node whose derived cluster token changed since its last start (root password file or inline root password rewritten) start anyway, and records the new fingerprint. Default false: the node refuses to start. See Security. *Since v26.11.1*	false

Setting	Description	Default Value
`arcadedb.ha.clusterTokenPath`	Path to a file containing the cluster token. Read only when `arcadedb.ha.clusterToken` is not set; the content is trimmed. Keeps the secret off the command line (e.g. a Kubernetes Secret mounted on `tmpfs`)	null
`arcadedb.ha.tls.enabled`	Negotiate TLS on the Raft gRPC transport (AppendEntries, RequestVote, and the Ratis admin/client calls). When true, the three file settings below must all name a readable PEM file or the node refuses to start. Independent of `arcadedb.ssl.enabled`, which covers the HTTP side channels. See mTLS for the Raft gRPC transport.	false
`arcadedb.ha.tls.certChainFile`	PEM file with this node's Raft certificate followed by any intermediate CAs. The certificate needs a SAN matching the address peers dial in `arcadedb.ha.serverList`, and both the `serverAuth` and `clientAuth` extended key usages. Required when `arcadedb.ha.tls.enabled` is true.	(empty)
`arcadedb.ha.tls.privateKeyFile`	PEM file with the **PKCS#8** private key matching `certChainFile`. Required when `arcadedb.ha.tls.enabled` is true. A `WARNING` is logged if the file is readable beyond its owner.	(empty)
`arcadedb.ha.tls.trustCertCollectionFile`	PEM file with the cluster CA certificate(s) signing every node certificate. Required when `arcadedb.ha.tls.enabled` is true, including with `mutualAuth=false`.	(empty)

Setting	Description	Default Value
`arcadedb.ha.tls.mutualAuth`	Require the dialing peer to present a client certificate signed by the cluster CA, binding peer identity to a certificate rather than a source IP. `false` gives server-only TLS, which encrypts the traffic but leaves the Raft port open to any client trusting the cluster CA. See Server-only TLS.	true
`arcadedb.ha.logVerbose`	Verbose HA logging: 0=off, 1=basic, 2=detailed, 3=trace	0
`arcadedb.ha.k8s`	The server is running inside Kubernetes (enables auto-join)	false
`arcadedb.ha.k8sSuffix`	DNS suffix used to reach the other servers in Kubernetes, e.g. `.arcadedb.default.svc.cluster.local`	(empty)

8.11. Backup of a Database

ArcadeDB allows to execute a non-stop backup of a database while it is used without blocking writes or affecting performance.

A backup, and a `CHECK DATABASE` verification, always reflect every transaction committed before they started.

There are two ways to perform backups:

- **Manual backup** - Execute a one-time backup using SQL or the Studio UI
- **Automatic backup** - Schedule recurring backups with retention policies (see Automatic Backup Scheduler)

For manual backups, you can execute the backup of a database from SQL. Look at Backup Database SQL command for more information.

8.11.1. Consistency of TimeSeries data

A backup captures the database schema at a single point in time. TimeSeries types keep their compacted history in separate `.ts.sealed` files, which are archived alongside it.

The schema and the list of those files are captured together, so an archive can no longer declare a TIMESERIES type whose data it does not contain. Previously a `DROP TYPE` running at the same moment as a backup could produce exactly that, and the backup still reported success.

If one of those files cannot be read while the backup runs, the backup now **fails**

with an error naming the file instead of skipping it and reporting success.

This is deliberate. An archive that silently omits data its own schema declares looks valid until a restore is attempted, which is the worst moment to discover it. A failed backup is visible, and simply running it again produces a consistent archive. The same applies when the database directory cannot be listed at all: that is an error rather than being read as "this database has no TimeSeries data".

Note that this costs nothing in availability. The database is locked only long enough to take the snapshot and list the files, not for the duration of the backup, so schema changes still run normally while a backup is in progress.

(Since v26.11.1) a backup that fails for any reason, an `OutOfMemoryError` halfway through a large archive included, deletes the partial archive before the error reaches the caller, so no truncated file is left for the retention policy to count or for an operator to restore from. Previously only an exception was cleaned up: an error left the partial file on disk, and when the backup ran with page flushing suspended (the fallback used when the point-in-time snapshot is disabled or cannot hold) it could even be completed and reported as a successful backup. If the filesystem refuses to delete the partial archive, the failure reported by the backup says so and names the file, which must then be removed by hand.

8.11.2. Vector indexes

A vector index keeps its proximity graph in a `.vecgraph` file, next to small side files that describe it: a manifest that vouches for the graph's pages (`.vecgraphfp`), an ordinal map (`.vecordmap`) and, for a `PRODUCT`-quantized index, the quantization (`.vecpq`). The graph is rebuilt from the vectors in the index pages whenever it cannot be verified, so a restored database always answers searches; what changes is whether its first search has to build the graph first.

(Since v26.10.2) a full backup archives the graph together with its side files when it can prove that they belong to the same graph as the pages captured by the backup: the files are unchanged across the point at which the backup took its snapshot. The restored database then verifies the graph against its manifest and reuses it. When the proof fails, for example because the graph was being persisted at that moment, or when the backup had to freeze the data files instead of using a point-in-time window (see how-to/operations/monitoring.pdf), the graph is archived empty and without side files. Each such vector index then builds its graph once, on its first search after the restore.

An archive taken by an earlier version carries the graph pages without their side files. After restoring it, a vector index reuses those pages without being able to verify them: the reuse is counted in `unverifiedGraphReuses` in the index's `getStats()` and logged at `WARNING` (`Vector index <name> is reusing a persisted graph that carries no manifest …`). Run `REBUILD INDEX` on each vector index after restoring such an archive. A `PRODUCT`-quantized index rebuilds its graph on its own, since its quantization file is missing.

8.11.3. Configuration

- `-f <backup-file>` (string) filename of, or path to the backup file to create.

- `-d <database-path>` (string) path on local filesystem where to find the ArcadeDB database.
- `-o` (boolean) true to overwrite the backup if already exists. If false and the `backup-path` already exists, an error is thrown. Default is false. *(Since v26.11.1)* a backup with `-o` that fails before it starts writing leaves the existing backup file untouched; once it has started writing, the previous file is already overwritten and the partial one is deleted.

8.11.4. Cloud Backups

In a container setting it may become necessary to send backups to an S3 bucket instead of a mounted volume. Currently, ArcadeDB does not support writing a backup to S3 directly, but there are two ways of achieving this:

1. Using an intermediary container which forwards a volume's contents to an S3 bucket. The project docker-s3-volume makes this easy.
2. Mounting an S3 bucket inside the ArcadeDB container directly with the S3 file system via filesystem in userspace.

8.12. Automatic Backup Scheduler

ArcadeDB provides an automatic backup scheduler that runs as a server plugin. This feature allows you to configure scheduled backups for all databases with flexible scheduling options, tiered retention policies, and high availability cluster support.

The auto-backup scheduler is activated when a `backup.json` configuration file exists in the server's configuration directory: `config/` under the server root by default, or the directory set by `arcadedb.server.configDirectory`.

8.12.1. Features

- **Frequency-based scheduling** - Run backups every N minutes
- **CRON-based scheduling** - Use CRON expressions for precise scheduling (e.g., daily at 2 AM)
- **Time windows** - Restrict backups to specific time periods (e.g., off-peak hours)
- **Tiered retention** - Keep hourly, daily, weekly, monthly, and yearly backups
- **Per-database configuration** - Override defaults for specific databases
- **HA cluster awareness** - Configure which node runs backups in a cluster
- **Follows the database lifecycle** - New databases are scheduled automatically, dropped or closed ones stop
- **Studio integration** - Configure and monitor backups through the web UI

8.12.2. Configuration File

Create a file named `backup.json` in the `config` directory of your ArcadeDB installation. The server will automatically detect this file on startup and enable the auto-backup scheduler.

Example backup.json

```
{
  "version": 1,
  "enabled": true,
  "backupDirectory": "./backups",
  "defaults": {
    "enabled": true,
    "runOnServer": "$leader",
    "schedule": {
      "type": "frequency",
      "frequencyMinutes": 60,
      "timeWindow": {
        "start": "02:00",
        "end": "06:00"
      }
    },
    "retention": {
      "maxFiles": 10,
      "tiered": {
        "hourly": 24,
        "daily": 7,
        "weekly": 4,
        "monthly": 12,
        "yearly": 3
      }
    }
  },
  "databases": {
    "production": {
      "schedule": {
        "type": "cron",
        "expression": "0 0 2 * * ?"
      },
      "retention": {
        "maxFiles": 20
      }
    },
    "analytics": {
      "enabled": false
    }
  }
}
```

8.12.3. Configuration Options

Global Settings

Setting	Default	Description
`version`	`1`	Configuration file version (for future compatibility)
`enabled`	`true`	Enable or disable the auto-backup scheduler globally
`backupDirectory`	`./backups`	Directory where backups are stored (must be a relative path within server root)

The `backupDirectory` must be a relative path and cannot contain `..` for security reasons. Absolute paths are rejected to prevent writing outside the server directory.

Default Settings

The `defaults` section defines settings applied to all databases unless overridden.

Setting	Default	Description
`defaults.enabled`	`true`	Enable backups for databases by default
`defaults.runOnServer`	`$leader`	Which server runs backups (see HA Cluster Support)

Schedule Settings

Setting	Default	Description
`schedule.type`	`frequency`	Schedule type: `frequency` or `cron`
`schedule.frequencyMinutes`	`60`	Minutes between backups (when type is `frequency`)
`schedule.expression`	-	CRON expression (when type is `cron`)
`schedule.timeWindow.start`	-	Time window start (HH:mm format, e.g., "02:00")
`schedule.timeWindow.end`	-	Time window end (HH:mm format, e.g., "06:00")

Retention Settings

Setting	Default	Description
`retention.maxFiles`	`10`	Maximum backup files to keep per database
`retention.tiered.hourly`	`24`	Number of hourly backups to retain
`retention.tiered.daily`	`7`	Number of daily backups to retain
`retention.tiered.weekly`	`4`	Number of weekly backups to retain
`retention.tiered.monthly`	`12`	Number of monthly backups to retain
`retention.tiered.yearly`	`3`	Number of yearly backups to retain

When tiered retention is configured, the scheduler keeps the oldest backup of each of the most recent time buckets of every tier, and deletes the backups no tier keeps (the most recent backup is always kept). For example, with `hourly: 24`, it keeps one backup per hour for the last 24 hours. Weeks are ISO weeks, starting on Monday, so a week that spans a year end is one bucket.

Per-Database Overrides

The `databases` section allows overriding any default setting for specific databases. Only specify the settings you want to override; unspecified settings inherit from defaults.

```
{
  "databases": {
    "critical-db": {
      "schedule": {
        "type": "cron",
        "expression": "0 0 * * * ?"
      },
      "retention": {
        "maxFiles": 48,
        "tiered": {
          "hourly": 48,
          "daily": 14
        }
      }
    },
    "dev-db": {
      "enabled": false
    }
  }
}
```

8.12.4. CRON Expressions

The auto-backup scheduler supports 6-field CRON expressions:

```
┌──────────────────────────── second (0 - 59)
│ ┌────────────────────────── minute (0 - 59)
│ │ ┌──────────────────────── hour (0 - 23)
│ │ │ ┌────────────────────── day of month (1 - 31)
│ │ │ │ ┌──────────────────── month (1 - 12)
│ │ │ │ │ ┌────────────────── day of week (0 - 6, Sunday = 0)
│ │ │ │ │ │
* * * * * *
```

Table 2. Common CRON Examples

Expression	Description
`0 0 2 * * ?`	Every day at 2:00 AM
`0 0 */6 * * ?`	Every 6 hours
`0 30 1 * * 0`	Every Sunday at 1:30 AM
`0 0 3 1 * ?`	First day of every month at 3:00 AM

Expression	Description
`0 0 0 * * 1-5`	Every weekday at midnight

Supported special characters:

- `*` - matches all values
- `?` - no specific value (use for day-of-week or day-of-month)
- `-` - range (e.g., `1-5` for Monday through Friday)
- `/` - step (e.g., `*/15` for every 15 units)

8.12.5. HA Cluster Support

In a high availability cluster, you can control which server performs backups using the `runOnServer` setting:

Value	Description
`$leader`	Only the current leader node runs backups (default, recommended)
`*`	All nodes run backups independently
`node-name`	Only the specified node runs backups (e.g., `"ArcadeDB_Replica1"`)

Using `$leader` is recommended because:

- Ensures exactly one backup is created per schedule
- Automatically fails over if the leader changes
- Avoids duplicate backups across the cluster

Example: Replica-based backup

```
{
  "defaults": {
    "runOnServer": "ArcadeDB_Replica1"
  },
  "databases": {
    "analytics": {
      "runOnServer": "ArcadeDB_Replica2"
    }
  }
}
```

8.12.6. Backup Directory Structure

Backups are organized by database name:

```
backups/
 ├─── database1/
```

```
│   ├── database1-backup-20250115-020000041.zip
│   ├── database1-backup-20250116-020000038.zip
│   └── database1-backup-20250117-020000105.zip
└── database2/
    ├── database2-backup-20250115-020000062.zip
    └── database2-backup-20250116-020000057.zip
```

The backup filename format is: `<database>-backup-<YYYYMMDD>-<HHMMSSmmm>.zip`, where the last three digits are milliseconds.

Archives created by an older version, whose name stops at the second, keep their name and are still listed and rotated normally, so nothing needs to be renamed after an upgrade.

8.12.7. Monitoring

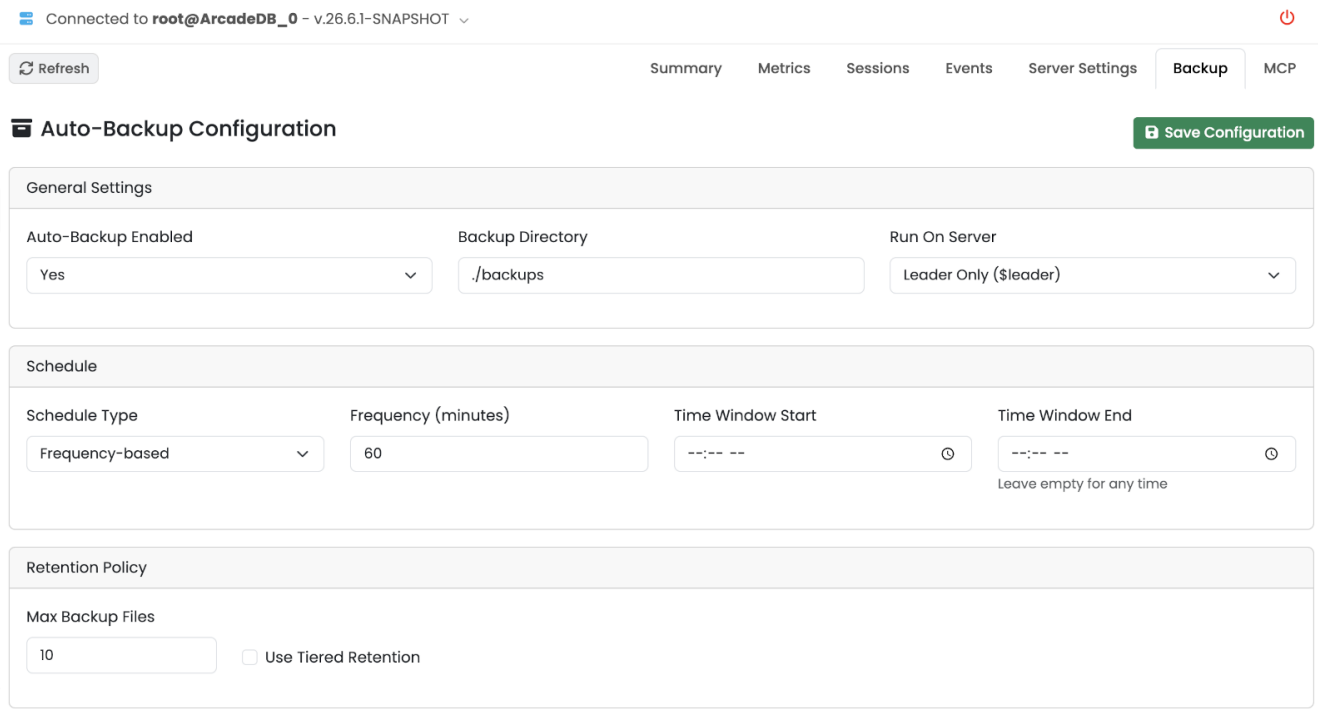
The auto-backup scheduler logs its activity:

```
INFO  [AutoBackupSchedulerPlugin] Auto-backup scheduler started. Backup directory:
./backups
INFO  [AutoBackupSchedulerPlugin] Scheduled automatic backup for database 'production'
INFO  [BackupScheduler] Starting scheduled backup for database 'production'
INFO  [BackupScheduler] Backup completed: production-backup-20250115-020000041.zip
(15.2 MB)
INFO  [BackupRetentionManager] Retained 10 backups, deleted 2 for database
'production'
```

Backup events are also recorded in the Server Event Log, accessible from the Studio Server panel.

8.12.8. Studio Integration

The auto-backup scheduler can be configured and monitored through the ArcadeDB Studio web interface. See Studio Backup Panel for details.



From the Studio, you can:

- View and modify the backup configuration
- Trigger immediate backups for any database
- View the list of available backups per database
- Monitor backup statistics and disk usage

8.12.9. Triggering Manual Backups

You can trigger an immediate backup through the Studio UI or via the HTTP API:

HTTP API

```
curl -X POST "http://localhost:2480/api/v1/server" \
  -u root:password \
  -H "Content-Type: application/json" \
  -d '{"command": "trigger backup myDatabase"}'
```

SQL Command

```
BACKUP DATABASE
```

Where an On-Demand Backup Goes

`trigger backup`, `list backups`, `delete backup`, `restore backup` and the SQL `BACKUP DATABASE` statement all use one definition of the backup directory, so an archive one of them writes the others can see, whether or not the scheduler is running. The directory is resolved in this order, first hit wins:

1. the `backupDirectory` of the running auto-backup plugin;

2. the `backupDirectory` in `config/backup.json`, even when that file sets `"enabled": false`;
3. the server setting `arcadedb.server.backupDirectory`.

It is resolved per command, so a `set backup config` that moves the directory is picked up by the next one.

Archives always live under `<backupDirectory>/<database>/`. `get backup config` reports the resolved directory as `backupDirectory` next to the configuration. A configured directory that fails validation (an absolute path, or one escaping the server root) makes every backup command fail with an error naming it, where `list backups` used to answer an empty list; `get backup config` reports the same failure as `backupDirectoryError`. A `config/backup.json` that cannot be read or parsed is skipped with a warning in the server log, and the chain continues to the server setting.

`list backups` reports each archive's `size` and `lastModified` as read from the file system. Since 26.11.1 an archive whose attributes cannot be read is left out of the list rather than reported with a size and a date of 0: one the retention sweep removed while the list was being built is skipped silently, any other failure (permissions, a broken mount) is logged as a warning naming the file. A database with no backup directory has no backups, while a directory that exists but cannot be reached or listed fails the command with an error naming the database.

The SQL `BACKUP DATABASE` statement is on this chain too. As a consequence `BACKUP DATABASE` fails, rather than quietly writing elsewhere, when the configured directory fails the validation above. Embedded databases and remote connections are unaffected: with no server configuration in play the statement uses `arcadedb.server.backupDirectory`.

Retention runs only while the scheduler is enabled: with auto-backup off, an on-demand archive stays on disk until it is removed with `delete backup`. While it is enabled, both on-demand entry points - `trigger backup` and, since 26.10.2, the SQL `BACKUP DATABASE` statement - prune the database's directory after a successful backup with its effective policy, including for a database that is not named in the configuration (the `defaults` retention applies). Only archives named `<database>-backup-<timestamp>.zip` are subject to retention; one named with `BACKUP DATABASE '<name>'` is never deleted by it.

A backup writes its archive as `<name>.zip.partial` and renames it once it is complete, so a backup killed while writing (`kill -9`, the process running out of memory) leaves the `.partial` file behind. No backup command lists, deletes or restores such a file. *(Since v26.12.1)* every successful on-demand backup, `trigger backup` or SQL `BACKUP DATABASE`, deletes from that database's directory the `<database>-backup-*.zip.partial` files that have not been written for one hour, whether or not auto-backup is enabled. A more recent one may belong to a backup still running and is kept. Previously these files were only removed by the retention of an enabled scheduler, so with auto-backup off (the default) they piled up until removed by hand.

One Backup at a Time

A database is backed up by one backup at a time. If you ask for a backup while another one of the same database is still running - a manual trigger during a scheduled run, or two clicks in a row - the second request is refused rather than run alongside the first: a second copy of the same data adds

nothing, and it would only compete for disk and CPU with the run already in progress.

The HTTP API answers with status `409` and the message `A backup of database '<name>' is already in progress`. Studio shows the same message. A scheduled run that arrives at a busy moment is simply skipped, with a warning in the log, and the next scheduled run covers the database as usual.

8.12.10. Database Lifecycle

Schedules follow their database, so you do not have to edit `backup.json` every time a database comes or goes:

- A database created, restored or imported while the server is running is scheduled straight away, with the same defaults (or per-database override) any other database gets.
- Dropping or closing a database cancels its schedule. The backups already on disk are left alone: a backup taken before the drop is exactly what you would restore from.
- A scheduled backup never opens a database by itself. If the database is not available on this server when its turn comes, that run is skipped.

The last point matters when you close a database deliberately, for maintenance for example: the scheduler will not reopen it behind your back.

8.12.11. Troubleshooting

Backup scheduler not starting

Verify that:

1. The `backup.json` file exists in the `config` directory
2. The JSON syntax is valid
3. The `enabled` field is set to `true`
4. Check the server log for error messages

Backups not being created

Check:

1. The backup directory exists and is writable
2. The `timeWindow` settings (backups only run within the window)
3. In HA mode, verify `runOnServer` matches the current server
4. Sufficient disk space is available

Permission denied errors

The backup directory must be:

- A relative path (e.g., `./backups`, not `/backups`)
- Within the server's root directory

- Writable by the server process

8.13. Restore a Database

ArcadeDB allows to restore a database previously backed up.

8.13.1. Restore via HTTP API (recommended)

The easiest way to restore a database on a running server is via the server command HTTP API. This supports both local files and remote URLs:

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "restore database mydb file:///path/to/mydb-backup.zip"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

Restoring from a remote URL (e.g., a backup hosted on a server or cloud storage):

```
$ curl -X POST http://localhost:2480/api/v1/server \
       -d '{"command": "restore database mydb https://example.com/backups/mydb-
backup.zip"}' \
       -H "Content-Type: application/json" \
       --user root:arcadedb-password
```

The database must not already exist. The server downloads the backup file, extracts it, and opens the database automatically.

For security reasons, restoring from an `http://`/`https://` URL refuses by default to reach a local-file, loopback, link-local, or private-network address, including a redirect that lands on one. This protects the server from being made to fetch from internal-only hosts (e.g. a cloud metadata endpoint) via a crafted or hijacked backup URL. If you need to restore from a backup hosted on such an address (for example, an internal artifact server), set `arcadedb.server.restoreImportAllowLocalUrls=true` on the server.

8.13.2. Restore via command line

If the server is not running, you can restore a database using the command-line tool.

If a server is running, it must be restarted in order to access a database restored via command line.

Example

Example for restoring the database "mydb" from the backup located in `backups/mysb/mydb-backup-20210921-172750767.zip`.

```
$ bin/restore.sh -f backups/mysb/mydb-backup-20210921-172750767.zip -d databases/mydb
```

A restore forces every restored file to disk (fsync) before it reports success, together with the database directory and its parent. A power loss after the restore completes cannot leave the restored database partially written.

(Since v26.11.1) the archive is extracted into a hidden staging directory next to `database-path`, and moved to `database-path` only once the extraction succeeded and the result was verified to be a database. A restore that fails (a truncated or corrupted archive, a wrong encryption key, a full disk) leaves nothing behind, so it can be retried with the same command. Previously the half-extracted directory stayed at `database-path`, and every later restore into it was refused with "already exist" until it was removed by hand. With `-o`, the database being replaced is kept until the restored copy is complete and only then replaced, so a failed restore no longer destroys it. This means the disk must have room for both the existing database and the restored copy while the restore runs, as was already the case for a restore through the server. If the process is killed while the restore runs, what it leaves are hidden directories next to `database-path`, never a half-written `database-path`: `.<name>.restore-<n>` is an unfinished restored copy and can be deleted, and `.<name>.replaced-<n>`, left only by a kill in the instant of the `-o` swap, is the previous database, which can be renamed back to `database-path` if that is missing.

(Since v26.11.1) a restore refuses an archive the backup did not finish writing (one missing the ZIP central directory a completed backup writes last, or whose central directory does not match its content), whatever the source: a local file, an encrypted archive or an `http://`/`https://` URL. Previously the single-threaded path (always used for encrypted and remote archives) stopped at the last complete file and reported success over a partial database. A local archive is checked before anything is extracted.

8.13.3. Configuration

- `-f <backup-file>` (string) path to the backup file to restore.
- `-d <database-path>` (string) path on local filesystem where to create the ArcadeDB database.
- `-o` (boolean) true to overwrite the database if already exists. If false and the `database-path` already exists, an error is thrown. Default is false.

8.14. Monitoring

ArcadeDB provides built-in metrics collection and integration with Prometheus and Grafana for production monitoring.

8.14.1. Enabling Metrics

Metrics are enabled by default. Verify with:

```
SELECT FROM ( SELECT expand(settings) FROM schema:database ) WHERE key =
'arcadedb.serverMetrics'
```

Or set in `arcadedb-server.properties`:

```
arcadedb.serverMetrics=true
arcadedb.serverMetrics.logging=true
```

8.14.2. Prometheus Integration

The Prometheus metrics plugin auto-registers when the `metrics` module is on the classpath (included in the standard distributions) and metrics are enabled (`arcadedb.serverMetrics=true`, the default). It exposes a scrape endpoint at `http://localhost:2480/prometheus`.

Authentication on the scrape endpoint is on by default; set `arcadedb.serverMetrics.prometheus.requireAuthentication=false` to allow unauthenticated scraping.

This setting accepts only `true` or `false`. Anything else (`yes`, `1`, `on`, or a misspelling) is refused with a warning at startup and the endpoint stays authenticated, so a typo cannot expose it by accident. See Settings.

Configure Prometheus to scrape ArcadeDB:

```
# prometheus.yml
scrape_configs:
  - job_name: 'arcadedb'
    scrape_interval: 15s
    metrics_path: /prometheus
    static_configs:
      - targets: ['localhost:2480']
    basic_auth:
      username: root
      password: arcadedb
```

8.14.3. Available Metrics

ArcadeDB exposes metrics for:

- **Database operations** — Query count, command count, read/write throughput
- **Cache** — Page cache hits, misses, evictions
- **Disk** — Read/write operations, compaction activity

- **Transactions** — Active transactions, commits, rollbacks
- **Network** — HTTP request count, response times
- **JVM** — Heap usage, GC pauses, thread count
- **Backup** — Point-in-time window activity (see below)
- **Time series reads** — How much data each query actually had to read (see below)
- **High availability** — Raft layer restarts and the HA worker pools (see below)

Disk space

`diskFreeSpace`, `diskTotalSpace` and `diskFreeSpacePerc` (on `GET /api/v1/server` and on the **Databases Disk** card in Studio) describe the filesystem holding the configured database directory (`arcadedb.server.databaseDirectory`), which on a container is normally a mounted volume rather than the filesystem the server process was started in. The `diskDirectory` field names the measured directory, so it is always clear which filesystem the figures refer to.

Free space is what the server process can actually write, so any quota or reserved space is already subtracted.

Backup point-in-time windows

A full backup, an HA database verify and an HA snapshot ship read a point-in-time image of the database. By default that image is served from a copy-on-write shadow, so writers keep running at full speed while it is read. When the shadow cannot be used the operation still completes, but by freezing the data files instead - which throttles every writer until it finishes.

That fallback is invisible from the outside: the backup succeeds either way. These counters are what make it alertable:

Metric	Meaning
`arcadedb_engine_snapshot_windows_overflowed_total`	The shadow reached `arcadedb.pageSnapshotMaxSize`. Raise it, or give the spill volume more room.
`arcadedb_engine_snapshot_windows_failed_total`	A page pre-image could not be read or written. Look at the disk.
`arcadedb_engine_snapshot_barrier_failed_total`	The window could not be opened at all. Split by cause below.
`arcadedb_engine_snapshot_barrier_failed_suspend_total`	Transient and expected under sustained load.
`arcadedb_engine_snapshot_barrier_failed_flush_total`	A pending write never landed. Points at the disk.

Any of these rising means backups have quietly gone back to throttling writers. See `arcadedb.pageSnapshotMaxSize`, `arcadedb.pageSnapshotMaxRAM` and `arcadedb.pageSnapshotSpillPath` in Settings.

Time series reads

A time series query does not read everything in its range. Older data is stored in compacted blocks, each carrying a summary of what it holds, so a query can skip a block entirely or answer from its summary without decompressing it. These counters say how often that is working for your data, and are tagged by `db`, `type` and `surface` (the endpoint that made the read).

Metric	Meaning
`arcadedb_timeseries_read_blocks_total`	Compacted blocks, split by `outcome`: `skipped` (not read at all), `fast-path` (answered from the block summary) and `slow-path` (decompressed in full).
`arcadedb_timeseries_read_pages_total`	Pages of recent, not-yet-compacted data, split by `outcome`: `scanned` or `skipped`.
`arcadedb_timeseries_read_rows_total`	Rows actually materialised, i.e. the ones that survived every filter.
`arcadedb_timeseries_read_overflow_buckets_total`	Aggregation buckets that fell outside the pre-sized window. Results are correct either way, but a rising value means the range estimate is coming up short.
`arcadedb_timeseries_read_phase_seconds`	Time spent per `phase`: `io`, `decompress-timestamps`, `decompress-values`, `accumulate`.

The ratio worth watching is `slow-path` against `skipped` and `fast-path`. A high `slow-path` share means queries are decompressing blocks they could have skipped - usually because the query filters on something the block summaries cannot narrow, or because the blocks predate the summary they would need.

These counters are collected only while metrics are enabled; with metrics off the read path is unchanged and nothing is measured.

High availability

On a node running HA, these metrics are published in addition to the schema full-load metrics described under how-to/operations/ha.pdf. *(Since v26.10.2)*

Metric	Meaning
`arcadedb_ha_in_place_restarts_recovered_total`	Times this node restarted its Raft layer in place and kept its Raft log, since the process started.
`arcadedb_ha_in_place_restarts_reformatted_total`	Times this node restarted its Raft layer in place and reformatted its Raft storage, rejoining through a snapshot.

Metric	Meaning
`arcadedb_ha_security_documents_not_durable`	Replicated security documents (users, groups, API tokens) in force on this node but not yet written to its configuration directory, because the write failed. While it is above 0 the Raft log is not compacted past the change, so a restart replays it: free or fix the configuration volume. It returns to 0 on its own once a retried write succeeds. *(Since v26.11.1)*

The same two counts are in `localInPlaceRestarts` on `GET /api/v1/cluster`. A rising `reformatted` count means a follower keeps losing its log; see how-to/operations/ha.pdf.

The worker pools of the HA layer are published as `arcadedb.executor.*` rows, like the server's other pools, with the `pool` tag set to `snapshot_install`, `channel_recovery`, `stalled_resync`, `database_deleter` or `sm_lifecycle` (next to `security_seed` and `security_catch_up`). They are also shown in Studio's Executor Pools card. Two gauges say what a full pool did with the work it could not queue:

Metric	Meaning
`arcadedb_executor_tasks_rejected`	Tasks refused and dropped: `snapshot_install` (Ratis retries the install) and `channel_recovery`. Lost work, shown highlighted in the Studio **Rejected** column when it is not zero.
`arcadedb_executor_tasks_coalesced`	Tasks skipped because the same work was already queued or running: `channel_recovery`, `security_seed`, `security_catch_up`.

`stalled_resync` and `database_deleter` run a task on the caller's thread when they are full, which `arcadedb_executor_tasks_caller_run_fallbacks` counts; `sm_lifecycle` has an unbounded queue, so its queue depth is the signal. The pools owned by the Raft state machine (`snapshot_install`, `sm_lifecycle`, `database_deleter`, `security_seed`, `security_catch_up`) get new counters when the Raft layer is restarted in place, which a Prometheus `rate()` reads as a counter reset.

`arcadedb_engine_bucket_recount_publishes_refused_total` *(Since v26.10.2)* counts the bucket `count()` recomputes a node could not cache because a replicated apply wrote to the bucket during the scan (`recountPublishesRefused` in the Profiler JSON). A few increments while a follower catches up are normal; a rate that never returns to zero means every `count(*)` on that follower is a full scan. See Metrics depth.

8.14.4. Periodic Metric Dumps

For environments without Prometheus, enable periodic metric logging:

```
arcadedb.dumpMetricsEvery=60
```

This dumps metrics to the server log every 60 seconds. Since v26.11.1 the timer thread behind it (`ArcadeDB-DumpMetrics`) exists only while a positive period is set: the default `0` starts no thread, a new

value replaces the schedule, and setting `0` at runtime stops the dumps and ends the thread. Before, the engine started an unnamed `Timer-N` thread on its first use whatever the setting said, which kept an embedded native image from tearing its isolate down.

8.14.5. Grafana Dashboard Setup

1. Install Grafana and add a Prometheus datasource pointing to your Prometheus instance
2. Import or create dashboards using ArcadeDB metrics
3. For time series data visualization, use ArcadeDB's native Grafana integration endpoints with the Infinity datasource plugin

8.14.6. Health Checks

Use the server info endpoint for health monitoring:

```
curl http://localhost:2480/api/v1/server
```

For time series health:

```
curl http://localhost:2480/api/v1/ts/mydb/grafana/health
```

8.14.7. Server health warnings

The server runs a health monitor that samples, every ten seconds, three things a running server can degrade on:

- free disk space on the filesystem holding the databases — the same filesystem the `diskFreeSpace` figures above describe, and the warning that precedes a database that can no longer write;
- available heap;
- JVM pause times.

When one of them degrades, a `WARNING` event is written to the server event log, visible in Studio and through `GET /api/v1/server/events`. Each kind of warning is rate limited so a server that stays degraded reports it periodically rather than on every sample: once a day for low disk, once every 30 minutes for heap and JVM pauses.

Set `arcadedb.server.healthCheck.enabled=false` to turn the monitor off.

The monitor is enabled by default. On an in-place upgrade a server already close to those thresholds starts producing event-log warnings it did not produce before.

8.14.8. Further Reading

- Cloud Observability — RED metrics, OTLP export, tracing, structured logging, and health probes
- Server Settings — All configuration parameters

- Time Series — Grafana DataFrame endpoints for time series dashboards
- Realtime Analytics Use Case — Complete monitoring example with Grafana

8.15. Cloud Observability

ArcadeDB exposes a vendor-neutral observability stack for production (Kubernetes) deployments: always-on RED metrics, OpenTelemetry distributed tracing, structured JSON logging with correlation IDs, and standard health probes. Everything described here is **opt-in and behavior-preserving** — with no new configuration, an upgraded server behaves exactly as before. The existing `/prometheus` scrape endpoint, `/api/v1/server` JSON, and `/api/v1/ready` semantics are never changed, only added to.

The design follows an *instrument-once* principle: each hot path (HTTP request, query/command, transaction) is wrapped in a single Micrometer `Observation`. When only metrics are enabled, an `Observation` is a timer; when the optional tracing plugin is loaded, the **same** instrumentation also emits a span. A correlation ID then ties the resulting metric, span, and log line together.

8.15.1. Metrics depth

ArcadeDB builds on the Micrometer framework already in the server. Metrics are enabled by default (`arcadedb.serverMetrics=true`) and exposed on the Prometheus scrape endpoint `/prometheus` (see Monitoring for scrape and Grafana setup). The series below are **additive** — existing metrics and the `/prometheus` output shape are unchanged.

RED latency timers

Two always-on timers record request/query latency with a percentile histogram (server-side p50/p95/p99). Query text is never used as a tag, keeping cardinality bounded.

Timer	Tags	Prometheus series
`arcadedb.http.requests`	`method`, `path` (route template, e.g. `/query/{database}` — never the raw URI), `status`, `db` (the database name, or `unknown` when the request names a database that does not exist)	`arcadedb_http_requests_seconds_*`
`arcadedb.query.duration`	`protocol` (`http`, `ws`, `bolt`, `postgres`, `mongo`, `grpc`, `redis`, `gremlin`, `mcp`, `internal`), `db`, `language` (`sql`, `opencypher`, `gremlin`, `graphql`, `mongo`), `type` (`query` or `command`)	`arcadedb_query_duration_seconds_*`

Queries that an MCP tool runs are reported as `mcp` on both MCP transports. Before 26.11.1 they were reported as `http` (the HTTP transport) or `internal` (stdio), so an agent's traffic could not be told from a REST client's, and a `protocol!="internal"` filter dropped the stdio traffic. The same tag is what tells a client request from the engine's own work under HA, where a database whose directory is being replaced from the leader's snapshot refuses client requests.

The `protocol` tag defaults to `internal` for embedded/in-process callers; dashboards typically filter `protocol!="internal"` to isolate client traffic. Redis commands and unfiltered Mongo scans do not pass through the query engine and are therefore not recorded by `arcadedb.query.duration`.

The `db` tag of `arcadedb.http.requests` carries a database name only when that database exists on the server, and the constant `unknown` otherwise. The `{database}` path parameter matches any URL segment, and the timer is also recorded for requests answered `401`, so echoing it verbatim let any caller register one permanent series per invented name. Requests against a non-existent database are therefore grouped under `db="unknown"` rather than appearing one series per name.

Engine gauges

`EngineMetricsBinder` bridges the internal engine `Profiler` into Micrometer gauges (engine-wide aggregates, refreshed from a one-second-memoized snapshot):

Gauge	Meaning
`arcadedb.engine.page.cache.hits` / `.page.cache.misses`	Page cache hit/miss counters
`arcadedb.engine.pages.read` / `.pages.written`	Pages read from / written to disk
`arcadedb.engine.wal.bytes.written` / `.wal.files`	Write-ahead-log bytes and file count
`arcadedb.engine.mvcc.conflicts`	MVCC concurrent-modification conflicts
`arcadedb.engine.files.open`	Open file handles
`arcadedb.engine.tx.write` / `.tx.read` / `.tx.rollbacks`	Write/read transactions and rollbacks
`arcadedb.engine.queries` / `.commands`	Query and command counts
`arcadedb.engine.databases`	Number of open databases
`arcadedb.engine.async.boundary.commits`	Asynchronous batch transactions cut short by a durability-flag change
`arcadedb.engine.async.workers.retiring`	Asynchronous workers a lowered parallel level retired and that are still draining
`arcadedb.engine.graph.light.edge.census.fallbacks`	Lightweight-edge lists the census could not count, whose edges each walk the list alone
`arcadedb.engine.bucket.recount.publishes.refused`	Bucket `count()` recomputes not cached because a replicated apply ran under the scan *(Since v26.10.2)*
`arcadedb.engine.wal.group.commits` / `.wal.group.commit.syncs`	Commits that shared a WAL flush, and the shared flushes (group commit) *(Since v26.11.1)*
`arcadedb.engine.commit.lock.holds` / `.commit.lock.hold.seconds`	Releases of the locks a commit takes on the files it writes, and the total time they were held *(Since v26.11.1)*

Gauge	Meaning
`arcadedb.engine.commit.lock.handoffs` / `.commit.lock.handoff.seconds`	Those locks passed to a waiting commit, and the total time from the release to that commit running *(Since v26.11.1)*

The `commit.lock.*` and `wal.group.*` counters are read as averages. `rate(…hold.seconds) / rate(…holds)` is how long a commit keeps the files it writes locked, and with many clients writing into the same type it is what limits the commits per second: about 30 microseconds is normal, milliseconds mean the commits are waiting for the disk while they hold the locks (`arcadedb.txWalFlush` set to 1 or 2 with `arcadedb.txWalGroupCommit=false`). `rate(wal.group.commits) / rate(wal.group.commit.syncs)` is how many commits each disk flush served. The same counters are listed in Studio's server page.

The two `async.*` metrics describe the asynchronous executor and are worth reading together with the throughput of your ingestion.

`async.boundary.commits` counts how often a change to `setTransactionUseWAL()` / `setTransactionSync()` forced a worker to close its open batch transaction early. Occasional increments are normal - a bulk load flips those flags at its start and end. A rate that stays high while asynchronous throughput drops means two callers are flipping them against each other on the same database, so batching is degrading towards one commit per task.

`async.workers.retiring` is a gauge, not a counter: it goes back to zero as the retired workers finish. Zero is the normal reading, and a brief spike during a `setParallelLevel()` call is expected. A reading that stays above zero means a retired worker is stuck inside user code and will never finish draining - a resize deliberately never interrupts one - so the thread stays alive until the database is closed.

`graph.light.edge.census.fallbacks` is a counter of how often counting the lightweight edges of one vertex's edge list failed, so that every lightweight edge read off that list resolves its occurrence by walking the list itself - the quadratic cost the census exists to avoid, which on a super-node is the difference between milliseconds and minutes. Zero is the normal reading. The first failure of each list of a database is logged at `WARNING` (at most 64 distinct lists per database; the rest at `FINE`), so the log shows which list could not be read and this counter shows how often it keeps happening. *(Since v26.11.1: one `WARNING` per JVM and no counter.)*

`bucket.recount.publishes.refused` (Prometheus `arcadedb_engine_bucket_recount_publishes_refused_total`; in the Profiler JSON, `recountPublishesRefused: {count: N}`) counts the times a bucket recomputed its record count but could not cache the result, because a replicated apply wrote to the bucket while it was being scanned. One or two increments are the normal cost of an HA follower catching up. A rate that never returns to zero means a follower on which every `count(*)` is a full scan. *(Since v26.10.2)*

Database open and close counters

(Since v26.11.1) Two counters follow the databases the server closes when they fall idle or exceed the cap on open databases (see Closing idle databases):

Counter	Tags	Meaning
`arcadedb.server.databases.idle_closed`	`reason` (`idle` or `cap`)	Databases closed because nobody used them for `arcadedb.server.databaseIdleTimeout` (`idle`) or to bring the open databases back under `arcadedb.server.maxOpenDatabases` (`cap`)
`arcadedb.server.databases.reopened`		Databases a request opened again after such a close
`arcadedb.server.databases.idle_close_failed`		Idle or cap closes that failed: the database is deregistered anyway and the next request opens it from its files

The gRPC server reports its own calls through `grpc.requests.total`, `grpc.request.duration` and `grpc.errors.total` (see gRPC metrics).

Optional OTLP metrics export

To push metrics to an OpenTelemetry Collector (in addition to the unchanged `/prometheus` scrape), enable the OTLP registry:

```
arcadedb.serverMetrics.otlp.enabled=true
arcadedb.serverMetrics.otlp.endpoint=http://otel-collector:4318/v1/metrics
# optional: push every 15 seconds instead of every minute (minimum 1000 ms)
arcadedb.serverMetrics.otlp.step=15000
```

The metrics exporter speaks **OTLP/HTTP** (protobuf), not gRPC, so the endpoint is the collector's HTTP receiver: port `4318`, path `/v1/metrics` (this is also the default). Port `4317` is the gRPC receiver and does not work for metrics; the server logs a warning if it is configured. A URL without a path gets `/v1/metrics` appended. Tracing is different: it uses OTLP/gRPC on port `4317` (see below).

The metrics and tracing plugins report the OpenTelemetry `service.name` resource attribute. The standard `OTEL_SERVICE_NAME` environment variable is honored and takes precedence over `OTEL_RESOURCE_ATTRIBUTES` and over the `arcadedb.serverMetrics.serviceName` setting (default `arcadedb`), so in a container set it with `-e OTEL_SERVICE_NAME=my-arcadedb`.

This registers an OTLP meter registry alongside Prometheus; the scrape path is unaffected whether or not OTLP is enabled. The OTLP exporter lives in the optional `metrics` module bundled with the standard distributions.

8.15.2. Distributed tracing

Distributed tracing is an **optional plugin** (the `tracing` module, shipped only in the `full` distribution). When absent or disabled, the in-core `Observation` calls are no-ops with zero span overhead. When enabled, the same instrumentation points used by the metrics timers also emit OpenTelemetry spans.

Enable tracing and point it at your OTLP collector:

```
arcadedb.serverMetrics.tracing.enabled=true
arcadedb.serverMetrics.tracing.endpoint=http://otel-collector:4317
arcadedb.serverMetrics.tracing.samplingRate=0.1
```

- `tracing.endpoint` is an **OTLP/gRPC** endpoint (default port `4317`).
- `tracing.samplingRate` is a parent-based ratio in `[0.0, 1.0]`: `1.0` samples every trace, `0.0` (default) disables sampling.
- If the endpoint is unreachable, tracing self-disables and logs an error — server startup is never blocked.

Two span types are produced and nest correctly (query spans are children of the HTTP span within one trace):

Span	Attributes
`arcadedb.http.server.requests`	`method`, `path` (route template), `db`, `status`
`arcadedb.query`	`protocol`, `db`, `language`, `type`, and `db.statement` (the full query/command text)

A failure marks its span with the `ERROR` status and records the exception as a span event (`exception.type`, `exception.message`, `exception.stacktrace`), which is what tracing backends filter and colour error traces by *(since v26.12.1; before, a failed request or query was exported like a successful one)*:

- an `arcadedb.query` span fails when the query or command throws while it is executed, a statement that does not parse included;
- an `arcadedb.http.server.requests` span fails when the request is answered with a `5xx` status. A `4xx` answer is the client's mistake and does not mark the HTTP span failed, as the OpenTelemetry HTTP server conventions prescribe, although the query span under it carries the failure.

A query span ends when the query or command returns its result set: an error raised later, while the caller reads rows that are produced lazily, is not recorded on it.

The `db.statement` span attribute contains the full query/command text, which may include sensitive data. Secure your OTLP collector endpoint and apply collector-side redaction if required.

ArcadeDB continues an upstream trace by reading the inbound W3C `traceparent` HTTP header. Resource attributes (`service.name`, `service.instance.id`), `OTEL_RESOURCE_ATTRIBUTES`, and trace-context propagation over the replication (Raft) and binary wire protocols are not yet implemented; each non-HTTP request starts a fresh root span.

8.15.3. Structured logging and correlation IDs

By default the server logs human-readable text, byte-identical to previous releases. Set `arcadedb.server.logFormat=json` to emit one JSON object per line instead, suitable for log shippers (Loki, ELK):

```
arcadedb.server.logFormat=json
```

See Logging for the configuration underneath this setting: log levels, rotation, log directory, and routing the engine through SLF4J when embedding.

Each JSON line contains: `timestamp` (local time, `yyyy-MM-dd'T'HH:mm:ss.SSS`), `level`, `logger`, `thread`, `message`, and — when present — `requestId`, `db`, `traceId`, `spanId`, and `exception` (full stack trace). Example:

```
{"timestamp":"2026-06-16T10:15:30.123","level":"INFO","logger"
:"com.arcadedb.server.ArcadeDBServer","thread":"http-1","message":"Query executed"
,"requestId":"req-abc","db":"mydb","traceId":"4bf92f3577b34da6a3ce929d0e0e4736"
,"spanId":"0123456789abcdef"}
```

Correlation IDs

Every HTTP request is tagged with a `requestId`: ArcadeDB reuses the inbound `X-Request-Id` header when present (sanitized, capped at 128 characters) or generates a UUID, and echoes it on the response. The `db` field is taken from the request's database path parameter. When the tracing plugin is active, `traceId`/`spanId` are added too — but `requestId` works for correlation even with tracing disabled.

For operators who keep text logs, `arcadedb.server.logIncludeTrace=true` appends ` [traceId=...]` to each line while a trace is active (default off, preserving current output):

```
arcadedb.server.logIncludeTrace=true
```

8.15.4. Health probes

ArcadeDB exposes two complementary, unauthenticated endpoints so orchestrators can tell a **live** process from a **ready** one. See the HTTP API reference for response details.

Endpoint	Probe	Behavior
`GET, HEAD /api/v1/health`	liveness	Returns `204` whenever the HTTP layer is up. Performs no database I/O and never returns a busy/`503` status, so a node that is merely warming up is not killed. On a High Availability cluster it returns `503` if the node has given up recovering from a crash loop, so the orchestrator restarts a node that otherwise cannot recover on its own.

Endpoint	Probe	Behavior
`GET, HEAD /api/v1/ready`	readiness	Returns `204` once the server is `ONLINE`; `503` (`Server not started yet`) before that. On a High Availability cluster it also returns `503` while this node's replication log writer has failed — a node in that state cannot catch up, so everything it serves is frozen at the moment the writer failed. It rejoins on its own once the writer recovers. This check is not behind `server.readinessRequiresHA`, see below. Otherwise unchanged by default.

Both endpoints also accept `HEAD`, so healthcheck tools that probe with a HEAD request (e.g. `wget --spider`, common in Docker `HEALTHCHECK` instructions) work without a body being returned.

Use `/api/v1/health` for the liveness probe and `/api/v1/ready` for the readiness probe:

```
livenessProbe:
  httpGet:
    path: /api/v1/health
    port: 2480
  initialDelaySeconds: 10
  periodSeconds: 10
readinessProbe:
  httpGet:
    path: /api/v1/ready
    port: 2480
  initialDelaySeconds: 20
  periodSeconds: 10
```

HA-aware readiness

On a High Availability cluster you can make readiness reflect Raft membership so traffic is withheld until a node has joined the group:

```
arcadedb.server.readinessRequiresHA=true
```

When this flag is `true` and HA is active, `/api/v1/ready` returns `503` (`Node has not yet joined the Raft group`) until a leader has been elected. Once a leader is known, a second gate checks that the node is a member of the current Raft configuration and, for a follower, that it has caught up to within `arcadedb.server.readinessHAMaxLag` log entries (default `100`) of the commit index; until then it keeps returning `503` (`Node is not yet in the Raft configuration or has not caught up`). Only then does it return `204`. This prevents a (re)joined follower with a wiped or lagging log from being marked ready mid-catch-up, which would otherwise let a rolling restart drop the write quorum. The default (`false` for `readinessRequiresHA`) preserves current readiness behavior.

With `readinessRequiresHA` on, a third reason can hold a node at `503`: its users, groups and API tokens may not be the cluster's. These live in `server-users.jsonl`, `server-groups.json` and `server-api-tokens.json`, outside the databases, so no snapshot carries them (see Cluster security documents).

Two kinds of node are held:

- a node added to the cluster while running (`POST /api/v1/cluster/peer`, `connect cluster` or a Kubernetes auto-join) that has not yet installed the documents the leader seeds to it;
- a node caught up by a snapshot install from the leader, until the leader confirms its copies or sends new ones.

Until then the node enforces its own copies, so a user dropped, a group narrowed or a token revoked while it was away would still work there. `/api/v1/ready` answers `503` with `Cluster security documents have not reached this node yet: <documents>`. A statically configured member that simply restarts is not held, and neither is a node while it leads, because nobody else can confirm its copies.

The hold is bounded by `arcadedb.ha.securityConvergenceReadinessTimeout` (default `30000` ms; `0` turns it off). When the window expires the node reports ready anyway, so a scale-up or rolling restart cannot stall behind a seed that never arrives, and logs a single SEVERE line naming the documents:

```
Reporting READY after waiting 30000ms for the cluster's security documents, which
never reached this node: server-groups.json. ...
```

When you see it, re-run `connect cluster` or re-POST the node to `/api/v1/cluster/peer` to send the seed again, or reissue the security change. The gRPC `Ready` RPC uses the same check.

The replication-log-writer check described in the table above is deliberately **not** behind this flag. This flag gates a node that is merely behind and which you may reasonably choose to serve from; a node whose log writer has failed cannot catch up at all, so it is withheld from traffic whether or not the flag is set. See Kubernetes deployment for the full StatefulSet example.

8.15.5. Further reading

- Monitoring — Prometheus and Grafana setup
- Kubernetes deployment — Helm chart and probe configuration
- Settings — all configuration keys

8.16. Logging

ArcadeDB writes its own logs through a small internal facade, `com.arcadedb.log.Logger`, which has two implementations:

- **`DefaultLogger`** (the default) writes to `java.util.logging` (JUL), configured by `config/arcadedb-log.properties`. This is what the server distribution uses, and its output is the familiar colored console plus a rotating file under `log/`.
- **`Slf4jLogger`** (opt-in) routes every line through the SLF4J facade instead, so an application that embeds ArcadeDB receives the engine's logs in whatever backend it already uses (Logback, Log4j2, reload4j, or JUL again via `slf4j-jdk14`).

Selecting an implementation is a startup decision, made with a system property and no code

change:

```
-Darcadedb.log.impl=slf4j     # default, or omitted: java.util.logging
```

An unrecognized value falls back to the default logger and reports the fact on `stderr`, so a typo does not silently look like a working configuration.

8.16.1. The default logger (java.util.logging)

The server reads its JUL configuration from the first of these that exists:

1. The file named by the `java.util.logging.config.file` system property. The `server.sh` and `server.bat` scripts set this to `$ARCADEDB_HOME/config/arcadedb-log.properties`, anchored to the installation directory rather than the working directory.
2. An `arcadedb-log.properties` resource on the classpath, which is how embedded applications ship their own configuration.
3. `config/arcadedb-log.properties`, relative to the working directory.

If none is found, JUL defaults apply and a warning goes to `stderr`.

The shipped file configures a console handler and a rotating file handler:

```
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler

.level = INFO
com.arcadedb.level = INFO

java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = com.arcadedb.utility.AnsiLogFormatter

java.util.logging.FileHandler.level = INFO
java.util.logging.FileHandler.pattern = ${arcadedb.server.logsDirectory}/arcadedb.log
java.util.logging.FileHandler.formatter = com.arcadedb.log.LogFormatter
java.util.logging.FileHandler.limit = 100000000
java.util.logging.FileHandler.count = 10
```

`limit` and `count` give ten rotating files of 100 MB each, so the log directory is bounded at roughly 1 GB.

8.16.2. Per-package verbosity

Logger names follow the emitting class, so any package prefix can be given its own level. To trace one subsystem without drowning in everything else:

```
com.arcadedb.level = INFO
com.arcadedb.query.sql.level = FINE
```

```
com.arcadedb.server.ha.level = FINE
```

The same mechanism silences noisy third-party libraries. The shipped configuration uses it to suppress the Apache Ratis per-retry replication flood, which repeats while a follower is briefly unreachable:

```
org.apache.ratis.grpc.server.GrpcLogAppender.level = SEVERE
```

Not every suppression is a line in this file. A properties entry sets one threshold for a whole logger, which is fine when everything that logger says at WARNING is noise. When a logger mixes expected noise with real failures, ArcadeDB installs a `java.util.logging.Filter` in code that drops only the one message. The server does this for the Ratis warning

```
Snapshot file SingleFileSnapshotInfo(t:N, i:M):[.../sm/snapshot.N_M] has missing MD5
file.
```

through `com.arcadedb.server.ha.raft.ratis.RatisSnapshotDigestWarningFilter`. The same logger, `org.apache.ratis.statemachine.impl.SimpleStateMachineStorage`, also reports `Failed to updateLatestSnapshot from …` when the snapshot directory cannot be read, a genuine I/O failure. Raising that logger to `SEVERE` would hide both. The filter drops the record only at WARNING or below, so if a future Ratis release logged the same text at SEVERE it would still show up. You will not find this suppression in `arcadedb-log.properties`, and there is no setting to bring the warning back; see the HA troubleshooting entry for why it is harmless.

Handlers filter independently of loggers, so raising a logger to `FINE` also requires lowering the handler: a `ConsoleHandler.level` of `INFO` discards `FINE` records no matter what the logger allows.

gRPC wire trace

Since v26.10.2 the gRPC server and client log every request and value conversion at `FINE`. In earlier versions this trace could not be enabled. Enable it like any other package:

```
com.arcadedb.server.grpc.level = FINE
com.arcadedb.remote.grpc.level = FINE
```

The trace describes values by type, length and size only, so record content does not reach the log. To include the values as well, start the JVM with `-Darcadedb.grpc.log.showString=true`, and only where logging that data is acceptable.

Since v26.10.2 `LogManager.isDebugEnabled()` and `setDebugEnabled()` no longer exist. Code that used them as a guard should call `LogManager.instance().isLoggable(requester, Level.FINE)`, which follows the logging configuration above.

8.16.3. Log directory

The file handler pattern supports `${…}` placeholders, resolved before JUL opens the file. The default pattern uses `${arcadedb.server.logsDirectory}`, which resolves, in order, from:

1. the `arcadedb.server.logsDirectory` system property,
2. the environment variable of the same name,
3. the `server.logsDirectory` setting,
4. the default `./log`.

The `server.sh` and `server.bat` scripts forward the `ARCADEDB_LOG_DIR` environment variable to that system property, which is the usual way to relocate logs in a container:

```
ARCADEDB_LOG_DIR=/var/log/arcadedb ./bin/server.sh
```

This matters on a read-only root filesystem, such as a Kubernetes pod with `readOnlyRootFilesystem: true`, where `./log` is not writable. Point `ARCADEDB_LOG_DIR` at a writable volume mount. The directory is created at startup if it does not exist.

An unresolvable placeholder falls back to `./log` rather than failing startup, so a typo in the pattern shows up as logs in an unexpected place rather than as a dead server.

8.16.4. Console format

The console formatter is chosen by the `server.logFormat` setting: `text` (the default, human-readable and colored) or `json` (one JSON object per line, for log shippers such as Loki or ELK). The JSON form carries the correlation fields `requestId`, `db`, `traceId`, and `spanId` when they are present. See Structured logging and correlation IDs for the field list and examples, and `server.logIncludeTrace` for appending the trace ID to text lines.

To leave the console handler entirely alone, so that only `arcadedb-log.properties` decides the formatter, disable the automatic install:

```
-Darcadedb.installCustomFormatter=false
```

8.16.5. Routing through SLF4J when embedding

An application that embeds ArcadeDB usually already has a logging backend and wants the engine's output in it, in the same format, filtered by the same rules. Setting `arcadedb.log.impl=slf4j` makes ArcadeDB log through `slf4j-api` and nothing else: the engine pins no backend, so the host application keeps whichever one it uses, and needs no dependency exclusions.

Add exactly one SLF4J binding. With Logback, which Spring Boot already brings in:

```
<dependency>
  <groupId>ch.qos.logback</groupId>
```

```
  <artifactId>logback-classic</artifactId>
</dependency>
```

With Log4j2:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j2-impl</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
</dependency>
```

Then select the logger at startup:

```
java -Darcadedb.log.impl=slf4j -jar my-application.jar
```

or programmatically, before the first database is opened:

```
LogManager.instance().setLogger(new Slf4jLogger());
```

Levels map onto SLF4J's five by numeric weight, so custom JUL levels also resolve: `SEVERE` to `ERROR`, `WARNING` to `WARN`, `INFO` to `INFO`, `FINE` to `DEBUG`, anything lower to `TRACE`. Logger names are unchanged (`com.arcadedb.*`), so the per-package filtering above translates directly:

```
<logger name="com.arcadedb" level="INFO"/>
<logger name="com.arcadedb.query.sql" level="DEBUG"/>
```

Per-request correlation is published to the SLF4J MDC, namespaced so it cannot clobber the host application's own keys: `arcadedb.requestId`, `arcadedb.database`, `arcadedb.traceId`, and `arcadedb.spanId`. Reference them from a pattern with `%X{arcadedb.requestId}`, or rely on a JSON encoder, which includes the MDC automatically. Any value already present under one of those keys is restored after the call.

Ready-to-use `logback.xml` and `log4j2.xml` files, reproducing both native formats, are in `examples/logging` in the source repository.

The SLF4J logger leaves all configuration to the backend and therefore never reads `config/arcadedb-log.properties`. On a packaged server, which bundles the `slf4j-jdk14` binding, setting `arcadedb.log.impl=slf4j` still logs through JUL, but the file handler and formatters declared in that file are not installed, so the rotating `log/arcadedb.log.*` files stop being written. Either keep the default logger on the standalone server, or point JUL at the file yourself with

`-Djava.util.logging.config.file=config/arcadedb-log.properties`.

8.16.6. Custom logger implementations

Both implementations satisfy `com.arcadedb.log.Logger`, a two-method interface. An application that needs to capture the engine's output directly, rather than route it to a backend, can install its own:

```
LogManager.instance().setLogger(new MyLogger());
```

Message parameters use printf-style formatting (`%s`), not SLF4J's `{}` placeholders, because that is how ArcadeDB messages are written. A custom implementation should check whether the level is enabled before formatting, since the engine passes format arguments unformatted precisely so that a disabled level costs nothing.

Since v26.12.1 `LogManager` asks the logger's `isLoggable(requester, level)` before every `log` call and hands it only the messages it answers `true` for. The default implementation answers `true`, so a custom logger that does not override it keeps receiving every message. One that overrides it to filter levels no longer sees the messages it refuses, and should not rely on its `log` methods to observe them.

8.16.7. Further Reading

- Cloud Observability — RED metrics, OTLP export, tracing, and health probes
- Structured logging and correlation IDs — the JSON console format and correlation fields
- Server Settings — `server.logFormat`, `server.logIncludeTrace`, `server.logsDirectory`
- Kubernetes deployment — read-only root filesystems and log volume mounts

8.17. Upgrade ArcadeDB

ArcadeDB automatically upgrades a database to a newer on-disk format when a newer version of ArcadeDB opens it. The migration is transparent and happens lazily on first open, so the only operator step is to point the new binaries at your existing data.

8.17.1. Breaking changes

This section lists behavior changes that may require action when upgrading.

26.12.1 - The `text.*` functions answer what APOC answers

Fifteen `text.*` functions, documented as APOC compatible, answered differently from APOC 5.26 on ordinary input and now answer the same (issue #12104; see Text Functions). The ones a query can notice:

- `text.replace()` and `text.split()` read the second argument as a Java regular expression, where it was literal text. `text.split()` drops the empty pieces at the end (`'a,'` is `['a']`) unless the new third argument `limit` is negative, and `text.split('a.b.c', '.')` is `[]`. The openCypher `replace()`

and `split()` and the SQL `.replace()` and `.split()` are unchanged and literal.

- `text.charAt()` answers the code point (`'hello', 1` is `101`), not the one-character string, and `text.code()` takes a code point and answers the string (`65` is `'A'`), where it answered the code of the first character of its argument.
- `text.jaroWinklerDistance()` answers the distance, 1 minus the similarity: 0.0 for equal strings.
- `text.format()` takes its values as one list, `text.format('%s %d', ['a', 1])`, and an optional language for the separators of numbers.
- `text.snakeCase()` joins the words with `-` (`'fooBarBaz'` is `foo-bar-baz`), `text.slug()` keeps the case and the non-ASCII letters (`'Hello World'` is `Hello-World`), `text.join()` writes a `null` element as `null`, and `text.hexValue()` is upper case and zero-padded (`255` is `00FF`).
- `text.sorensenDiceSimilarity()` compares the bigrams of each word as a multiset (`'aa'` and `'aaaa'` is 0.5) and takes an optional language tag, `text.camelCase()` and `text.upperCamelCase()` no longer capitalize the first word after a leading separator, and `text.capitalizeAll()` and `text.decapitalizeAll()` split the words at spaces only and trim the result.

Impact: a query that used `text.replace()` or `text.split()` with a delimiter that is a regular expression operator (`.`, `|`, `(`, `[`, `$`) matches something else, or fails to compile the pattern. A filter such as `text.jaroWinklerDistance(a, b) > 0.85` now keeps the dissimilar strings. `text.format()` with the values as separate arguments reads the first as the one value and the second as the language, so a format with several specifiers fails. A stored or compared value made with `text.snakeCase()`, `text.slug()` or `text.hexValue()` changes.

Action: escape the regular-expression characters of the delimiter (`'\\.'` for a dot) or use the openCypher `split()` and `replace()`, write `text.jaroWinklerDistance(a, b) < 0.15` for `> 0.85`, pass the values of `text.format()` as a list, and regenerate the values made with the changed functions.

26.12.1 - `create.vNode`, `create.vRelationship`, the `meta.*` procedures and `path.slice()` answer what APOC answers

These functions and procedures, documented as APOC compatible, now answer what APOC 5.26 answers (issue #12109):

- `create.vNode()` and the `nodes` and `relationships` of `meta.graph()` are virtual nodes and relationships, where they were maps with `_type`, `_id` and `_labels` keys. `labels()`, `type()`, `properties()` and `id()` work on them, and `id()` is negative. `create.vRelationship()` reads its arguments in the documented order `(startNode, type, properties, endNode)`, where it read `(startNode, type, endNode, properties)` and failed on the documented order. The `CALL create.vNode(…) YIELD node` and `CALL create.vRelationship(…) YIELD rel` forms answer the element instead of `null`.
- The third argument of `path.slice()` is a length, where it was an end index: `path.slice(p, 1, 1)` is the second relationship, `path.slice(p, 0, -1)` the whole path, and `path.slice(p)` is accepted.
- `meta.stats()` yields APOC's columns (`nodeCount`, `relCount`, `labels`, …), and counts each node once, where a node of a type with supertypes was counted once per supertype. `meta.nodetypeproperties()` and `meta.reltypeproperties()` list the properties found in the data too, not only those declared in the schema, write `nodeType` and `relType` as `` :`Name` `` , and name the value types as APOC does (`Long` and `Double` instead of `Integer` and `Float`). `meta.graph()` answers

one relationship per start label, type and end label.

Impact: a query that read the `_labels` or other keys of the old maps, called `create.vRelationship()` with the end node third, used `path.slice()` with an end index, or compared `nodeType` or `propertyTypes` with the old values answers differently or fails. `meta.stats()` and `meta.graph()` read every relationship record, so on a large graph they take longer than before.

Action: use `labels(n)` and `n.prop` on the virtual elements, pass the end node of `create.vRelationship()` last, pass the length (`end - start`) as the third argument of `path.slice()`, and update the comparisons with the new `nodeType` and `propertyTypes` values. `YIELD value` of `meta.stats()` still works. See Schema & Meta Procedures, create.vNode() and path.slice().

26.12.1 - An openCypher shortest path from a node to itself answers only a minimum of zero

`shortestPath()` and `allShortestPaths()` whose start and end are the same node answered the zero-length path also when the pattern requires at least one hop: `[*]`, `[*1..]` or a relationship without a quantifier (issue #11913). That path is shorter than the pattern allows. Now only a written minimum of zero (`[*0..]`, `[*0..2]`) answers it; any other pattern has no row for that pair, and in the expression form `shortestPath(…)` returns `null` and `allShortestPaths(…)` an empty list.

Impact: a query such as `MATCH (a), (b) MATCH p = shortestPatha)-[*]→(b` no longer returns the rows where `a` and `b` are the same node, and a count or a minimum length over such rows changes.

Action: write `[*0..]` where the zero-length path of a node to itself is wanted. See shortestPath() and Hop Bounds.

26.12.1 - An openCypher pattern predicate over a null variable is false

A pattern predicate such as `WHERE (x)-→()`, `existsb)-→(x` or `WHERE NOT (x)-→()` that names a node or relationship variable an `OPTIONAL MATCH` left null treated that variable as any node, so it was true whenever the pattern held somewhere in the graph (issue #13124). It is now false, and its `NOT` true, as Neo4j answers. A variable not in scope at all still stands for any node, and `EXISTS { }` subqueries and pattern comprehensions are unchanged.

Impact: rows where the variable is null are no longer kept by the positive predicate and are now kept by `NOT` of it, so an anti-join written as `OPTIONAL MATCH … WHERE NOT (n)-[:R]-()` returns the rows it used to drop.

Action: none, unless a query relied on the old answer.

26.12.1 - openCypher `toString()` of a time or a datetime prints the seconds and trims the fraction, as Neo4j 5 does

`toString()` of a `localtime()`, `time()`, `localdatetime()` or `datetime()` used Java's text, which leaves out zero seconds (`10:00`) and keeps the trailing zeros of the fraction (`.500`). It now prints the seconds always and removes the trailing zeros of the fraction: `10:00:00`, `2020-01-01T10:00:00.5Z` (issue #11912). `toStringOrNull()`, `toStringList()`, `convert.toString()` and `apoc.convert.toString()` print the same, and a `java.time` value bound as a parameter prints as the Cypher temporal it stands for.

Impact: a key, an id or a comparison built from `toString()` of a time or a datetime changes (`'10:00'`

becomes `'10:00:00'`). A temporal value that a query returns, over HTTP or SQL, keeps its text, and string concatenation with a temporal is unchanged.

Action: regenerate or normalize stored text that was made with `toString()` of a time or a datetime.

26.12.1 - An openCypher `CALL` of an unknown namespaced name no longer runs the SQL function of its last segment

A `CALL` of a namespaced name that is neither a procedure nor a function ran the SQL function named by the last segment of the name, whatever the namespace (issue #12110): `CALL apoc.atomic.concat(n, 'name', 'zz')` ran SQL `concat()`, returned no row and wrote nothing, and `CALL nosuch.max(1, 2)` returned 2. Such a call now fails as an unknown procedure, as in Neo4j. `CALL` of an SQL function by its full name (`CALL abs(-5)`, `CALL vector.neighbors(…)`) is unchanged.

Impact: a `CALL anything.max(…)` that relied on the last segment fails with `UnknownProcedureException` (`Neo.ClientError.Procedure.ProcedureNotFound` over Bolt).

Action: call the SQL function by its full name or with the `sql.` prefix: `CALL sql.max(1, 2)`. See CALL.

26.12.1 - Naming an object that does not exist is the client's error on every protocol

A statement that names a type, an index, a bucket, a property, a trigger or a view that does not exist raises a `SchemaException`, a missing record a `RecordNotFoundException`, an unknown method an `UnknownFunctionException`, and a Cypher `CALL` of an unknown procedure an `UnknownProcedureException` (issue #11384). Many of these raised a plain `CommandExecutionException` before.

Impact: over PostgreSQL these answer `42P01` or `42883` instead of `XX000` (see Naming an object that does not exist), over Bolt `Neo.ClientError.Statement.SemanticError` or `Neo.ClientError.Procedure.ProcedureNotFound` instead of `UnknownError`, and over HTTP `400` instead of `500`. Embedded Java code that caught `CommandExecutionException` for a missing type or index no longer catches it. `OPTIONAL CALL` of an unknown procedure now fails instead of returning a `null` row.

Action: catch `SchemaException` (or `ArcadeDBException`) for a missing schema object, and expect the new error codes in clients that keyed on the old ones.

The Java API follows the same rule (issue #11633). A missing edge type in `GraphBatch` or in the hybrid, full-text and vector search functions, a missing bucket behind a Gremlin label, and a missing embedded document type raise a `SchemaException`. A function or function library that does not exist (`Schema.getFunctionLibrary()`, `FunctionLibraryDefinition.getFunction()`) raises an `UnknownFunctionException`. All of these raised an `IllegalArgumentException` before. The batch endpoint (`/api/v1/batch`) still reports what it committed before such an error.

26.12.1 - A write whose outcome is unknown over HTTP answers `504`, not `500`

A write that an HA replica forwarded to a leader which died or did not answer, and a write handed to the replication whose quorum was not confirmed, answered `500` before (the first with `exception: java.io.EOFException` or a connection reset), although the write may have been committed (issue #11887).

Impact: these answer `504` with exception `com.arcadedb.exception.TransactionOutcomeUnknownException`, on the leader and on the replica. A client that handled this case by its `500` status must now handle `504`; the Java remote client does not retry it.

Action: on a `504` from `/api/v1/command` or `/api/v1/commit`, check whether the write landed before sending it again. See Outcome-unknown response contract.

26.12.1 - The line protocol write refuses an unknown `precision`

`POST /api/v1/ts/{database}/write` read a `precision` it did not recognize (`sec`, `seconds`, `h`, a typo) as nanoseconds and answered `204`, so a timestamp in seconds was stored in 1970 (issue #11875).

Impact: any value other than `ns`, `us`, `u`, `ms` or `s` (in any case) answers `400` with an `error` naming the accepted values, and nothing in the request is written. An absent or empty `precision` still means nanoseconds.

Action: send one of the accepted values. See Precision parameter.

26.12.1 - A mistake in a SQL script's own values or order is a client error, not `500`

`SLEEP` of a value that is not a number, an unknown `CONSOLE` level, a `LOCK` after a modification or a second `LOCK`, a `COMMIT` with no transaction open inside a loop or an `IF`, a `BEGIN` in a loop past the nested-transaction limit, and `<`, `⇐`, `>`, `>=` or `BETWEEN` between two records answered `500` before, with a `NullPointerException`, `ClassCastException`, `TransactionException` or `CommandExecutionException` (issue #11893).

Impact: over HTTP these answer `400`, with `TypeMismatchException` (`SLEEP`, comparison), `CommandSQLParsingException` (`CONSOLE`, `COMMIT`) or `CommandSemanticException` (`LOCK`, `BEGIN`), and over the other protocols the matching validation or parsing error. An unknown `CONSOLE` level is now refused when the script is parsed, so none of the statements before it runs. Embedded Java code that caught `TransactionException` around a script with `LOCK` or `BEGIN` must catch `CommandSemanticException` (or `ArcadeDBException`); `acquireLock()` and `begin()` called directly from Java still throw `TransactionException`.

Action: none for a script that ran. A client that retried or alerted on `500` for these cases stops doing so. See SQL Script.

26.12.1 - `algo.maxFlow` follows the relationship direction

`algo.maxFlow` treated every relationship as an undirected pipe, so flow could run against a relationship and two opposite relationships added up (issue #11798). It now follows the relationships outgoing by default, and takes an optional fifth argument, `direction` (`OUT`, `IN` or `BOTH`). See algo.maxFlow.

Action: pass `'BOTH'` as the fifth argument to keep treating the relationships as undirected.

26.12.1 - The second argument of `node.relationship.types()` is APOC's relationship pattern

`node.relationship.types(node, [relTypes], [direction])` (and `apoc.node.relationship.types`) takes

APOC's relationship pattern as its second argument, as APOC does, and the direction moved to the third position (issue #12103). `(n, 'KNOWS>|<LIKES')` answers the types of that list the node has a relationship of, `(n, null, 'IN')` every incoming type. The same release makes `node.degree*`, `node.relationship.exists()`, `algo.dijkstra`, `algo.astar` and `algo.allsimplepaths` read the pattern too (see node functions and the relationship pattern), and the fifth argument of `algo.dijkstra` is APOC's `defaultWeight` when it is a number. An edge type whose name contains `|`, `<` or `>` (or `,` in the `algo.*` procedures) can no longer be named in these arguments.

Impact: a call written for the old signature keeps its answer: a second argument that is a direction word (`IN`, `INCOMING`, `OUT`, `OUTGOING`, `BOTH`, in any case) with no third argument is still read as the direction. In a database with an edge type named exactly like that word (an edge type `OUT`), `(n, 'OUT')` is now the pattern naming that type and answers `['OUT']` or `[]`. A call that passed a type name, `(n, 'KNOWS')`, used to fail as an unknown direction and now answers.

Action: write the direction as the third argument, `(n, null, 'OUT')`, where an edge type has the name of a direction word.

26.12.1 - `algo.kTruss` returns only the vertices of the k-truss when `k` is passed

`algo.kTruss(relTypes, k)` validated `k` and then ignored it, returning every vertex whatever `k` was (issue #12303). A call that passes `k` now returns only the vertices whose truss number is at least `k`, and a `k` below 2 is refused. A call without `k`, or with `null`, still returns every vertex with its truss number. See algo.kTruss.

Action: drop the `k` argument (or pass `null`) to keep receiving every vertex.

26.12.1 - MongoDB wire filters match as mongod does

Filters over the MongoDB protocol (`find`, `count`, `update`, `delete` and the top-level `$match` stages of an aggregation) are evaluated by ArcadeDB's own matcher, which follows mongod's rules, instead of the one bundled with mongo-java-server (issue #12074). Several filters now answer differently: `{a: {}}` matches only an empty document, a regular expression matches only strings, `null` reached through an array, `$mod` on longs, `$gte`/`$lte` against `null`, comparisons of arrays and documents, NaN and `$elemMatch`. A filter with an unknown operator or a malformed operand is refused before any document is read, so an upsert with such a filter inserts nothing.

26.12.1 - `arcadedb.server.defaultDatabases` decodes `%XX` in its credentials and refuses an ambiguous one

A user name, password or group of a `arcadedb.server.defaultDatabases` credential is percent-decoded, so a separator inside it can be written (`%3A` for `:`, `%2C` for `,`, `%3B` for `;`, `%5B` for `[`, `%5D` for `]`, `%25` for `%`), see issue #11269. A credential with more than three `:`-separated fields, an empty user name or an empty password is refused with a WARNING naming the user, and so is an entry with text after its `]` that opens no `{…}` command list.

Impact: a password written with a raw `:` used to be cut at it, the text after it becoming the user's group; that credential now creates no user and grants nothing. A password that contains `%` followed by two hexadecimal digits (`%41`) now reads as the decoded character: an existing user written that way is reported `already exists with a different password` and keeps its password, a new one is created with the decoded password.

Action: write a `%` that is followed by two hexadecimal digits as `%25`, and any `:`, `,`, `;`, `[` or `]` inside a field encoded as above. See `server.defaultDatabases`.

26.12.1 - A schema change refused because of the data already stored is a `ValidationException`

`CREATE PROPERTY` and `ALTER PROPERTY` refused because a record already stored violates the constraint (`MANDATORY`, `NOTNULL`, `MIN`, `MAX`, `REGEXP`), and the same refusal over a sample already stored in a TIMESERIES type, raise a `ValidationException`, whose kind names the constraint where there is one. An update that would change the partition key of a type with `partitioned(…)` bucket selection raises a `ValidationException` too (issue #12037). Before, they raised a `CommandExecutionException` (the partition key an `IndexException`), and `MANDATORY`, `NOTNULL` and the partition key were answered over HTTP with `500`.

Impact: over HTTP these refusals answer `400`, with the constraint in `exceptionArgs`. Embedded Java code that caught `CommandExecutionException` or `IndexException` for these cases no longer catches them.

Action: catch `ValidationException` (or `ArcadeDBException`) for a schema change refused over the stored data. See CREATE PROPERTY.

26.12.1 - A schema change over PostgreSQL answers no result set, and a violated constraint has its own error code

Over the PostgreSQL protocol a schema change (`CREATE TYPE`, `CREATE PROPERTY`, `CREATE INDEX`, `ALTER TYPE`, `DROP TYPE`, `REBUILD INDEX`, …) is answered as PostgreSQL answers `CREATE TABLE`: no result set and a command tag naming the statement, such as `CREATE DOCUMENT TYPE` (issue #11312). The command tag of every statement the server parses comes from the statement, not from its text, so a write with a leading comment reports its row count (issue #11313). A value refused by a constraint declared on its property raises a `ValidationException` that names the constraint (`ValidationException.getKind()`: `MANDATORY`, `NOT_NULL`, `MIN`, `MAX`, `MAX_LENGTH`, `REGEXP`, `READONLY`, `TYPE`, `ENCODING`), a number a property's type cannot hold raises a `ValueOutOfRangeException` (an `IllegalArgumentException`), and a call of a function that does not exist raises an `UnknownFunctionException` (a `CommandExecutionException`), see issue #11314.

Impact: over PostgreSQL, a client that read the row a schema change returned gets none (JDBC `execute()` returns `false` for it), and the refusals answer the SQLSTATEs listed in A refused value and an unknown function instead of `22023` and `XX000`. Over HTTP a violated constraint and an unknown function answer `400` instead of `500`, with the constraint in `exceptionArgs`; over Bolt a violated constraint is `Neo.ClientError.Schema.ConstraintValidationFailed` and an unknown SQL function `Neo.ClientError.Statement.SyntaxError`; over MongoDB a violated constraint is `121` (`DocumentValidationFailure`); over gRPC an unknown function is `INVALID_ARGUMENT` instead of `INTERNAL`. The Java remote clients rebuild the `ValidationException` with its kind.

Action: read a schema change's outcome from the success of the statement rather than from a returned row. Clients that keyed on the old error codes should expect the new ones.

26.12.1 - A database's directory settings must stay inside the server-level directory

`arcadedb.externalPropertyBucketPath` and `arcadedb.pageSnapshotSpillPath` name a directory the server creates files in. A value set for one database (`ALTER DATABASE`, `set database setting`) must now

lie inside the database's own directory or inside the directory the server-level value of the same setting names, compared after resolving `..` and symbolic links. `ALTER DATABASE` of either needs the `updateSecurity` permission, where `updateDatabaseSettings` was enough. A new bucket is never created over a file that already exists at its path.

Impact: setting either one for a database to a directory outside both is refused. A value stored by an earlier version that lies outside both is not used to create anything: the database opens and reads the external-property buckets already there, but a new EXTERNAL property is refused with a `SecurityException` naming the setting, and a snapshot spills into the database directory, each with a WARNING in the log. Setting the server-level value itself is unaffected.

Action: set the server-level value (`-Darcadedb.externalPropertyBucketPath=/mnt/bulk`, or the environment) to the directory the databases may tier into, and give each database a directory under it, or move the per-database value to the server level. Grant `updateSecurity` to the group that manages these settings. See Tiered storage.

26.12.1 - TIMESERIES sealed files record the type they belong to, which older builds cannot read

Every TIMESERIES type created by 26.12.1 draws a random owner id, stored in the schema (`sealedOwner`) and written into the header of every `.ts.sealed` file of its shards, at sealed format version 4 (issue #11274). A sealed file at the type's path that was written for another type, another shard or another database (a file copied in by hand, for example) is refused when the type is opened: the type stays registered with its storage reported unavailable, and the file is neither read nor overwritten. Types of databases written before 26.12.1 have no owner and read their files as before.

Impact: as with the sealed format versions 2 and 3, a sealed file that 26.12.1 rewrites whole (compaction, retention, downsampling) is at version 4 and cannot be read by an older build, and an older build refuses at schema load a type created by 26.12.1 (`sealedFormatVersion` 4). A `.ts.sealed` file copied from another type or database into the place of one of a type created by 26.12.1 is refused. Restoring a whole database directory (the schema and its files together) and HA replication are unaffected.

Action: none to upgrade. To go back to an older build afterwards, export the database with 26.12.1 and re-import it with the older version. If a type reports its storage unavailable because of a foreign sealed file, move that file away and reopen the database. See Integrity Checking.

26.12.1 - A database opened read-write is refused by older builds, unless fixed-width float arrays are turned off

A build before 26.11.1 cannot read the fixed-width float and double arrays introduced in 26.11.1 (see Float and double arrays are stored fixed-width), and 26.9.1 and 26.10.x did not refuse such a database: they read those arrays as `null`, left the records out of `vector.neighbors()`, and the first `UPDATE` of such a record erased its arrays (issue #12234). While `arcadedb.serializer.fixedWidthFloatArrays` is `true` (the default), 26.12.1 labels the dictionary file of a database with format version 2 (`dictionary.<id>.<pageSize>.v2.dict`, same content as version 1) when it creates the database, when it opens it read-write, and when the setting is turned on. Builds from 26.8.1 to 26.11.x refuse to open such a database with `Dictionary '…' was written with format`

`version 2, which this version of ArcadeDB cannot read (latest known is 1)` instead of losing data. The label does not depend on the content: a database that holds no float or double array is labelled too. A database opened read-only is not labelled. If the label cannot be applied (an I/O error), a WARNING is logged and arrays are written in the old varint form until the next open.

Impact: once 26.12.1 has opened a database read-write, no released build opens it any more, 26.11.x included (it reads the fixed-width form but refuses the label). Builds before 26.8.1 do not check the label and still open it, with the data loss described above. In an HA cluster, a node still on an older build that has to install a full snapshot from an upgraded node refuses the dictionary until it is upgraded too.

Action: none to upgrade. To keep a rollback possible, set `arcadedb.serializer.fixedWidthFloatArrays` to `false` before the database is first opened read-write by 26.12.1, or take a backup before the upgrade: the label stays at version 1 and arrays are written in the varint form, which every build reads. Turning the setting off later does not remove the label. To go back to 26.11.x only, stop the server and rename `dictionary.<id>.<pageSize>.v2.dict` back to `dictionary.<id>.<pageSize>.v1.dict`. Never do that to open the database with 26.10.x or older when it may hold float or double arrays: export it with 26.12.1 and re-import it with the older version instead. See `serializer.fixedWidthFloatArrays`.

26.12.1 - `DROP INDEX` refuses the bucket index of an index, and an index missing one is not used

`DROP INDEX` naming the bucket index of an index (for example `S_0_4559193709411875`, the part of `S[k]` on one bucket), from SQL or Cypher, is refused with a `SchemaException` naming the index it belongs to (issue #11303). An index that lacks the bucket index of one of the buckets it covers (left by an interrupted rebuild of an earlier version) is no longer used by any query planner: queries fall back to a scan. `database.lookupByKey()` on it raises an `IllegalArgumentException`, and Cypher `SHOW INDEXES` reports it with state `FAILED`.

Impact: a script that drops a bucket index on its own fails. Queries that read such an incomplete index answer every row, more slowly, where they used to return short results.

Action: drop the index itself instead of one of its bucket indexes, and run `REBUILD INDEX <name>` on an index `SHOW INDEXES` reports as `FAILED`. See `DROP INDEX` and `REBUILD INDEX`.

26.12.1 - An index on an undeclared property written by an older version finds its old list and number keys

An index on a property the schema does not declare keys each value by its text. 26.10.1 changed that text for a floating point or decimal number (`1.0` is keyed `1`, `-0.0` is keyed `0`, `1e300` by its full digits, where 26.9.1 wrote `1.0`, `-0.0` and `1.0E300`), and 26.12.1 for a list holding such a number (`[1.5, 2.0]` is keyed `[1.5, 2]`). Nothing re-keyed the entries an older version wrote, so after an upgrade a seek missed those records, `MERGE` created a second node, `IS UNIQUE` accepted a copy, a delete or an update left the old entry behind, and `CHECK DATABASE` reported errors (issues #12160, #12233). Such an index now also reads, removes and checks the old spelling of a list or number key, and a `STRING` spelled like an old key (`'1.0'`, `'[1.5, 2.0]'`) is still a different value. An old entry is found by the text it was written with: `1.0` finds the entry 26.9.1 wrote for `1.0`, and `1` finds it only after a rebuild.

Impact: every `LSM_TREE` index created before 26.12.1 is treated as possibly holding old keys. On an undeclared property, its seeks, removals and unique checks of a floating point number or of a list

holding one pay an extra read, a seek of a `STRING` that reads as a number or a list (`1.0`, `2.500`, `[1.5, 2.0]`) loads the record behind each entry it finds, and a unique `STRING` key that reads as a number or a list (`12345`, `2.5`) loads one record at commit. Any other text pays nothing, and so does an index created by 26.12.1.

Action: run `REBUILD INDEX <name>` (or `REBUILD INDEX *`) once after upgrading: it writes every entry under its current key and removes the extra cost. See `REBUILD INDEX`.

26.12.1 - A materialized view keeps the brackets of its query, and `@cat` is a reserved name

A materialized view or a continuous aggregate stores its `SELECT` as text. Earlier versions dropped brackets from that text: `a = 1 AND (b = 2 OR c = 3)` was stored as `a = 1 AND b = 2 OR c = 3`, and `NOT (x AND y)` lost its brackets too, so the view held rows its query does not return (issue #11299). A back-ticked `@cat` was stored without its back-ticks, and the view could not be created (issue #11308). The stored text now means what was written. `@cat` is now a token of its own like `@rid` and `@type`, so it can be written without back-ticks.

Impact: a view created by an earlier version keeps the text it stored, and keeps answering the wrong rows.

Action: drop and recreate any materialized view or continuous aggregate whose query has a bracketed `OR` or a `NOT (⋯)`. See `CREATE MATERIALIZED VIEW`.

26.12.1 - Bolt enforces the access mode and `tx_timeout`, and `DISCARD n` discards `n` records

A write in a READ transaction or a READ auto-commit query (`session.executeRead`, a READ session) is refused with `Neo.ClientError.Statement.AccessMode`. The `tx_timeout` a driver sends (`TransactionConfig.withTimeout()`) bounds the explicit transaction or the auto-commit query, failing with `Neo.ClientError.Transaction.TransactionTimedOut`. `DISCARD n` discards the next `n` records and keeps the result open while records remain (issue #11392).

Impact: code that wrote inside `executeRead`, or whose transactions ran longer than the timeout it declared, now fails. A client that relied on `DISCARD n` discarding the whole result gets `has_more = true` back.

Action: move writes to `executeWrite` or a WRITE session, and size `tx_timeout` for the work. See Access mode and transaction timeout.

26.12.1 - An index equivalent to an existing one answers a duplicate code

Creating an index or constraint equivalent to an existing one under a different name answers `Neo.ClientError.Schema.EquivalentSchemaRuleAlreadyExists` over Bolt and `42P07` (`duplicate_table`) over PostgreSQL, as a same-name duplicate does (issue #11392).

Impact: over PostgreSQL this was `22023` (`invalid_parameter_value`).

Action: clients that keyed on the old code should expect the new one. See Creating an object whose name is taken.

26.12.1 - An MCP configuration update that cannot be saved is refused

`POST /api/v1/mcp/config` writes `mcp-config.json` before applying the change. If the file cannot be written, the request answers HTTP `500` naming the file and the configuration in force is unchanged (issue #11390).

Impact: with a read-only configuration directory (a ConfigMap mount, a read-only root filesystem) an update used to answer `200`, apply in memory only and be lost at the next restart; it is now refused.

Action: make the configuration directory writable, or change the mounted file instead. See Configuration Endpoint.

26.12.1 - The gRPC client raises typed exceptions for streamed reads and `UNAUTHENTICATED`

A failure on a later batch of `iterateType`, `iterateBucket` or `queryStream` raises the same exception a unary call does (`TimeoutException`, `SecurityException`, ...) instead of a bare `RuntimeException`, an interrupt raises a `RemoteException` with the interrupt flag kept, and a bare `UNAUTHENTICATED` status raises a `SecurityException`, also from `RemoteGrpcServer.listDatabases()` (issue #11393).

Impact: code that caught `RemoteException` for an authentication failure no longer catches it; code that caught `RuntimeException` still does.

Action: catch `SecurityException` for an authentication or permission refusal, and `TimeoutException` for an expired deadline. See Java API (gRPC).

26.12.1 - The gRPC Java client returns record rows and record columns as records

A row that returns a record, such as Cypher `MATCH (n) RETURN n`, now reaches a `RemoteGrpcDatabase` caller as that vertex or edge with its own properties, and a column that holds a record (`RETURN a, r, b`, SQL `SELECT @rid, @type, @this`, the records of `collect(n)`) as a `Document`, `Vertex` or `Edge`, as the embedded API returns them (issue #12196). Before, `RETURN n` gave a vertex whose only property was `n`, `SELECT @rid, @type, @this` a record whose only property was `@this`, and the other record columns raw maps. A projection row with a `@type` column, such as `SELECT @type, name FROM P`, is now a projection row with both columns; before, it reached the client as a record of type `P` without the `@type` column (issue #12519).

Impact: code that read the properties through the alias (`vertex.get("n")`) or cast a record column to `Map` no longer works. A map a query builds with a valid `@rid` and the name of an existing type as `@type` is also returned as a record. The columns of a projection keep the order of the query. In an HA cluster, a command a follower forwards to the leader still answers a row that has a `@type` column as a record of that type.

Action: read the properties from the record itself (`result.getVertex().get().get("name")`), and treat a record column as a `Document`, `Vertex` or `Edge`. See Java API (gRPC).

26.12.1 - The HTTP Java client returns record columns as records

A column of a projection row that holds a record, such as Cypher `MATCH (a)-[r]→(b) RETURN a, r, b`, SQL `SELECT @this AS me, name` or `SELECT first(out('Knows')) AS friend`, reached a `RemoteDatabase`

caller as a list of RIDs, with every property lost: a vertex as `[#1:0]`, an edge as `[edge rid, in rid, out rid]`. It now arrives as a `Document`, `Vertex` or `Edge` with its properties, as the embedded API returns it, and so do the records held in a list, a map or a path of a column (`collect(a)`, `{who: a}`) (issue #12184). A row that is itself a record (`RETURN a`) was not affected.

Impact: code that read such a column as a `List` of RIDs no longer works. A map a query builds with a `@cat` of `d`, `v` or `e` and a `@type` is also returned as a record. The change is in the client alone and works with any server version.

Action: treat a record column as a `Document`, `Vertex` or `Edge`, and call `getIdentity()` on it where only the RID is needed. See Java Remote API.

26.12.1 - A vector index whose definition in `schema.json` is invalid is left out at open, alone

When a database opens, the settings of each `LSM_VECTOR` index in `schema.json` are checked with the same rules as `CREATE INDEX` (`pqClusters` at most 256, `maxConnections`, `beamWidth`, `efSearch`, `pqTrainingLimit` at least 1, ...). An index whose definition breaks one is left out: it is not used, queries on its type scan instead, its definition and its file are kept, and a `SEVERE` log line names the index, the invalid value and the fix (issue #11568).

Impact: such a definition, typically from a hand-edited or restored `schema.json`, used to be accepted as is, or, for a value already checked at open (an unknown similarity, a negative `rerankOversample`), to stop the schema load half way: the database opened without every index loaded after it, UNIQUE ones included. A left-out UNIQUE index does not enforce its constraint until it is fixed, and the log says so.

Action: correct the value in `schema.json` and reopen the database, which brings the index back with its data, or create the index again with `CREATE INDEX`. See Product Quantization.

26.12.1 - A `READ_COMMITTED` commit is refused when it read stale records from a page it had written

Once a `READ_COMMITTED` transaction has written a bucket page, it reads that page from its own copy, taken at its first write, which does not show what other transactions committed there afterwards. The commit-time merge (`txPageSlotMerge`, and `arcadedb.graph.edgeAppendMerge` for edge lists) used to re-apply the transaction's writes on the newer page and the commit succeeded, so a statement decided on those stale reads was committed: an `UPDATE` or `DELETE` that matched none of the rows committed before it ran, a Cypher `MERGE` that created a second node, a relationship `DELETE` that found nothing (issue #11590). Now such a commit fails with a retryable `ConcurrentModificationException`. A page whose records the transaction read still hold the same content on the committed page keeps merging, and so do blind inserts and updates.

Impact: a transaction that writes a page and then reads records of the same page that another transaction changed meanwhile can now be refused where it used to commit, and so can a transaction that adds a relationship to a vertex and then traverses or counts that vertex's relationships in the same direction while another transaction adds one there too. `database.transaction(…)` and the other retrying APIs retry it, and the retry reads the committed records. `REPEATABLE_READ` is unchanged.

Action: none, beyond retrying on `ConcurrentModificationException`, which every concurrent writer

already has to. See Isolation.

26.12.1 - A delete of a record changed since it was read is refused

Under `READ_COMMITTED` a delete removed whatever sat at the record's RID when it got there, with no check that it was still the record read (issue #12472). If another transaction deleted the record and an insert reused its RID, the delete removed the unrelated new record and committed without an error; a SQL `DELETE … WHERE` racing an `UPDATE` removed a record the update had made stop matching. Now a delete of a record that another transaction changed or replaced after it was read fails with a retryable `ConcurrentModificationException`, as an update in the same position already did. A record deleted by its RID without being loaded first is deleted as it is found.

Impact: under contention more deletes are retried. `database.transaction(…)` and a single-request command over HTTP retry them; a transaction spanning several HTTP requests now receives the exception at the delete instead of a silent wrong delete.

Action: none, beyond retrying on `ConcurrentModificationException`, which every concurrent writer already has to. See Isolation.

26.12.1 - The remote Java client sends links, decimals, binary and date-times with their type

The remote Java client sent query parameters and the values of a saved record as plain JSON, so on a property the schema does not declare a `RID` was stored as a STRING, a record set as a value as an EMBEDDED copy, a `BigDecimal` as a DOUBLE (`SELECT :a + :b` with `0.1` and `0.2` answered `0.30000000000000004`, and a declared DECIMAL lost its scale), a `byte[]` as a LIST, and a `LocalDate`, `LocalDateTime`, `java.util.Date`, `Instant` or `ZonedDateTime` as a number or a string. A `LocalDate` or `ZonedDateTime` saved with `save()` was then not found by `WHERE p = :v` with the same value. The client now sends these values as typed markers (`$rid`, `$decimal`, `$bytes`, `$datetime`, `$localDateTime`, `$instant`, `$zonedDateTime`, next to `$date`), and they are stored as LINK, DECIMAL, BINARY, DATE, DATETIME and ZONED_DATETIME, as the embedded API stores them. A malformed marker is refused with HTTP 400 before the statement runs (issues #12186, #12187).

Impact: there is no version negotiation: a server older than this change receives a one-key map for such a parameter or saved value. `Short`, `Byte` and `Float` still travel as JSON numbers. A record read back still holds an undeclared date, decimal or binary property as a string, a double or a list, so `modify()` and `save()` write such an untouched property back with that type (issue #12599). An undeclared link now reads back as a `RID` (see the next note).

Action: upgrade the server before, or together with, the client. See Java Remote API.

26.12.1 - The remote Java clients read a link back as a `RID`

A link stored in a property the schema does not declare was read back by the HTTP client (`RemoteDatabase`) as its rid text, such as `"#1:0"`, and a list or map of links as a list or map of strings, while a projection of the same single link answered a `RID`. The gRPC client (`RemoteGrpcDatabase`) answered the rid text for every link, in records and projections. `(Identifiable) doc.get("friend")` threw `ClassCastException`. Both clients now answer a `RID`, and a list or map of `RID`s, as the embedded API does. For HTTP the server marks those properties in the record it sends; a string that only looks like a RID, such as `'#1:0'` stored as text, still reads back as a string (issue #12195).

Impact: code that read a link through either client and expected a `String` now gets a `RID`. The `RID` is not bound to the remote database, so `asDocument()` or `asVertex()` on it fails on a thread with no active database (issue #12603). Links in a property declared as a bare `LIST` or `MAP`, without `OF LINK`, still read back as strings over HTTP (issue #12598).

Action: call `toString()` on the value where the rid text is needed, and load the linked record with `lookupByRID(rid, true)`. For HTTP, upgrade the server; for gRPC, upgrade the client. See Java Remote API and Java API (gRPC).

26.12.1 - The remote Java clients read numeric arrays as primitive arrays

A property declared `ARRAY_OF_SHORTS`, `ARRAY_OF_INTEGERS`, `ARRAY_OF_LONGS`, `ARRAY_OF_FLOATS` or `ARRAY_OF_DOUBLES` was read from a record by the HTTP client (`RemoteDatabase`: `lookupByRID()`, `iterateType()`, a `SELECT FROM` row) as an `ArrayList` of whatever numbers the JSON held (`Integer` for a long array, `Double` for a float array), while a projection of the same property answered the primitive array. The gRPC client (`RemoteGrpcDatabase`) answered a list in records and projections alike. `(long[]) doc.get("ids")` threw `ClassCastException`. Both clients now answer `short[]`, `int[]`, `long[]`, `float[]` and `double[]`, as the embedded API does (issue #12189). Over gRPC the server marks every primitive numeric array it sends with its `logical_type` (`long[]`, `float[]`, …), so a vector embedding also arrives as a `float[]`; a client that ignores the mark still reads a list.

Impact: code that read such a value through either client as a `List` now gets the primitive array.

Action: read the value as the primitive array. For HTTP, upgrade the client; for gRPC, upgrade the server and the client. See Java Remote API and Java API (gRPC).

26.12.1 - The remote Java clients return a vertex's edges in RID order

`Vertex.getEdges()` and `getVertices()` on a vertex read through `RemoteDatabase` or `RemoteGrpcDatabase` fetched a neighbourhood larger than one page (2,000 edges) with an offset into the edge list, so a loop that deleted the edges or the vertices while iterating removed only 2,500 of 4,500, and edges another client deleted meanwhile made the iteration skip live ones (issue #12191). Each page now resumes after the RID of the last edge read.

Impact: the edges, and the vertices at their other end, come in RID order instead of the order of the edge list; `BOTH` returns the outgoing ones, then the incoming ones. A vertex deleted since its edge was read is left out, so `getVertices()` can return fewer vertices than there are edges. `getEdgesPaged()` and `getVerticesPaged()` throw `IllegalArgumentException` for a page size of zero or less. The single-page `getEdges(direction, limit, skip, …)` and `getVertices(direction, limit, skip, …)` are unchanged.

Action: none, unless the code relied on the order of the edges, in which case sort them. See Java Remote API.

26.12.1 - The HTTP Java client's `iterateType()`, `iterateBucket()` and streams return every row

`RemoteDatabase.iterateType()`, `iterateBucket()`, `queryStream()` and `commandStream()` stopped at the server's default row cap for buffered responses (`arcadedb.server.httpQueryDefaultLimit`, 20,000 rows) unless the application called `setMaxResultRows()`, while `countType()` reported the full count (issue #12185). `iterateType()` and `iterateBucket()` now return every record, and the streams every

row, up to the server's hard ceiling `arcadedb.server.httpQueryMaxResultRows`. A result above the ceiling is refused with an exception instead of returned in part. The same applies to `queryStream(language, query, params)` on `RemoteGrpcDatabase`, which streams over HTTP. The buffered `query()` and `command()` are unchanged.

Impact: `iterateType()` and `iterateBucket()` ignore `setMaxResultRows()`. `queryStream()` and `commandStream()` still honor it.

Action: none. Code that called `setMaxResultRows(-1)` only to stream every row can drop the call. See Streaming results.

26.11.1 - MCP writes need `allowSchemaChange` to create a type

A Cypher `CREATE` or `MERGE` naming a type that does not exist creates it. Through MCP (`execute_command`, `upsert_entity`, `upsert_relationship`) this happened even with `allowSchemaChange` off. It is now refused.

Impact: with the default MCP configuration (`allowSchemaChange: false`) the upsert tools no longer create missing types.

Action: create the types first, or enable `allowSchemaChange` for the agent. See MCP.

26.11.1 - A Postgres, Redis or BOLT connection must authenticate within 30 seconds

A connection that has not authenticated within `arcadedb.network.preAuthTimeout` (30 s) of being accepted is closed, even if it keeps sending data. Before, only a silent connection was closed.

Impact: a client that waits more than 30 s before sending its credentials is disconnected.

Action: none for normal clients. Raise `arcadedb.network.preAuthTimeout` if needed, or set it to 0 to disable it.

26.11.1 - Schema listings hide the types a user may not read

The PostgreSQL catalog (`\dt`, `information_schema`, `pg_class`...), GraphQL introspection, the MCP `get_schema` tool, the Studio AI assistant, the gRPC database info and the Cypher `meta.*` procedures now list only the types the user can read, as `SELECT FROM schema:types` already did.

Impact: a user restricted by per-type permissions no longer sees the other types in those listings.

Action: none.

26.11.1 - Float and double arrays are stored fixed-width, which older builds cannot read

An `ARRAY_OF_FLOATS` or `ARRAY_OF_DOUBLES` value (a vector embedding, for example) is now written as its element count followed by the raw little-endian IEEE-754 elements, 4 bytes per float and 8 per double, where each element used to be a zigzag varint of 5 bytes per float and 9 to 10 per double (issue #10856). A record holding a 1536-dimension float embedding shrinks from about 7,695 to about 6,150 bytes, about 22% less disk and page cache. Records written before keep the old form and are still read; an update rewrites the value in the new form. In an HA cluster no node writes the new form until every peer advertises the `fixed-width-float-arrays` capability, so a rolling

upgrade keeps writing the old form until the last node is upgraded.

Impact: once a database holds records in the new form, a build older than 26.11.1 cannot read them.

Action: none to upgrade, and no migration is needed. To go back to an older build afterwards, export the database with 26.11.1 and re-import it with the older version. See Data Types.

26.11.1 - A Graph Analytical View rebuilds once after the upgrade, and a failed build reads `FAILED`

The CSR a view persists at a clean close is written in a new format (version 5), streamed in frames so that writing and reading it need no more heap than the view itself (issue #11011). A file written by an earlier version is not read: the first open after the upgrade rebuilds the view from a full scan, as an open after an unclean stop does, and the next clean close writes the new format. A build that fails, an `OutOfMemoryError` included, now ends in the new `FAILED` status with a `buildError` column in `schema:graphAnalyticalViews`, where it used to read `BUILDING` for good after an out-of-memory error. The default of `arcadedb.cypher.algoMaxWorkingMemory` is now `0`, sized from the heap at each call and never below the eighth of the heap it used to be.

Impact: one full rebuild of each persisted view on the first open. Code that compares `GraphAnalyticalView.getStatus()` against every value of `Status` meets a new one, `FAILED`.

Action: none. See Sizing the Heap for a View.

26.11.1 - The rows of the vector search functions no longer list the record twice

Each row of `vector.neighbors()`, `vector.sparseNeighbors()`, `vector.fuse()`, `vector.rerank()` and the other vector functions holds the record's properties, `@rid`, `@type` and its score, and used to hold the whole record again under a `record` key. `SELECT expand(…)`, `SELECT *` and the JSON of the rows serialized every property twice, embeddings and passages included (issue #10898). The `record` key is no longer listed: a path such as `record.name`, a projection of `record`, and a Java caller that reads the key from the row still get the record.

Impact: a client that read the nested `record` object from the JSON of an expanded or unprojected row no longer finds it.

Action: read the properties from the row itself, or project `record` explicitly (`SELECT record, distance FROM (SELECT expand(vector.neighbors(…)))`). See vector.neighbors().

26.11.1 - A vector index object held across `REBUILD INDEX` refuses to answer

A Java caller that kept the bucket-level `LSMVectorIndex` or `LSMSparseVectorIndex` from before a `REBUILD INDEX` or a drop was answered as if the index were empty (0 neighbors, 0 entries) by the dense index, and with an `IllegalStateException` ("engine is closed") by the sparse one (issue #10897). Both now throw an `IndexException` ("Index '...' is not valid. Probably has been drop or rebuilt") on their searches, `get()` and `countEntries()`, as the `TypeIndex` over them and an `LSM_TREE` bucket index already did. `getStats()` still answers: it describes that instance.

Impact: code that held such an object and read an empty answer now gets the exception.

Action: fetch the index again by name from the schema after a rebuild.

26.11.1 - Function options refuse a value of the wrong kind

A boolean option of a function's options map (`parallel`, `tieBreaker` and `emptyIfMaxDepth` of `astar()` and `dijkstra()`, `edge` of `shortestPath()`, and every other boolean option read through the same option reader) read any value other than a boolean or `'true'` as `false`, so a typo such as `'ture'`, `'yes'` or `1` silently ran another query. It now accepts only a boolean or the strings `'true'` and `'false'` (in any case), and refuses any other value with an error that names the option (issue #10788). `maxDepth` must be a non-negative whole number: `2.5`, `'abc'` and `-1` are refused the same way. For `shortestPath()`, `maxDepth: null` now means no bound, where it was read as 0.

Impact: a query that passed such a value fails where it used to answer.

Action: pass `true` or `false`, and a whole `maxDepth`. See astar(), dijkstra() and shortestPath().

26.11.1 - The default-named indexes of a renamed type answer to the new name

After `ALTER TYPE Foo NAME Bar`, the index `Foo[a]` was listed as `Bar[a]` but resolved only by `Foo[a]`, so `DROP INDEX` and `REBUILD INDEX` with the listed name failed, and an index created with a name of its own lost it to the default name. A default-named index now answers to `Bar[a]` at once and no longer to `Foo[a]`, and an index with a name of its own keeps it across the rename and a restart (issue #10751).

Impact: a script that reached a renamed type's index by its old name fails, since that name no longer resolves.

Action: use the name the schema lists. See ALTER TYPE.

26.11.1 - SQL refuses a placeholder with no key in the parameter map

A `?` or a `:name` bound through a parameter map (`query`/`command` with a `Map` for `sql` and `sqlscript`, the HTTP `params` given as an object, the GraphQL `@sql` directive) or run with no parameters at all was bound to `null` when the map had no key for it, without a word: an `INSERT` stored `null` and a `WHERE v = ?` matched nothing. Such a statement is now refused before it runs with a `CommandParameterMissingException` (HTTP 400, `Expected parameter(s): v`), as openCypher already refuses it (issue #11098). A key bound to `null` still binds `null`. A `?` is keyed by its position (`"0"` for the first), a `:name` by its name, `:name` or its position. A SQL function defined with `DEFINE FUNCTION` and called with fewer arguments than it declares binds the omitted ones to `null`, and so does an argument a GraphQL query omits for a native-query directive (or its default value, when the field declares one).

Impact: code that relied on a missing key being bound to `null` fails where it used to answer.

Action: bind every placeholder, with an explicit `null` where that is what you mean.

26.11.1 - `pow()` and `sqrt()` answer the value, not the width of the argument

`pow()` answered null for every negative base and narrowed its result into the type of the base (`pow(2, 31)` returned `2147483647`, `pow(s, 3)` on a SHORT holding 300 returned `-832`), and read a

fractional exponent as a whole one (`pow(2, 2.5)` returned 4). `sqrt()` rounded the root of an integral argument down into its type (`sqrt(2)` returned `1`). Both now answer the value (issue #11047): an exact integral power widens to a Long and then to an exact Decimal, a fractional exponent and a negative exponent of an integral base answer a Double, a negative base is accepted, and the root of an integral or Float argument is a Double. A power no type can hold raises an arithmetic error instead of a saturated number.

Impact: `sqrt(4)` returns `2.0` (a Double) where it returned `2`, and `pow()` of an INTEGER base can return a Long or a Decimal.

Action: none, unless code relied on the type of the result; cast it with `.asInteger()` or `.asLong()` where an integral value is needed. See pow() and sqrt().

26.11.1 - SQL refuses positional values that do not match the placeholders

A call that binds positional values (`query`/`command` with `Object…` for `sql` and `sqlscript`, the HTTP `params` given as a JSON array or as an object keyed `"0"`, `"1"`, …, and the remote Java client) dropped a value no placeholder read and bound a placeholder with no value to `null`, without a word. A list spread into several values for one `IN ?` answered fewer rows than the caller meant. The values are now counted against the placeholders before the statement runs, and a mismatch is refused with a `CommandSQLParsingException` (HTTP 400) that names both counts (issue #10721). Each `?` and each positionally bound `:name` takes exactly one value, a script counts the placeholders of all its statements, and a Postgres-style `$N` may leave lower positions unread. Values bound by name (a map) are not counted.

Impact: code that passed a surplus value, or relied on a missing value being bound to `null`, fails where it used to answer.

Action: pass one value per placeholder, `null` included, and a list meant for one `IN ?` as one value (a `List` or a Java array). See positional parameters over HTTP.

26.11.1 - A plain `CHECK DATABASE FIX` relinks orphan edge records

An orphan edge record, whose `@out` and `@in` are live vertices but that no vertex's edge list references, is reported as `ORPHAN RECORD` under `unreachableEdgeRecords`. A plain `CHECK DATABASE FIX` left it untouched, and only `FIX DELETE ORPHANS` repaired it, by deleting it. It now relinks it into the outgoing list of its `@out` vertex and, for a bidirectional edge type, the incoming list of its `@in` vertex (issue #10741). The result counts them under `relinkedOrphanEdgeRecords`. The usual source is a `GraphBatch` whose process died before `close()`, and `CHECK DATABASE FIX` is now the supported recovery for it.

Impact: edges that no traversal reached before the repair are traversed after it, and counted by every query that walks those vertices.

Action: none if those edges are wanted. To remove them instead, run `CHECK DATABASE FIX DELETE ORPHANS`, which takes precedence. See CHECK DATABASE and GraphBatch.

26.11.1 - `count(*)` repeats inside a `REPEATABLE_READ` transaction

`count(*)` on a type and `countType()` followed the commits of other transactions at every isolation

level, so inside one REPEATABLE_READ transaction two counts could differ, and a count could disagree with a scan. Under REPEATABLE_READ the committed counter of each bucket is now read at the first count of the transaction and kept, plus the transaction's own inserts and deletes (issue #10712). READ_COMMITTED is unchanged.

Impact: a REPEATABLE_READ transaction that polled `count(*)` to watch other transactions' inserts no longer sees them change.

Action: poll with separate transactions, or at READ_COMMITTED. See Isolation.

26.11.1 - The HTTP server's IO thread count is `arcadedb.server.httpIoThreads`

The setting that sets the number of IO threads of the HTTP server was keyed `arcadedb.server.httpsIoThreads`, which read as the HTTPS-only setting declared beside it, so the natural spelling `-Darcadedb.server.httpIoThreads` was ignored without a word (issue #10599). Its key is now `arcadedb.server.httpIoThreads`. The old key is still read from a system property, an environment variable, the server configuration file and the settings commands, and the first use of it is reported in the log.

Impact: none for a deployment that set the old key. A deployment that set `arcadedb.server.httpIoThreads` before, and saw no effect, now gets the value it asked for. When both are given, the new key wins.

Action: none. Switch to the new key when convenient, the old one may be removed in a later release. See server settings.

26.11.1 - A MongoDB `deleteMany` over a TIMESERIES type no longer accepts a number for the timestamp

`find` and `count` over a TIMESERIES type hand the timestamp to a MongoDB client as a date and, as MongoDB does, do not match a number against it. A `deleteMany` with a number on the timestamp (`{ts: 1792396800000}`, `{ts: {$gte: <number>, $lt: <number>}}`) was accepted and removed the samples at those epoch milliseconds, so the usual preview with `find` or `count` and then delete of the same filter removed samples the preview had shown none of (issue #10598). It is now refused with an error that names the type and asks for a date.

Impact: a client that deleted time-series samples by a number of milliseconds gets an error instead of a deletion. Nothing is removed by a refused delete.

Action: compare the timestamp with a date (a `Date`, a `datetime`, a BSON date): `deleteMany({ts: {$gte: ISODate("2026-10-19T08:00:00Z")}})`. On a type restored from an export written before `MILLISECOND` became the only precision, and still declaring `SECOND`, `MICROSECOND` or `NANOSECOND`, no condition on the timestamp is accepted, a date or a number, because the unit of its timestamps is not known: the error names `ALTER TIMESERIES TYPE <name> PRECISION MILLISECOND`, which confirms the unit and opens the date form (issue #10740). See Query Languages.

26.11.1 - A DATETIME string with an offset is stored as the UTC wall clock on every path, whatever the database time zone

The conversion the gRPC service and the other callers of `DateUtils.dateTimeToTimestamp` apply to a

string moved the instant onto the database's time zone before storing it as a UTC wall clock, so a database whose zone is not UTC stored `2020-01-02T03:04:05+02:00` hours away from the same instant written through SQL, and `2020-01-02T03:04:05Z` shifted by the zone's offset (issue #10498). Every path now stores the UTC wall clock of the instant, as SQL, HTTP and Bolt already did.

Impact: none for a database whose zone is UTC. In a database with another zone, a `DATETIME` written from a string with an offset or a `Z` through the gRPC service (or another bulk path) before the upgrade holds a value shifted by the zone's offset; values written after the upgrade are right. Stored values are not changed.

Action: none for a UTC database. Otherwise rewrite the values loaded that way, by reloading them or by correcting them with an `UPDATE` on the records those loads wrote. See Managing Dates.

26.11.1 - A pattern named in asDateTime(), date() and date.parse() converts the offset of the text, and the time zone argument of date() is used

`'2020-01-02T03:04:05+02:00'.asDateTime("yyyy-MM-dd'T'HH:mm:ssXXX")` and `date('2020-01-02T03:04:05+02:00', "yyyy-MM-dd'T'HH:mm:ssXXX")` answered `2020-01-02T03:04:05`, the offset the pattern had read dropped, where the same text read with the schema's own pattern, a bound parameter or an `OffsetDateTime` gave `2020-01-02T01:04:05` (issue #10596). A named pattern is now read under the same rule: the instant the text names is the UTC wall clock. The third argument of `date()`, which was ignored, is the zone a text that names none was written in. `date.parse()` and `date.fields()` read the offset of the text, and so does the `datetimeFormat` of the importer.

Impact: a query or an import that names a pattern with an offset or zone field and a text with a non-zero offset gets the converted value, two hours earlier in the example; one that passes a time zone to `date()` for a text without an offset gets the instant of that zone in UTC. Stored values are not changed.

Action: none if the converted value is what the query meant. To keep the wall clock written, drop the offset from the text, or read it with a pattern that has no offset field. See Managing Dates.

26.11.1 - SQL equality on a list or a map compares the elements as it compares two scalars

`[<Date>] = v`, where `v` holds the `LocalDateTime` of the same instant, was false and `<>` was true; a `Timestamp` on the left of `=` was never equal to the same instant (issue #10595). Two values `=` reports equal are now equal as the elements of a list or a map too, for `=`, `<>`, `IN` and `CONTAINS`.

Impact: a filter on a list or map property compared with a parameter that holds the same instants in another Java class (`Date`, `Instant`, `Calendar`, `Timestamp`) now matches the row. A list of one kind compared with a list of another that converts matches as the scalars do: `[1] = ['1']` is true, as `1 = '1'` is.

Action: none. Compare with a typed parameter if a list of strings must not equal a list of numbers.

26.11.1 - Redis commands on in-memory keys are no longer refused with `TRYAGAIN`

`GET`, `SET`, `GETDEL`, `EXISTS`, `INCR`, `INCRBY`, `INCRBYFLOAT`, `DECR` and `DECRBY` no longer wait for the query admission control, and are no longer listed by `list queries` / `SHOW TRANSACTIONS` (issue #10499). Each is a single in-memory map operation.

Impact: when the server is busy these commands are answered instead of refused with `TRYAGAIN`. A client that used `TRYAGAIN` on them as a back-pressure signal no longer receives it. The commands that read or write records (`HSET`, `HGET`…) are still gated.

Action: none for most installations.

26.11.1 - A database that a restore holds or a `close database` closed is answered as unavailable on every protocol

A database that exists but cannot serve now (an `overwrite` restore holds its name, a `close database` closed it, it is being dropped) used to reach a Redis or MongoDB client as a generic server error, and a Redis write to `<database>.<key>` was kept as a plain key of the connection and acknowledged with `OK` (issue #10553). Every protocol now answers that the database is not available: `LOADING` for Redis, error `365 TemporarilyUnavailable` for MongoDB, `UNAVAILABLE` for gRPC, `57P03` for Postgres, `DatabaseUnavailable` for BOLT. A name that is not a database keeps its answer.

Impact: a client that treated the old Redis or MongoDB error as a server fault now sees a retryable "not available" and can try again later. A Redis write to a held database is refused, where it used to be acknowledged and lost once the database came back.

Action: none for most installations. See Opening a database on demand.

26.11.1 - The Java remote clients raise the engine's exception for a failing statement

A failing statement used to reach a `RemoteDatabase` or `RemoteGrpcDatabase` caller as an untyped `RemoteException`, which was not an `ArcadeDBException`, for the whole command family: a syntax error, an unknown function, a missing bucket, an `INSERT` sent through `query()` (issue #11045). The clients now raise the exception the embedded engine raises (`CommandSQLParsingException`, `CommandExecutionException`, `QueryNotIdempotentException`, `IllegalArgumentException` for an unknown query language, …), with the server's sentence as the message, and `RemoteException` extends `ArcadeDBException`.

Impact: a `catch (RemoteException e)` around a statement no longer catches these failures; a `catch (ArcadeDBException e)` now catches them, and every other `RemoteException` too. Code that parsed the class name out of a message of the form `Error on executing remote operation … (cause:<class> detail:<message>)` receives the type instead.

Action: catch `ArcadeDBException`, or the specific engine exception, where the code caught `RemoteException` for a failing statement. See Remote Java API.

26.11.1 - The vector functions return `NULL` for a `NULL` vector, and `0` or `false` for an empty one

`vector.magnitude()` (`vector.l2Norm()`), `vector.normalize()` and the two-vector functions (`vector.cosineSimilarity()`, `vector.dotProduct()`, `vector.l1Distance()`, `vector.l2Distance()`, `vector.add()`, `vector.subtract()`, `vector.multiply()`, `vector.scale()`) used to fail the whole query on a `NULL` vector, while the other vector functions returned `NULL` (issue #11052). They all return `NULL` now. `vector.l1Norm()` and `vector.lInfNorm()` returned an error on an empty vector whose `vector.l2Norm()` was `0`; the three return `0`, and `vector.hasNaN()`, `vector.hasInf()`, `vector.isNormalized()`, `vector.normalize()`, `vector.clip()` and `vector.toString()` answer an empty vector too.

Impact: a projection over rows without an embedding returns `NULL` for those rows instead of failing. A query that relied on the error to detect a missing or empty vector no longer gets one.

Action: none for most installations. Filter with `WHERE embedding IS NOT NULL` where a missing vector has to be told apart. See NULL and empty vectors.

26.11.1 - A name that is not a database is the client's error, not a server fault

A client that names a database the server does not have (a typo in the connection's database, or a database dropped while a connection still names it) used to be answered as a server fault by every protocol but HTTP and BOLT: SQLSTATE `XX000` for a statement on Postgres (`HV00Q` when the connection opened), status `INTERNAL` for gRPC, its admin calls included, an uncoded "Unknown error" for MongoDB, and the server logged it with a stack trace (issue #10795). It is now answered as the client's to correct: SQLSTATE `3D000` invalid_catalog_name for Postgres, the code PostgreSQL itself gives, status `NOT_FOUND` for gRPC (the Java gRPC client raises a `DatabaseNotFoundException`), error `26 NamespaceNotFound` for MongoDB, `Neo.ClientError.Database.DatabaseNotFound` for BOLT also when the database is lost under a statement, and `ERR Database '<name>' not found` for Redis, the reply its `SELECT` already gave. HTTP keeps its `404`.

Impact: a Postgres client that matched `HV00Q` to detect a missing database now receives `3D000`. A gRPC client that read `INTERNAL` as a server fault now receives `NOT_FOUND`, and does not retry a request that can never succeed.

Action: update any client code that matched the old codes.

26.11.1 - `-Darcadedb.profile=low-ram` keeps its 16MB page cache

The profile given as a `-D` system property or as an environment variable (the way `server.sh`, the Docker image and Kubernetes configure it) set every one of its settings except `arcadedb.maxPageRAM`: the page cache stayed at its computed default, a quarter of the heap (issue #10446). Any other setting the profile lists that also has a computed default, such as `arcadedb.server.httpIoThreads` or `arcadedb.timeSeriesDecodedBlockCacheRAM`, lost its profile value too on a machine where the computed default differs. The first embedded database of a JVM also logged `Configuration profile PARTIALLY applied` for a profile that was set before any database was open.

Impact: a server started with the `low-ram` profile and no explicit `arcadedb.maxPageRAM` now has a page cache of 16MB instead of a quarter of the heap, as the profile always documented. A database larger than 16MB is read from disk more often.

Action: none if the profile was chosen to save memory. If the larger page cache was relied upon, set `arcadedb.maxPageRAM` explicitly, which wins over the profile. See RAM Configuration.

26.11.1 - An update of a record the transaction already updated, refused by a constraint or a listener, makes the commit fail

When a transaction updates a record and a later statement updates it again, a `MAX`, `MIN`, `REGEXP` or other constraint (or a before-update listener that throws) can refuse the second update after its values were set on the copy of the record the transaction writes at commit. That copy kept the refused values and the commit stored them (issue #10482). The statement still fails with the constraint's error, and `commit()` now throws a `TransactionException` ("Transaction cannot be

committed: a statement failed after part of its records were written ..."), like any other statement that fails after writing. A before-update listener that returns `false` to skip the record, which was a silent veto, now makes the commit fail too when the record was already updated by the transaction, since the skipped values would otherwise be committed.

Impact: code that caught the error of such an update, or relied on a listener's veto of an already updated record, and went on to commit, now gets a `TransactionException` at `commit()`. A refused update of a record the transaction had not updated yet is unchanged: nothing of it is left. An openCypher statement whose existence-constraint check takes back every record it created, with the edges it connected to them, no longer makes the commit fail.

Action: roll the transaction back and retry, or avoid refusing an update after the same record was updated in the transaction. See Transactions.

26.11.1 - `DELETE FROM` and `TRUNCATE TYPE` remove the samples of a TIMESERIES type

`DELETE FROM <timeseries type> WHERE <time range> [AND <tag> = <value> ⋯]` and `TRUNCATE TYPE <timeseries type>` now remove samples, from the mutable pages and the sealed blocks of every shard, replicated under HA as a compaction is (issue #10342). Before, #10336 refused them, and before that they reported rows deleted or success and kept every sample. A `WHERE` that is not exactly a time range and tag equalities (a predicate on a `FIELD` column, an `OR`), a `LIMIT`, a `BATCH` and `RETURN BEFORE` are refused. `UPDATE` stays refused. The refusal of `ALTER PROPERTY ⋯ MAX`/`MIN`/`REGEXP`/`MANDATORY` over a stored sample that violates it now names the `DELETE` that removes the sample.

Impact: a delete is not part of the transaction it runs in, like the append of an `INSERT`: the samples are gone when it returns and a `ROLLBACK` does not bring them back. It needs the `deleteRecord` permission on the type. A read that straddles a delete fails with the retryable error a downsample gives (HTTP `503`). A delete does not recompute a continuous aggregate over the type (issue #10428): the buckets it already holds keep the deleted samples until they are rebuilt.

Action: none for installations that did not delete. See Deleting Samples.

26.11.1 - `database.async().appendSamples()` refuses what the other ingestion surfaces refuse

The embedded async API converts every value as `INSERT` does and holds it to the `MANDATORY`, `NOTNULL`, `MIN`, `MAX` and `REGEXP` constraints of the type's columns before it returns, on the calling thread (issue #10381). A value no column can store, or a sample a constraint refuses, raises a `ValidationException` naming the column, and the whole batch is refused. Before, such a sample was stored (and then blocked the same constraint from being declared again), and a value no column could store was lost inside the async worker while `waitCompletion()` returned normally.

Impact: application code that fed `appendSamples()` values a constraint refuses now fails at the call. Valid batches behave as before, including text for a numeric column, which is converted. The public constructor of `DatabaseAsyncAppendSamples` that took the boxed arrays is gone, since it built a grid of its own: code that created the task directly must go through `database.async().appendSamples()`.

Action: handle the `ValidationException`, or fix the values. See Deleting Samples for removing a sample that was stored before.

The form that takes a `TimeSeriesBatch` is held to the same rules (issue #10487): a `TAG` or `STRING` text of

more than 256 UTF-8 bytes is refused on the calling thread with the whole batch, where it used to be lost inside the worker. `TimeSeriesBatch.setValue()` converts as `INSERT` does and raises an `IllegalArgumentException` for a value its column cannot store, where it stored a `Long` past the `INTEGER` range modulo 2^32 and any text of a `BOOLEAN` column as `false`. The typed setters (`setLong()`, `setString()`, …) are unchanged.

26.11.1 - SQL `split()` keeps the empty pieces at the end of the list

SQL `split()`, the `split()` function and `text.split()` answer the same list as the openCypher `split()` (issue #10485): `'a,'.split(',')` is `['a', '']` and `',','.split(',')` is `['', '']`, where the empty pieces at the end were dropped (`['a']` and `[]`). An empty delimiter splits a string into its characters, a character outside the Basic Multilingual Plane (an emoji) being one, and `replace()` with an empty search puts the replacement between characters and never inside one.

Impact: a query that counted the pieces of a text ending with the delimiter, or compared the last one, sees one more piece.

Since 26.12.1 `text.split()` is no longer part of this rule: it is APOC's split around a regular expression (see the entry on the `text.*` functions above), and it drops the empty pieces at the end again unless its `limit` is negative.

Action: remove the empty last piece in the query if it is not wanted, for example by trimming the delimiter from the end of the text before splitting it.

26.11.1 - The JSON of a record writes a `DATETIME_MICROS` and a `DATETIME_NANOS` property as text

`record.toJSON()` wrote a `DATETIME_SECOND`, `DATETIME_MICROS` or `DATETIME_NANOS` property as a count of seconds, microseconds or nanoseconds, which `fromJSON()` and the server read as milliseconds (issue #10486). It now writes epoch milliseconds for a `DATETIME_SECOND` property, as for `DATETIME`, and the text with all its digits (for example `"2020-01-02 03:04:05.123456789"`) for `DATETIME_MICROS` and `DATETIME_NANOS`, which milliseconds would truncate. The remote client sends a record as this JSON, so it now stores those properties right on the server.

Impact: a reader of `toJSON()` (the change events of a WebSocket subscription carry it) that took those numbers as seconds, microseconds or nanoseconds gets milliseconds, or text, instead.

Action: read a number as epoch milliseconds, and parse the text of a `DATETIME_MICROS` or `DATETIME_NANOS` property.

26.11.1 - A declared `LIST OF` / `MAP OF` property survives a JSONL export and import

`EXPORT DATABASE` and `IMPORT DATABASE` in the `jsonl` format now round-trip a declared `LIST OF` or `MAP OF` property of any element type (issue #10391). Before, a declared `LIST OF DATE`, `MAP OF DATE`, `LIST OF BINARY`, `MAP OF BINARY`, `MAP OF DOUBLE` or `MAP OF FLOAT` holding a non-finite number aborted the import of the whole database, and a `MAP OF DECIMAL` lost the scale of its values.

Impact: the export of such a property carries the type tree (`y`) it did not carry before, so the file is slightly larger. Files exported by earlier versions import where their data allows (see the JSONL round trip note).

Action: none.

26.11.1 - gRPC `BeginTransaction` refuses `SERIALIZABLE`

A gRPC `BeginTransaction` that asks for `SERIALIZABLE` isolation is now refused with `INVALID_ARGUMENT`, as is an isolation value the server does not know, a newer client's enum constant (issue #10386). Before, the server began the transaction at `REPEATABLE_READ` (or, for an unknown value, at `READ_COMMITTED`) and answered success, so a caller that needed write-skew prevention was told its transaction started and got a weaker guarantee. HTTP, SQL, openCypher and PostgreSQL already refused a level the engine cannot provide, and SQL `BEGIN ISOLATION` and openCypher `START TRANSACTION ISOLATION` now say so in the same words as HTTP: `Invalid isolationLevel 'SERIALIZABLE': accepted values are [READ_COMMITTED, REPEATABLE_READ]` (SQL raised the JVM's `No enum constant …`).

Impact: a hand-written gRPC client that sends `SERIALIZABLE` now fails at `BeginTransaction`. The first-party `RemoteGrpcDatabase` never sent it. `READ_UNCOMMITTED`, the proto default, still begins, at the stronger `READ_COMMITTED`, and the response reports it in `effective_isolation`. The server no longer logs a throttled warning for `SERIALIZABLE`, because there is nothing left to warn about.

Action: ask for `REPEATABLE_READ`, the strongest level the engine provides. See Transactions.

26.11.1 - gRPC sends the records of a traversal, a list or a map as projected records

A projected value that holds records - `SELECT out() AS n` and the other graph traversal functions (`in()`, `outE()`, `inE()`, `both()`, `bothE()`), a list, a map, an array or a nested result row - is now encoded like a top-level projected record, following the request's `ProjectionSettings` (issues #11067, #11044, #11048). Before, `out()`, `in()`, `outE()` and `inE()` reached a gRPC client as a Java class name string (`com.arcadedb.graph.GraphEngine$10@63e44da5`), and the records inside a list (`both()`) as bare RIDs, while HTTP returned the records.

Impact: such a column is now a `list_value` (empty for a vertex with no edges) whose elements are the records, in JSON by default, instead of a string or a list of `link_value`. `soft_limit_bytes` now also counts the records nested inside lists and maps.

Action: a client that wants RIDs only sets `projection_encoding` to `PROJECTION_AS_LINK`. See Projected records.

26.11.1 - gRPC search hits send list and set properties as lists

A list, set or array property of a `VectorSearch`, `HybridSearch` or `FullTextSearch` hit is now a `list_value`, as the same property is in an `ExecuteQuery` result (issue #11207). Before, it reached the client as a `string_value` holding its JSON text. A set value in the time series RPCs was sent as text in the same way.

Impact: a client that parsed that string now receives a list.

Action: read the property as a list. See Vector, hybrid, and full-text search.

26.11.1 - gRPC `grpc.errors.total` counts the failures a response reports with `success = false`

A call closed with status `OK` whose response declares a top-level boolean `success` field set to `false`, such as an `UpdateRecord` or `DeleteRecord` a listener vetoed, is now counted in `grpc.errors.total`, with `status OK` (issue #11054). Before, only `ExecuteCommandResponse` was checked.

Impact: the error rate of the gRPC dashboards rises by the in-band failures it did not count.

Action: none. See gRPC metrics.

26.11.1 - A database an `overwrite` restore replaces refuses writes until the restore ends

While `restore backup … overwrite` downloads and extracts the backup, the database it replaces stays open for reads and refuses every write, on every protocol, including connections and transactions opened before the restore started (issue #10689). The write is answered as "not available": `57P03` for Postgres, `DatabaseUnavailable` for BOLT, `UNAVAILABLE` for gRPC. A database setting change is refused the same way. The restore first waits for the commits already running. Before, such writes were acknowledged and then lost when the restored copy replaced the database.

Impact: an application writing to a database while it is restored now gets an error instead of a success that is silently lost. Reads are unchanged until the swap. When the restore fails, writes are accepted again.

Action: none for most installations. Retry the writes after the restore ends.

On an HA cluster the same holds for writes made through the other servers (issues #10965, #10974): while the leader restores a database, a commit on a replica is refused with a retryable error, and the commits the leader had already accepted from replicas are completed before the download starts (at most 10 seconds, otherwise the restore is refused and can be retried). A `restore database` or `restore backup` is refused, retryable, when the server running it is not the leader, or when the leadership moves before the restored copy replaces the database: a new leader may have accepted writes meanwhile, and the restore would destroy them. Requests sent to a follower are forwarded to the leader, as before.

26.11.1 - BOLT: a message sent out of turn fails the connection until `RESET`

A BOLT connection that receives a message its current state does not allow, for example a second `HELLO` on an authenticated connection, now answers with a protocol error and moves to the failed state, as it already did for `RUN`, `PULL` or `COMMIT` sent out of turn (issue #10771). Until the client sends `RESET`, every request is answered `IGNORED`, `HELLO` and `LOGON` included. Before, `HELLO` and `LOGON` answered the error but left the connection serving, and answered an error instead of `IGNORED` on a failed connection.

Impact: none for the Neo4j drivers, which never send these messages out of turn. A hand-written client that did must send `RESET` after the error.

Action: none for most installations.

26.11.1 - Postgres, Bolt, gRPC, Redis, MongoDB, MCP and Gremlin clients no longer open a database a restore holds or `close database` closed

These protocols resolved the database a client names through the server's own entry point, which sees past the marks an operator puts on a name, so a connection could open the copy an `overwrite` restore was about to replace, or a database closed for a cold backup (issue #10533). They now resolve it through the entry point HTTP uses, and the client is told that the database is not available.

Impact: a connection that names such a database while the restore runs, or after `close database` and before `open database`, is refused instead of opening it. A database that is already open is served for reads as before and refuses writes while the restore runs (see the note above), and a restore that targets a name nobody has open no longer loses the writes a client made to the copy it replaces.

Action: none for most installations. Retry the connection once the restore ends, or run `open database` after the cold backup.

26.11.1 - Database listings include the databases that are on disk but not open

The database listings and existence checks now include a database that is on disk but not open: one not opened at startup because `arcadedb.server.databaseLoadAtStartup` is `false`, and one closed by the new idle close (issue #11068, see Closing idle databases), because every protocol opens it on its first request. This covers HTTP `GET /api/v1/databases` and `GET /api/v1/exists/{database}`, the `list databases` server command, the gRPC admin `ListDatabases`, `ExistsDatabase`, `GetDatabaseInfo`, `CreateDatabase` (`if_not_exists`) and `DropDatabase` (`if_exists`), Bolt `SHOW DATABASES` and its default-database fallback, the MCP `list_databases` tool and schema resources, Gremlin traversal sources and the `drop database` command. A database closed with `close database`, being restored or being dropped is still not listed.

Impact: with `databaseLoadAtStartup=false`, a database nobody has opened yet is listed and reported as existing where it used to be missing. Nothing changes with the default `databaseLoadAtStartup=true` and no idle close configured.

Action: none. A tool that read the listing as "the databases open now" should not rely on it: the listing is the databases a client can use.

26.11.1 - Studio's result table has no PDF export button

The **PDF** button of the result table (see Table view) is removed, together with the PDF library behind it, which depended on unmaintained packages and was about 2 MB of the Studio download. **Copy**, **Excel**, **CSV** and **Print** are unchanged.

Action: to save a result as PDF, use **Print** and choose the browser's "Save as PDF".

26.11.1 - Studio's GraphML export writes the properties of the records

Studio's graph **Export** > **GraphML** is now written by Studio itself instead of the unmaintained `cytoscape-graphml` plugin, in the layout of the GraphML database export (see Graph view). The plugin declared no `<key>` elements and wrote the properties of every record as the single text `[object Object]`, so the file carried none of them. Now the type is under `labelV` / `labelE`, every property is under a typed key, and maps and lists are JSON text marked with `arcadedb:type`. The

values the canvas computed to draw an element (`label`, `fullLabel`, `size`, `weight`) are no longer written.

Impact: a script that read the old file by its `label`, `size` or `weight` data, or by the `type` data that held the element's type, finds them gone: the type is under `labelV` / `labelE`, the record's own properties under their names.

Action: none for most users. Re-export a graph saved with an older Studio to get its properties.

26.11.1 - A plugin route that another plugin or the server already holds fails the server start

The routes a plugin declares through `registerAPI(HttpServer, PluginRoutes)` are now one namespace shared by every plugin (issue #10385). Declaring another method on a path a different plugin declared merges into the same entry. Declaring a method a plugin already declared on that path, a path that fills a route the server registers for itself (for example `GET /api/v1/server` or `/api/v1/query/mydb`), or a prefix equal to one of the server's own mounts (`/api/v1`, `/api/v1/ai`, `/ws`), is refused with an `IllegalStateException` naming both parties, and the server does not start. So is an exact path on the path of a prefix another party holds, and a prefix on the path of an exact path another party holds (issue #10532): an exact path is matched before any prefix, so one of the two would answer everything under the path but not the path itself. One plugin may declare both on one path, to handle the root in one handler and everything below it in another. Before, the second registration silently replaced the first plugin's route, which then answered `405` naming the other plugin's method, a plugin could shadow a core route, and a plugin declaring the `/api/v1` prefix replaced the whole core router, with no log line. The same check applies to a plugin still implementing the deprecated `registerAPI(HttpServer, PathHandler)`.

Impact: a third-party plugin that relied on replacing another route no longer starts the server. The plugins shipped with ArcadeDB are unaffected.

Action: give the route another path, or remove the plugin that also declares it. A route declared `GET` now also answers `HEAD` with the same handler, as the core health routes do, so container healthchecks such as `wget --spider` work against it (issue #10384). Every core `GET` route does the same, and a `405` from a core route names the methods the route answers in an `Allow` header, as a plugin route does (issue #10531).

26.11.1 - A taken schema object name is a conflict, not a server error

Creating, or renaming onto, a name already taken by a type, property, index, constraint, bucket, trigger, view or function now raises `com.arcadedb.exception.AlreadyExistsException`, a `SchemaException` (issue #10153). Before, the SQL statements raised `CommandExecutionException`, and defining a function or a library twice, or renaming or copying a type onto a taken name, raised `IllegalArgumentException`.

Impact: over HTTP these answer `409` instead of `500` (or `400` for a rename); over PostgreSQL `42P07`, `42701` or `42710` instead of `XX000` or `42P01`; over Bolt a `Neo.ClientError.Schema.*AlreadyExists` code; over gRPC `ALREADY_EXISTS`. Embedded or remote Java code that caught `IllegalArgumentException` or `CommandExecutionException` for this case no longer catches it.

Action: catch `AlreadyExistsException` (or `SchemaException`) where you handle a name that is already taken. Clients that keyed on the old status codes should expect the new ones.

26.11.1 - `REFRESH INCREMENTAL` views track every type their query reads

An `INCREMENTAL` materialized view now refreshes on changes to every type its query names, at any depth (subqueries in `FROM`, `WHERE`, projection, `LET`), and to all their subtypes (issue #10205). Before, only the first `FROM` type was tracked, without its subtypes, and the view went stale while staying `VALID`. `REFRESH INCREMENTAL` is refused for a query that reads a `TRAVERSE`, a `MATCH`, a function, a bucket, an index, `schema:` metadata, record ids or parameters in `FROM`.

Impact: existing views have their source list completed from their query on open, so they refresh more often, and `DROP TYPE` of any type they read is refused while the view exists. Creating such a view, or switching one to `INCREMENTAL`, with an untrackable source fails.

Action: use `REFRESH MANUAL` or `REFRESH EVERY` for a view over an untrackable source. See INCREMENTAL.

26.11.1 - `CREATE PROPERTY ... STRING` over stored non-string values is refused

`CREATE PROPERTY T.n STRING` now fails when records of `T` or its subtypes already hold `n` as a number, a date, a boolean or another non-string value, as `CREATE PROPERTY ... INTEGER` over a stored `'abc'` always did (issue #10204). Before, it was accepted without converting the values, and an index on `n` compared them as text while a scan compared them natively.

Impact: a schema script that declares such a property fails, naming the first offending record.

Action: convert the values first, for example `UPDATE T SET n = n.asString()`, then create the property. See `CREATE PROPERTY`.

26.11.1 - SQL `replace()` and `length()` on `NULL` return `NULL`

`s.replace('a', 'b')` on a `NULL` `s` returns `NULL` (it returned `'a'`, its first argument), and so does `replace()` with a `NULL` argument. `s.length()` on `NULL` returns `NULL` (it returned 0), as SQL's `REPLACE()` and `LENGTH()` do (issue #10193). `''.length()` is still 0.

Impact: `WHERE s.length() = 0` no longer matches records where `s` is `NULL`.

Action: add `s IS NULL OR ...` where `NULL` values must keep matching. See length() and replace().

26.11.1 - JSONL import clears links to records outside the export, and restores value types and bucket counts

A link whose target is not in a `jsonl` export (deleted before the export, or excluded by `includeTypes`, `excludeTypes` or `includeRecords`) is set to `null` on import, or dropped from a `LIST OF` / `MAP OF`, and counted as `danglingLinks` in the import result (issue #10201). Before, it kept its old RID, which after the import named another record. Values with no declaration now keep their type (`DECIMAL` scale, `LONG`, `SHORT`, `BYTE`, `FLOAT`, full date precision, `BINARY`, arrays, links), and each type is recreated with its exported bucket count.

Impact: a record whose `MANDATORY` or `NOT NULL` link points outside the export fails to import like any other record error. The export is format version 2 (`exporterVersion` 2); files exported by older versions import as before.

Action: none for most installations. Check `danglingLinks` in the import result. See the JSONL round trip note.

26.11.1 - GraphSON and GraphML exports write every value type, and Gremlin accepts primitive arrays

A `graphson` export no longer fails for the whole database on a `BYTE`, `SHORT`, `DECIMAL`, date or `BINARY` property: it writes TinkerPop's `gx:` types and ArcadeDB's own `arcadedb:` types (issue #10202). A `graphml` export declares a key with a GraphML type only when all its values share it, and writes the other values as text with an `arcadedb:type` attribute that ArcadeDB's GraphML import reads back. The Gremlin API now accepts `byte[]`, `short[]`, `int[]`, `long[]`, `float[]` and `double[]` property values and stores them natively; the graph features report `Integer`, `Long`, `Float` and `Double` array values as supported. String and boolean arrays are still refused.

Impact: a GraphSON file with `gx:` types needs TinkerPop's `GraphSONXModuleV3` to be read outside ArcadeDB, and the `arcadedb:` types are readable by ArcadeDB only. Links are written as plain RID text in both formats.

Action: none for ArcadeDB to ArcadeDB transfers. See the graph formats note.

26.11.1 - HTTP: committing a session the server does not hold answers `404`, not `204`

`POST /api/v1/commit/{database}` with an `arcadedb-session-id` the server does not hold for the caller (never issued, already rolled back, ended by `terminate transaction`, expired, or owned by another user) answered `204`, the answer of a real commit, although nothing was committed. It now answers `404` "Remote transaction session not found or expired" (issues #10164, #10172). The idempotent retry of a commit that did succeed still answers `204`: the server remembers committed sessions for the session timeout, at most 10,000 of them, per server and in memory.

Impact: a client that treated `204` on commit as "committed" for a stale or foreign session now gets `404`. A commit retry sent to another cluster node, or after a server restart, gets `404` even when the first commit succeeded, and so does a retry that arrives while the first commit of the same session is still running. `POST /api/v1/rollback` still answers `204` for a stale session id.

Action: handle `404` on commit. See Commit a transaction.

26.11.1 - HTTP: wrong Basic credentials answer `401`, not `403`

A request with Basic credentials that do not authenticate (wrong password, unknown user, or a user locked out after too many failed attempts) answered `403` "Security error". It now answers `401` "Authentication failed", as the bearer-token and API-token paths already did, and so do a malformed Basic header and an unsupported `Authorization` scheme (issues #10109, #10180). No `WWW-Authenticate` header is sent, so browsers do not open their native login dialog over Studio's login form. The response body still names `ServerSecurityException`, so the Java client raises the same `SecurityException` as before.

Impact: a client that matched on `403` to detect a failed login, including a lockout, no longer sees it. `403` remains the answer for an authenticated user who lacks permission.

Action: match on `401` for a failed login.

26.11.1 - HTTP: plugin routes answer only the method they declare

The HTTP routes of the plugins shipped with the server (the HA cluster endpoints, MCP and metrics) ran their handler for any HTTP method, so `GET /api/v1/cluster/stepdown` stepped the leader down like the documented `POST`. Every shipped plugin route now declares its method, and any other method answers `405` with an `Allow` header before the handler runs, and before authentication (issues #10157, #10297). MCP keeps its JSON-RPC error body on a `405`. `HEAD` on a GET-only plugin route answers `405` too.

Plugins declare their routes through `ServerPlugin.registerAPI(HttpServer, PluginRoutes)`, which takes each route with its HTTP methods (issues #10296, #10311). The old `registerAPI(HttpServer, PathHandler)` is deprecated for removal: a plugin that still implements it keeps working and still answers every method on its routes, and the server logs a WARNING at startup that names the plugin. A plugin should implement only one of the two overloads. If it implements both, both are called with a separate WARNING, and a path registered in both is registered twice, the later raw registration winning.

Impact: a client that called a plugin endpoint with an undocumented method now gets `405`.

Action: use the documented method of each endpoint, and move third-party plugins to the `PluginRoutes` overload.

26.11.1 - SQL `INSERT` into a time series refuses a number for a BOOLEAN or STRING column

SQL `INSERT INTO` a TIMESERIES type converted each value to the column's type before the time series check ran, so `bo = 7` was stored as `true` in a BOOLEAN column and `s = 42` as `"42"` in a STRING column. SQL now refuses it, through `SET`, `CONTENT` and `VALUES`, with a `CommandExecutionException` naming the column, as the line protocol, gRPC and Prometheus remote write already did. An out-of-range number in a non-temporal column (for example 300 into a BYTE column) fails with the same exception instead of a bare `IllegalArgumentException` (issues #10137, #10236).

Impact: an insert that relied on a number being coerced into a BOOLEAN or STRING time series column now fails. TIMESTAMP and DATE/DATETIME columns are unchanged: SQL still reads a number there as epoch milliseconds.

Action: send a value of the column's type.

26.11.1 - MongoDB: writes on a time-series collection are refused unless they can be applied exactly

A filtered update or delete, and the existence check of an upsert, on a TIMESERIES type matched nothing and reported success. `deleteMany` now deletes the samples when its filter is an AND of conditions the time-series engine applies exactly: `$eq`, `$gt`, `$gte`, `$lt` and `$lte` on the timestamp (whole milliseconds, or a date), and equality on a TAG field with a value of the tag's own kind. An unfiltered `deleteMany` deletes every sample. Any other filter (a FIELD, `$or`, `$ne`, `$in`, an unknown field, a tag value of another kind) is refused with error 20 (`IllegalOperation`) naming the type. `deleteOne`, and every update, replace and upsert, are refused the same way. Every entry of a multi-entry delete command is checked before any runs (issues #10464, #10477).

Impact: a MongoDB client that updated, upserted or deleted one document in a time-series

collection and got a success that changed nothing now gets an error. As in SQL, the samples a delete removes are not part of the command's transaction: they are gone when it returns. `insertOne` into a time-series collection fails with a misleading `NamespaceNotFound` (issue #10476).

Action: see Query languages.

26.11.1 - Java API: a failed `transaction()` that joined the caller's transaction leaves it open

When the block passed to `database.transaction(…)` joined a transaction the caller had already begun, and threw anything other than a retryable exception, the call rolled back the caller's whole transaction, and the caller's next write failed with "Transaction not begun". `transaction()` now rolls back only the levels it opened itself, embedded and through the remote drivers (HTTP and gRPC) (issues #10135, #10241). A nested transaction the block began and left open no longer survives the failure, and over the remote drivers an `Error` thrown by a block also releases the server-side transaction the call owns.

Impact: code that relied on a failed joined `transaction()` rolling back the outer transaction now finds it still active.

Action: commit or roll back the outer transaction yourself.

26.11.1 - A UNIQUE key moved off its holder can be given to only one record

When a transaction moved a UNIQUE key away from the record that held it, and then gave the key to two other records, the commit succeeded with only one of the two in the index, a duplicate the index could no longer see. The in-transaction duplicate check now treats a key that was handed over as taken, on UNIQUE and UNIQUE_HASH indexes alike, so the second claim fails with `DuplicatedKeyException` (issues #10265, #10298). Three related defects are fixed with it: inside the transaction, a lookup on a UNIQUE key that one record gave up no longer hides the records still holding it; committing such a transaction on a UNIQUE_HASH index no longer deletes the committed holder's index entry; and a valid replacement whose removals sit in several buckets is no longer refused at commit.

Impact: a transaction that gives one UNIQUE key to two records, directly or through a SQL script, now fails with `DuplicatedKeyException` instead of committing a corrupt index.

Action: none, other than fixing the offending transaction.

26.11.1 - Remote drivers: deleting a missing or vetoed record throws, and `LIST OF DATE` reads as `LocalDate`

A remote delete that deleted nothing could look like a success. Both drivers now fail the way the embedded engine does for a missing record: a missing record throws `RecordNotFoundException` (HTTP now matches gRPC and embedded), and a vetoed delete throws `RecordVetoedException`, as a vetoed remote `save()` already does. The embedded `deleteRecord` still stays silent on a veto. `RemoteGrpcDatabase.deleteRecord(String rid, long timeoutMs)` returns `false` for a vetoed delete. `delete()` on a remote vertex or edge goes through the driver instead of sending its own SQL, and `delete()` on a remote immutable document no longer throws `NullPointerException`. The `deleteRecord` statistic counts a delete only after the server confirms it (issues #10230, #10301, #10141, #10231).

The remote client reads `LIST OF DATE` items as `LocalDate` instead of Strings, in projections and in whole-record reads; `java.util.Date` items stay unhinted (issues #10909, #10995).

Impact: code that deletes a record over HTTP that may already be gone, and relied on the call being silently idempotent, must catch `RecordNotFoundException`. Code that expected a vetoed remote delete to return normally must catch `RecordVetoedException`. An older gRPC client talking to a new server gets `DatabaseOperationException` for a vetoed delete.

Action: catch the two exceptions where deletes may race. See Events.

26.11.1 - A remote `LocalDate` query parameter is sent as a typed marker

The remote Java client sent a `LocalDate` parameter as the epoch milliseconds of its UTC midnight, which a comparison reads as epoch days, so `WHERE fecha = ?` matched nothing remotely. It now sends `{"$date": "2024-01-15"}`, which the server decodes to a `LocalDate`, so `=`, `IN` and `>` match a DATE. A malformed or non-string `$date` payload is refused before execution (issues #10908, #10997).

Impact: there is no version negotiation: a server older than this change receives a map for a `LocalDate` parameter. `LocalDateTime` and `java.util.Date` parameters keep their epoch-millis form. The gRPC and Python clients were not examined.

Action: upgrade client and server together.

26.11.1 - A number written into a DATE property is a day count under every `dateImplementation`

Under `arcadedb.dateImplementation=java.util.Date` (also `Calendar` and `LocalDateTime`) a number written into a DATE property was read as epoch milliseconds, while the default `LocalDate` reads days, so the days the remote client sends were stored as 1970-01-01. A number, and a numeric string, written into a DATE property, or into the items of a `LIST OF DATE` or `MAP OF DATE`, is now a day count for every implementation, with the midnight-millis reading kept (a non-zero exact multiple of 86,400,000). The day-to-millis multiplication is overflow-guarded and fails with an `IllegalArgumentException`. A number that is no day count for a `LocalDate`, such as a "now" in millis, is still read as millis (issues #10910, #10996).

Impact: a caller that wrote a small millis value (below about 1.07e11 and not a midnight) into a DATE under `java.util.Date`, `Calendar` or `LocalDateTime` now gets that many days, as `LocalDate` always did. A numeric string wins over a configured `dateFormat`.

Action: write dates as dates, or as day counts. See Managing dates.

26.11.1 - An update of a record with a property that cannot be decoded is refused

An update of a record one of whose properties cannot be decoded used to save the record back without that property, because the lenient read logs `Skipping corrupted property` and returns the rest, which turned a possibly recoverable read problem into permanent data loss. The update (`MutableDocument.save()`, SQL `UPDATE`, graph vertex and edge saves, async updates) is now refused with a `SerializationException` naming the record and the property, and the stored bytes are left untouched (issues #10955, #10983). Reads stay lenient. An explicit full overwrite (`fromMap`) is still allowed.

Impact: an update of a damaged record now fails instead of silently dropping the damaged property.

Action: investigate the record with `CHECK DATABASE`, and restore it from a backup if needed.

26.11.1 - Docker: every image runs as UID 1000, and the volumes changed

The JVM image and both native images (amd64 and arm64) run as `1000:1000`. The amd64 native image ran as root and the arm64 one as UID 65532, so a volume written by one image could not be opened by another (issues #10319, #10331). The image a custom distribution generates with `package/arcadedb-builder.sh` ran as `100:101`, and now runs as `1000:1000` too (issues #10330, #10351). The generated image now declares the same five volumes as the official images (`config`, `databases`, `replication`, `backups`, `log`) instead of three (issues #10387, #10401), and the `docker-compose.yml` quickstart mounts a named volume on all five instead of only `databases` (issues #10402, #10453).

Impact: a volume or bind mount written by an earlier native image is owned by root or 65532, and one written by an earlier custom image by `100:101`, so the new image cannot write to it. A `config` volume of a custom image no longer picks up default files that a later image adds, and a bind mount on `config` hides the image's defaults. Users, groups and API tokens created with the earlier compose file are in an anonymous volume that the new file does not mount.

Action: before upgrading, run `chown -R 1000:1000 <volume or bind-mount path>` once on each volume. To keep the users, groups and tokens of an old compose stack, copy `/home/arcadedb/config` out of the old container and into the new one. See Image user and volumes.

26.11.1 - Kubernetes: the plain StatefulSet example is removed, use the Helm chart

The distribution and the Docker images no longer ship `config/arcadedb-statefulset.yaml`. The Helm chart is the only supported Kubernetes deployment (issue #10542). The server-side fix that lets a node started with only `arcadedb.server.rootPasswordPath` form an HA cluster is kept.

Impact: a deployment applied from `arcadedb-statefulset.yaml` keeps running, but the file no longer comes with new releases.

Action: move to the chart, see Kubernetes. `helm template` renders plain manifests from it.

26.11.1 - SQL `shortestPath()` `maxDepth` limits the edges, not the vertices

`shortestPath(…, {maxDepth: n})` counted vertices, so a path of exactly `n` edges was not found and `maxDepth: 1` could not find even a direct edge. It now limits the edges on the path, as documented (issues #10439, #10445). Cypher `shortestPath` with `*..n` returns the same results as before.

Impact: a query that set `maxDepth` one higher to work around this now allows one more hop and can return a longer path.

Action: lower `maxDepth` by one. See shortestPath().

26.11.1 - A pattern on a time-series type fails in Cypher, Gremlin and GraphQL

A Cypher pattern whose node label or relationship type names a TIMESERIES type returned zero rows. It now fails with a `CommandSemanticException` that names the type and points to SQL or

PromQL, for every pattern of the statement, including `OPTIONAL MATCH`, `MERGE`, pattern predicates, `CALL`/`EXISTS`/`COUNT` subqueries and `UNION` branches (issues #10440, #10443). A Gremlin traversal fails with an error naming the type when a `has` step asks for a time-series label, or a hop names one as its edge label, including inside child traversals; a label under an odd number of nested `not()` steps is still answered. A GraphQL `@relationship` directive that names a time-series type, which returned an empty list, now fails (issues #10442, #10468). MongoDB QL now returns the samples of a time-series collection instead of nothing.

Impact: a query that names a time-series type fails even when no row would reach that pattern.

Action: query time series with SQL or PromQL.

26.11.1 - SQL: on an edge, `out` and `in` read the endpoints

On an edge, SQL reads the endpoints as `@out` and `@in`. The plain names `out` and `in` answered `null` (`SELECT out.id FROM E, WHERE in.name = 'x', ORDER BY out`). On an edge with no property of that name they now resolve to the edge's endpoints, like `@out` and `@in`. A property named `out` or `in` on the edge record (even with a `null` value), or declared under that name by the edge's type, still wins. Vertices, documents and projection rows are unchanged (issues #10684, #10806).

Impact: `SELECT FROM E WHERE out IS NULL` used to match every edge without an `out` property and now matches none.

Action: prefer `@out` and `@in`: `WHERE @out = #rid` can use the index or the adjacency shortcut, while `WHERE out = #rid` answers correctly but through a scan. *Since v26.12.1* `WHERE out = #rid` takes the adjacency shortcut too, as long as no record or type of the database uses a property named `out` or `in` (issue #10807).

26.11.1 - Number arguments of SQL and Cypher-style functions are no longer truncated or wrapped

`if`, `randomInt` and `vector.sparseCreate` read a numeric argument through a cast to int, which dropped the fraction and wrapped a value past the int range. `if(0.5, a, b)` and a condition of `3000000000` now return `a`, and `-3000000000` returns `b`. `randomInt` and `vector.sparseCreate` refuse a fractional or out-of-range bound, dimension or index with an error naming the function (issues #10699, #10851). The Cypher-style functions (text, collection, aggregation, path, temporal and others) refuse a fractional or out-of-range INTEGER argument with a `CommandSemanticException` naming the function and the argument, and keep a value past the int range as a long where the function can still answer (`left('abc', 3000000000)` returns `'abc'`). The same check covers the SQL `duration`, `date` and `promql` amounts, `ts.timeBucket`, `substring()`, `util.sleep` and `date.*`. A TEXT argument such as `'2.5'` is refused the same way, with `<fn>() requires a whole number <arg>, but received '2.5'`, instead of a raw `NumberFormatException` (issues #10850, #10953, #10954, #10981).

Impact: an `if` whose condition is a fraction between -1 and 1, or a number past the int range, can take the other branch. A query that passed `2.5`, or a number past the int range, to one of these functions ran on the truncated or wrapped value and now fails.

Action: pass whole numbers.

26.11.1 - Redis: a malformed message gets one error and a close, and inline commands are answered

Every malformed message gets one `-ERR Protocol error` reply and one WARNING, and the connection is closed. A command that does not start with a RESP type marker is read as an inline command, so `PING` from `nc`, `telnet` or a health probe is answered (issues #10597, #10846). An array whose first element is not a string is answered with an error and the connection stays open (issues #10845, #10963).

Impact: a top-level simple string, integer or bulk string (`+`, `:` or `$`) is refused and the connection is closed, which is stricter than Redis.

Action: none for standard clients. See Malformed messages and inline commands.

26.11.1 - HA: a node refuses to start when the root password behind its cluster token was rewritten

With no explicit cluster token, HA derives the token from `arcadedb.ha.clusterName` and the root password at every start. Rewriting the password file (`arcadedb.server.rootPasswordPath`) in place, as rotating a Kubernetes Secret does, or changing the inline `arcadedb.server.rootPassword`, gave each node a new token at its next restart, and during a rolling restart the nodes on the new token and those on the old one refused each other's requests with `401`. A node whose token is derived from the password now records a fingerprint of it in `<raft-storage>.cluster-token-fingerprint` and refuses to start when a later start derives a different token, unless `arcadedb.ha.clusterTokenAllowRotation=true` (default false) (issues #10601, #10826, #10924, #11232). The startup WARNING now describes how the token is derived, and a refused cluster token answers `401` "Invalid cluster token" with a WARNING on the refusing node.

Impact: the root password can no longer be rotated with a rolling restart when the token is derived from it. `<raft-storage>.cluster-token-fingerprint` must be writable, otherwise startup fails with a `ConfigurationException`. A node already running on the inline source has no record yet, so a rotation made before the upgrade cannot be detected at its first start. A new node, or one without persisted Raft storage, has nothing to compare against, so a differing token is still refused per request.

Action: either restart every node together with `arcadedb.ha.clusterTokenAllowRotation=true` and then remove the setting, or switch to an explicit `arcadedb.ha.clusterTokenPath`. See HA security.

26.11.1 - openCypher: an explicit write of a lookup key makes the query eager

A `MATCH` filter now sees the graph as it was before a later write of the property it filters on, as the language requires. A `SET`, `REMOVE`, `SET +=` with a map literal, `MERGE … ON CREATE/ON MATCH SET`, `FOREACH` or writing subquery that names the property read by an earlier `MATCH` pattern or `WHERE` plants an eager barrier before the write (issues #11173, #11228). An opaque map write that can rewrite a lookup key plants it too (issues #11230, #11261): `MATCH (n {id: row.id}) SET n += <anything but row>` holds its rows in memory, as Neo4j does. A map taken from the same variable the key is read from keeps streaming.

Impact: a query that read its own later write returns a different result, and `UNWIND … MATCH (n {id: row.id}) SET n.id = …` holds its input in memory.

Action: none for most queries. See Reads and writes in the same query.

26.11.1 - HA: a database whose crash recovery did not complete is quarantined

After a power loss in production mode, crash recovery could abort on a WAL gap and the database opened anyway, serving wrong property values and missing index data. Such a database is now quarantined under the cause `LOCAL_RECOVERY_INCOMPLETE`, and a database that opens with an unavailable unique index under `UNAVAILABLE_UNIQUE_INDEX`. The node refuses readiness, hands leadership off and resyncs the database from the leader (issues #11250, #11259, #11264).

Impact: a sole voter has no leader to resync from and only logs a SEVERE message. Known limits are listed in After a power loss.

Action: none. Watch `/api/v1/ready` and the `local-resync-in-progress` alert after a power loss.

26.11.1 - A statement that fails inside a transaction is taken back, `UPDATE … SET` reads the row as it was, and an explicit `NULL` stays `NULL`

A multi-row `INSERT`, `UPDATE`, `DELETE`, `CREATE EDGE`, openCypher or `sqlscript` statement that failed after writing part of its records used to leave them in the caller's transaction, and the commit published half a statement. The engine now takes the statement back on its own, with its records, index entries, edges and counters, and the transaction goes on (issues #10346, #10194, #10379). When it cannot take the statement back exactly (more than `arcadedb.txStatementSavepointMaxEntries` entries, 100,000 by default, a statement that touched a Graph Analytical View, dropped a file or ran Gremlin) the commit is refused with a `TransactionException` that names the reason.

Two SQL write semantics changed with it (issue #10194): every right-hand side of `UPDATE … SET` reads the row as it was before the statement, so `SET a = b, b = a` swaps the two values, and an explicit `NULL` is stored as `NULL`: a `DEFAULT` fills only a property that is left out of the statement.

Impact: code that caught the error of a multi-row statement and relied on the rows written before the error now finds none of them. A `NOTNULL` property with a default refuses an explicit `NULL`. A `SET` list that relied on a later item reading the new value of an earlier one gives a different result.

Action: write the dependent assignment out (`SET b = 7, a = 7`), and leave a property out of the `INSERT` to get its default. See Transactions.

26.11.1 - String functions count characters, not UTF-16 code units

SQL `length()`, `left()`, `right()`, `subString()`, `indexOf()`, `lastIndexOf()`, `charAt()` and `size()`, and `text.charAt()`, `text.indexOf()`, `text.lpad()` and `text.rpad()`, counted and cut UTF-16 code units, so a character outside the Basic Multilingual Plane (an emoji) counted as 2 and could be split into a lone surrogate. They now count characters, as openCypher already did (issues #10248, #10378, #10523, #10524).

Impact: the result of these functions on a string that holds such characters changes (`'a'.length()` was 3 and is 2). Strings that stay in the Basic Multilingual Plane are not affected.

Action: none, unless an index or an offset was computed from the old counts.

26.11.1 - Cypher temporal values compare, add and construct as in Neo4j

Two durations are no longer comparable with `<` and `>` (the answer is `null`), a `localdatetime()` no longer equals a `datetime()` at the same wall-clock time, `ORDER BY` over several temporal types orders by type instead of throwing, duration arithmetic is exact (`duration('P1D') / 24` is `PT1H`), and calendar units follow the local calendar across a daylight-saving change (issues #10198, #10199, #10200). `sum()` and `avg()` accept durations.

Impact: a query that compared durations with `<` or `>`, or compared a `localdatetime()` with a `datetime()`, answers differently.

Action: compare two values of the same temporal type, and compare durations by their components or by a number you compute from them. See Temporal comparison.

26.11.1 - A DATE compared with a DATETIME is midnight of that day, and an impossible date is refused

A `DATE` compared with a `DATETIME` is now compared as midnight of its day, and a date such as `2021-02-30` is refused instead of being moved to the last day of its month (issue #10206).

Impact: `WHERE day = '2021-02-30'` matches nothing, and a `DATE` no longer equals a `DATETIME` of the same day at noon.

Action: none for valid dates. See Managing dates.

26.11.1 - A `PRODUCT` vector index ranks through its PQ codes and re-ranks

A search of a `PRODUCT` index walks the graph on the in-memory PQ codes, keeps `k * rerankOversample` candidates and scores them on the stored vectors, so the returned distances stay exact. Before, every visited node was scored on the stored vectors. New index settings `rerankOversample` (default 10) and `fusedPQ`, a per-query `rerank` option, and `pqTrainingLimit` is now honoured (issues #10514, #10975).

Impact: the ranking of a `PRODUCT` index follows the PQ codes first, so recall at a low `rerankOversample` can differ from before.

Action: raise `rerank` or `rerankOversample` if recall matters more than speed. See Product quantization search.

26.11.1 - PostgreSQL: `format_type()` and `regtype` print PostgreSQL's type names

`format_type()` and the `regtype` cast print `integer` and `character varying`, as PostgreSQL does, and the catalog answers the queries of psql (`\dt`, `\d`, `\di`, `\dm`, `\df`, `\du`) and psqlodbc (issue #10228).

Impact: a client that compared the old names of `format_type()` must compare the PostgreSQL ones.

Action: none for standard drivers. See PostgreSQL driver.

26.11.1 - HA: an index compaction with an unknown outcome quarantines the leader

When the cluster cannot say whether it accepted an index compaction (the publication timed out,

or the entry was lost to a leadership change), the leader is quarantined and resyncs instead of leaving its index silently short (issue #11240).

Impact: the leader can hand off leadership after such a compaction, and the database resyncs from the new leader.

Action: none. See Quarantine.

26.11.1 - Storage: the chunk-link format has an opt-in setting

A multi-page record is stored as a chain of chunks, each holding a link to the next one, and a chunk at position 0 of a bucket could not be linked (issues #10701, #10854). The database setting `arcadedb.bucketSelfDescribingChunkLinks` (default `false`) makes the writer store the link as `-(position + 1)`, which lets position 0 be linked (issues #10949, #11263). Every 26.11.1 server reads both forms, and nothing writes the new one by default.

Impact: with the setting `true`, a database that has stored the new form cannot be read by a version older than 26.11.1. Making it the default is planned for 27.1.1 (issue #10950).

Action: turn it on only when no node, replica or restore target runs an older version. In HA the setting is documented, not gated: set it on every node. See Settings.

26.10.2 - Client requests wait in a queue when the server is busy

Every request of a remote client (HTTP, Postgres, BOLT, Redis, gRPC, Gremlin Server, MCP, MongoDB) now goes through a query admission gate: at most twice the number of cores run at once, and a request that arrives when they are all taken, or when the running queries hold more than 80% of the query heap budget, waits in a queue and starts in arrival order (issue #9518). Before, every request started at once, and under a burst of heavy queries the last ones were refused by the heap budget or exhausted the heap.

Impact: under load a request can wait before it starts, up to `arcadedb.queryQueueTimeout` (30 seconds by default), and is refused as retryable once that runs out or when `arcadedb.queryQueueMaxSize` requests already wait (HTTP `503`). A MongoDB request is refused at once instead of waiting. A BOLT result stream that is not read to the end keeps its slot until it is.

Action: none for most installations. Size the gate with `arcadedb.queryMaxConcurrent`, `arcadedb.queryAdmissionHeapWatermark`, `arcadedb.queryQueueTimeout` and `arcadedb.queryQueueMaxSize` (see Settings), or set `arcadedb.queryMaxConcurrent=0` for the previous behavior.

26.10.2 - The Gremlin plugin fails the server start when its port is busy

If the Gremlin port (`gremlin.port`, default `8182`) is already in use, the server now stops with an error. Before, it started without a Gremlin listener and still advertised the port in `GET /api/v1/server` (issue #9319). With `gremlin.port=0` a free port is chosen at startup and advertised.

Impact: a server that used to start while another process held the Gremlin port no longer starts.

Action: free the port or set `gremlin.port` to a free one.

26.10.2 - A plain `=` / `IN` on a `COLLATE CI` index is case sensitive

The planner used to return every case variant for `WHERE Name = 'JOHN'` when a `COLLATE CI` index served the query, while the same predicate evaluated by a scan was case sensitive (issue #9403). The index no longer changes the answer: a plain `=` or `IN` is case sensitive with or without it.

Impact: a query that relied on `Name = 'john'` finding `"John"` through a `COLLATE CI` index now returns fewer rows.

Action: write the lookup on `Name.toLowerCase()`, for example `WHERE Name.toLowerCase() = 'john'`, which the index serves. See Case-insensitive indexes.

26.10.2 - Cypher: a path variable over a quantified group is one path, and ignored GQL constructs are refused

`MATCH p = m)-[r]→(n{2}` used to bind `p` to a list with one path per repetition. It now binds one path running through every repetition, as any other path variable does and as Neo4j Cypher 25 does: `valueType(p)` is `PATH NOT NULL` and `length(p)` is 2 (issue #9940). Variables declared inside the group (`m`, `r`, `n`) are still lists.

GQL constructs that the parser accepted and the engine ignored now fail with a `FeatureNotImplemented` error that names the construct:

- the path search prefixes `ANY SHORTEST`, `ALL SHORTEST`, `SHORTEST k`, `SHORTEST k GROUPS`, `ANY` and `ANY k`
- the match mode `REPEATABLE ELEMENTS`
- `x IS DIRECTED`
- `USE` of a graph other than the current database
- a quantified group nested inside another one

An `ORDER BY`, `SKIP`, `OFFSET` or `LIMIT` written after `RETURN` (`RETURN … LIMIT 2 OFFSET 1`) is now applied after the RETURN's sort, as one page: sort, then skip, then limit.

Impact: a query that read such a `p` as a list (`size(p)`, `p[0]`) gets a path instead. A query that used one of the refused constructs fails instead of returning rows the construct did not select.

Action: use `length(p)`, `nodes(p)` and `relationships(p)` on the path. Replace a shortest-path prefix with `shortestPath()` or `allShortestPaths()`, and drop `REPEATABLE ELEMENTS`, `IS DIRECTED` and `USE`. See ISO GQL Coverage Statement.

26.10.2 - Cypher `DURATION` arithmetic is exact, and `date * duration` is an error

Durations stored in a property are read back as durations, including short ones such as `P1D`, so `ORDER BY` and comparisons on them now work (issue #9338). Duration arithmetic raises an arithmetic error when the result does not fit, instead of returning a wrong value. A date or time combined with a duration using `*`, `/` or `%` is now a type error, and `duration + date` is accepted.

Impact: a query that relied on a stored duration being plain text, or on `date * duration` returning a date, changes.

Action: none for normal use. A range index on a duration property still orders by its text form (issue #9348).

26.10.2 - Sparse-vector top-K is split across workers from 50,000 postings

`vector.sparseNeighbors` on an `LSM_SPARSE_VECTOR` index now splits a query across the scoring workers once its terms hold 50,000 postings in total (setting `arcadedb.sparseVectorScoringMinPostingsForPartitioning`, default `50000`, was `200000`). Queries between 50,000 and 200,000 postings used to run on one thread and set the slowest latencies. A query just past the threshold uses two ranges and leaves the other workers free; wider queries use more (issue #9482).

Impact: results are the same. Queries in that size range are faster, and use more CPU when the server is otherwise idle.

Action: none. On a small or shared server where CPU matters more than latency, raise `arcadedb.sparseVectorScoringMinPostingsForPartitioning` (for example back to `200000`).

26.10.2 - Production mode conceals error text on Bolt, Redis, `/ws` and MCP

In `production` server mode the `/ws` insert session, `/ws` change-event subscriptions, Bolt, Redis and MCP now answer an engine failure with "The request failed. Check the server log for the details" and write the detail to the server log, as HTTP, gRPC, PostgreSQL, MongoDB and Gremlin Server already did (issue #8749). Before, they returned the engine's message as is, including the stored values of a duplicated key.

Impact: a client of these protocols that reads the error text in production gets the generic message. Error codes, Neo4j status codes, RESP prefixes and exception class names are unchanged, and so is text the protocol writes about the request itself.

Action: branch on the code rather than on the message, and read the detail from the server log. See Error details in production mode.

26.10.2 - `list queries` names the client on every protocol

Every entry of `list queries` and `SHOW TRANSACTIONS` now carries the client that asked for the work: `clientAddress` for MCP tool calls (`stdio` for the stdio transport), `/ws` insert sessions, the Gremlin Server and gRPC, which listed none before, plus a `connectionId` for the Gremlin Server and `/ws` (issue #9962). A client whose address the transport cannot report has no `clientAddress` field, where it used to be listed as `"null"`.

Impact: tools that read the listing see the new fields. Plugins that open entries through `RunningQueryRegistry.register(…)` or `open(…)` now pass a `RunningQuery.Client` (`RunningQuery.Client.NONE` when there is no client).

Action: none for clients. Plugin code: add the client argument. See `list queries`.

26.10.2 - Graph algorithms answer the incoming side of unidirectional edge types

The `algo.*` and `path.*` procedures, the path-finding SQL functions (`dijkstra()`, `astar()`,

`bellmanFord()`, `cchShortestPath()`, `duanSSSP()`) and the `node.degree*` / `node.relationship.*` functions now follow the edges of a `UNIDIRECTIONAL` type from their target too, as a Graph Analytical View already did (issue #8629). Before, they saw such an edge from its source only, so for example a walk with direction `IN` or `BOTH` missed it. `refactor.mergeNodes`, `refactor.cloneNodesWithRelationships` and changing a node's labels in Cypher no longer drop the incoming unidirectional edges of the node.

Impact: results over unidirectional edge types can change (more paths, higher in-degrees, different scores), and now match the results with a Graph Analytical View.

Action: none. See `CREATE TYPE` for the heap a walk from the target takes.

26.10.2 - Cypher counts match an undirected self loop once

An undirected relationship pattern such as `(n)-[:KNOWS]-(m)` matches a self loop (an edge from a vertex to itself) once, as the openCypher specification and Neo4j do. The rows of such a pattern always did, but the counts computed without materializing the rows counted it twice: `MATCH (a)-[:KNOWS]-(b) RETURN count(*)`, a `count(*)` grouped by the start node, `OPTIONAL MATCH … WITH n, count(m)` and the `COUNT { }` subquery, with and without a Graph Analytical View (issues #8750, #9540). The same release corrects the count of a chain with an inequality such as `WHERE a <> c` when the chain has a labelled or multi-hop part before or after the two positions the inequality compares.

Impact: those counts can be lower than before on graphs with self loops, or on such chains, and now equal the number of rows the same `MATCH` returns.

Action: none.

26.10.2 - Cypher `count(*)` of an `OPTIONAL MATCH` with several arms, and of a Cartesian product

A `count(*)` over a star pattern no longer counts an `OPTIONAL MATCH` that reaches the central node from two sides, or holds two patterns, as two independent optional matches: when one side finds nothing the clause contributes one row, not the number of matches of the other side. A star next to a node pattern that shares no variable with it, `MATCH (a)-[:R]→(b), (a)-[:S]→(c), (t:Tag)`, now multiplies by the `Tag` count, which it used to leave out. Both were answered by the count push-down without building the rows (issue #9596).

`count(*)` over parts of a `MATCH` that share no variable is now the product of the parts' counts and builds no row, so a count whose result does not fit a 64-bit integer raises an error instead of running until the command timeout.

Impact: the counts of those two star shapes change, and now equal the number of rows the same `MATCH` returns.

Action: none.

26.10.2 - A Postgres cancel request stops the statement and keeps the connection

A `CancelRequest` (what JDBC `Statement.cancel()`, pgjdbc's `setQueryTimeout()` and libpq's `PQcancel` send) now stops the statement the connection is running with SQLSTATE `57014` and leaves the connection open, as PostgreSQL does (issue #9689). Before, it closed the whole connection, and

since every connection had the backend process id `0`, a cancel could reach a different connection than the one it named.

Impact: after a cancel or a query timeout the connection stays usable instead of being closed.

Action: a client that relied on a cancel request closing the connection must call `pg_terminate_backend(pid)` instead. See Cancelling a statement.

26.10.2 - An HTTP client that disconnects terminates its statement

A client that closes its connection while its statement runs (its own timeout, a killed process, `curl --max-time`) now terminates the statement, and what it wrote is rolled back unless it had already committed, instead of leaving the server working on it until it completes or reaches `arcadedb.command.timeout` (issue #9689). This applies to plain HTTP/1.x connections, not to TLS or HTTP/2. Likewise a gRPC client that cancels a call, or whose deadline expires, terminates the server-side work.

Impact: a client that sends a write and closes the connection without waiting for the answer can find the write rolled back.

Action: wait for the response of a write before closing the connection. See List and terminate running queries.

26.10.2 - openCypher temporals are stored natively

openCypher used to store `time()`, `localtime()`, zoned `datetime()` and `duration()` values as Strings and guessed the type back from the text, so a String such as `"09:00-17:00"` read back as a time. They are now stored in four new property types, `OFFSET_TIME`, `LOCAL_TIME`, `ZONED_DATETIME` and `DURATION`, and a new database never reinterprets a String (issues #8572, #9462, #9636). A zoned `datetime()` written to a `DATETIME` property is stored as an instant and reads back as the UTC wall clock; before, it kept the local wall clock.

Impact: a database written by an earlier version keeps the old reading of undeclared properties until it is migrated, so existing queries keep working after the upgrade. The migration is one-way and converts every String with a temporal shape on an undeclared property, including one meant as text.

Action: take a backup, quiesce writers, and run `CALL db.migrateLegacyTemporals() YIELD records, properties` once per database (under HA, on the leader; the user needs `updateDatabaseSettings`). To convert only some properties, declare each with its new type and copy the values through a scratch property instead. See Migrating Cypher temporals stored as Strings.

26.10.2 - Weighted path finders share one edge-weight rule

Every weighted shortest-path finder now reads an edge weight the same way: an edge without the weight property, or with a value that is not a number, weighs 1, and an edge whose weight is negative, NaN or infinite is not walked (issue #9443). Before, `dijkstra()`, `astar()`, `algo.dijkstra` and `algo.astar` treated a missing weight as 0, and several finders walked negative weights. `duanSSSP()` now accepts an options map `{ direction, edgeTypeNames }` as its fourth argument, answers through the same engine as `cchShortestPath()` and honors the command timeout. `bellmanFord()` and

`algo.bellmanford` are unchanged.

Impact: on a graph where some edges have no weight, `dijkstra()`, `astar()`, `algo.dijkstra` and `algo.astar` return different paths and distances. A graph with negative weights is now routed around those edges. Among paths of equal weight `duanSSSP()` may return a different one. Embedded code: the `MIN` constant of `SQLFunctionHeuristicPathFinderAbstract` is gone.

Action: give every routed edge a weight if 0 was intended, and use `bellmanFord()` where negative weights must be walked. A custom subclass reads the rule from `com.arcadedb.graph.EdgeWeight`. See `dijkstra()` and `duanSSSP()`.

26.10.2 - SQL `UPSERT` requires a UNIQUE index

`UPDATE ⋯ UPSERT WHERE` without a sub-query used to accept any index, so an upsert matched through a non-unique index could create duplicates. It now needs an equality on every property of a UNIQUE index, as the sub-query form already did (issues #9339, #9342).

Impact: an `UPSERT` served by a non-unique index, by part of a composite key, or by a range, `IN` or `OR` condition fails with an error stating the requirement.

Action: create a UNIQUE index on the properties of the `WHERE` condition, or rewrite the condition as equalities on all of its properties. See `UPDATE`.

26.10.2 - SQL `sum()` over an empty or all-NULL group returns NULL

SQL `sum()` now returns NULL for an empty or all-NULL group, like `avg()`, `min()` and `max()`. It used to return 0 (issues #9351, #9384). Cypher `sum()` keeps returning 0, as openCypher requires.

Impact: SQL queries that read `sum()` of such a group get NULL.

Action: wrap the aggregate where 0 is expected, for example `ifnull(sum(amount), 0)`.

26.10.2 - SQL `k = <list>` no longer expands the list

An index lookup used to expand a list on the right of `=`, so `k = [1, 2]` behaved like `k IN [1, 2]`, while a scan compared the field with the list. Only `IN` and the other multi-value operators expand a list now; `=` and the range operators compare with the list as one value, which matches nothing on a scalar field (issues #9370, #9385).

Impact: a query that relied on `=` with a list to match any element returns no rows.

Action: use `IN`.

26.10.2 - A statement that hides a write inside a read is refused on the query path

SQL decided whether a statement reads, writes or changes the schema from its top-level node only, so a write placed in a `LET`, a projection, a `FROM` sub-query, a `CASE`, a `RETURN` or a script block ran durably through the read-only paths (issues #9628, #9687). A statement now counts everything nested in it. `PROFILE` takes the classification of the statement it wraps, so `PROFILE INSERT` is a write (issue #9340). `EXPLAIN` stays read-only.

Impact: `Database.query()`, `GET` and `POST /api/v1/query` and MCP identities without write permission refuse such statements. A DDL inside `IF`, `FOREACH` or `WHILE` needs `allowSchemaChange` on MCP.

Action: send these statements as commands (`Database.command()`, `POST /api/v1/command`).

26.10.2 - SQL `include()` and `exclude()` return a map

`include()` and `exclude()` used to build a record carrying the source record identity, so calling `save()` on a projection overwrote the stored record. They now always return a map with no `@rid`, `@type` or `@cat` unless one of the names asks for it, and they work on edges (issues #9840, #9873). `UPDATE V SET copy = @this.exclude('secret')` now stores an embedded map instead of a link to the record itself.

Impact: code that treated the result as a record (saving it, reading its `@rid`, calling `.out()` on it) changes.

Action: apply traversals before the projection, and name `@rid` in `include()` when you need it. See `include()`.

26.10.2 - SQL result rows: `values()` follows `keys()`, and a bare `ORDER BY @rid` on a `MATCH` with columns is refused

`values()` on a sub-query row or `LET` variable used to return the record metadata as extra unlabeled values. It now returns one value per key, in the order of `keys()` (issue #9843). A SQL `MATCH` that returns columns, `$paths` or `$patterns` builds rows with no `@rid`, so `ORDER BY @rid` sorted on null and kept the pattern order; it now fails at planning time (issue #9911). The same applies to other bare record attributes such as `@type`.

Impact: code that read the extra values of `values()` sees fewer, and a keyset pagination that used a bare `ORDER BY @rid` on such a `MATCH` fails.

Action: write `ORDER BY <alias>.@rid`, or return the RID as a column and order by it. See `MATCH`.

26.10.2 - Schema refuses a `REGEXP` no value can match, and constrained properties on `LIGHTWEIGHT` edges

`CREATE PROPERTY`, `ALTER PROPERTY` and `Property.setRegexp()` refuse a pattern that no value of the property type can match, such as `[A-Z]+` on a `LONG` or a `DATE`, which used to be accepted and then refused every write (issues #9853, #9871). A pattern the check cannot decide in time is accepted with a WARNING. A `MANDATORY` or default-valued property can no longer reach a `LIGHTWEIGHT` edge type through a supertype, by `CREATE EDGE TYPE … EXTENDS … LIGHTWEIGHT`, `ALTER TYPE … SUPERTYPE`, `ALTER TYPE … WITH lightweight = true` or by declaring the property on the supertype (issue #9846).

Impact: such DDL fails with a `SchemaException`. A schema that already holds such a pattern or combination still opens.

Action: fix the pattern, or remove it with `ALTER PROPERTY … REGEXP null`. Make the inherited property optional, or keep the edge type record-backed. See `ALTER PROPERTY` and Lightweight edges.

26.10.2 - Cypher `MERGE` matches what `MATCH` matches, and declared properties compare in their declared type

`MERGE` used its own equality and disagreed with `MATCH`: it missed a stored `FLOAT`, creating a duplicate, and matched a temporal against the text of a `STRING` property (issues #9693, #9696). It now uses the comparison `MATCH` applies. On a property declared `STRING`, a stored text also equals a number, boolean or temporal whose written text it is, so `MATCH (n:S) WHERE n.v = 1` finds the node holding the text `'1'`; on a property declared `INTEGER`, `LONG`, `FLOAT`, `DOUBLE`, `BOOLEAN`, `DATE` or `DATETIME`, a text equals the value it converts to (issues #9695, #9794). Undeclared properties keep strict Cypher typing.

Impact: a repeated `MERGE` no longer duplicates nodes in these cases, and `=`, `<>`, `IN` and joins on declared properties can return more rows than before.

Action: none, unless a query relied on the old mismatch. See `MERGE`.

26.10.2 - `DROP CONSTRAINT <name>` does not drop named constraints created by older releases

A named Cypher constraint now records what its `CREATE CONSTRAINT` created, and `DROP CONSTRAINT <name>` removes exactly that; before, it did nothing (issues #9646, #9707). Creating a constraint under a name in use and dropping an unknown name now raise an error, unless `IF NOT EXISTS` or `IF EXISTS` is given.

Impact: a named constraint created by an earlier version has no such record, so `DROP CONSTRAINT <name>` raises an error for it.

Action: drop an old constraint by its index name, or through SQL (`DROP INDEX`, `ALTER PROPERTY … MANDATORY false`). See `DROP CONSTRAINT`.

26.10.2 - The `path.*` procedures answer as `apoc.path.*` does

`path.expandconfig` now reads its `uniqueness` option, with the APOC default `RELATIONSHIP_PATH`, and `path.expand` is breadth-first with `RELATIONSHIP_PATH` (issue #10049). Before, the breadth-first walk was always `NODE_GLOBAL` and the depth-first walk and `path.expand` always `NODE_PATH`. `path.subgraphall` returns every relationship between the returned nodes, whatever the `relationshipFilter` (issue #10048). `relationshipFilter` and `labelFilter` read the APOC syntax, and an unknown relationship type or label is refused (issue #10025).

Impact: on a graph with cycles or several routes to a node these procedures return more paths, growing quickly with `maxLevel`. A walk too wide for `arcadedb.queryMaxHeapElementsAllowedPerOp` or `arcadedb.queryMaxHeapRAM` fails the query.

Action: to keep the old answers add `uniqueness: 'NODE_GLOBAL'` to a breadth-first `path.expandconfig`, `uniqueness: 'NODE_PATH'` to a depth-first one, and replace `path.expand` with the equivalent `path.expandconfig` call with `bfs: false, uniqueness: 'NODE_PATH'`. See `path.expandconfig`.

26.10.2 - An ISO duration text is parsed as Neo4j parses it

`duration('<text>')`, a `DURATION` property set from text and a gRPC value tagged `duration` now follow the Neo4j grammar (issue #10055). Refused: a text with no component (`P`, `PT`, `P1YT`), a malformed number (`P.5Y`, `PT1.S`) and a fraction on any component but the last one (`P1.5Y2M`). Newly accepted: a

sign before the `P` or on a component, and a decimal comma.

Impact: writes and queries that used such texts fail instead of storing a zero or truncated duration.

Action: correct the duration texts. See `DURATION`.

26.10.2 - Time-series writes refuse what they used to store silently

- A line protocol, gRPC or remote write with an empty tag value (`cpu,host= value=1`) is refused with HTTP `400` or `INVALID_ARGUMENT`, as InfluxDB does. Leave the tag out instead (issues #8647, #9364).
- A sample carrying a tag, field, property or label its type does not declare in that role is dropped and reported as a partial write (HTTP `400` with `undeclaredKeys`, or gRPC `undeclared_keys`). SQL `INSERT` fails, and Prometheus remote write drops the series (issues #8646, #9365, #9366, #9384). A sample that omits a declared key is still accepted. A Prometheus label with an empty value is an absent label.
- A column codec that cannot encode the column type is refused at `CREATE`; types already persisted keep loading (issues #9310, #9384).
- A line protocol line cut right after a tag or field `,` separator is refused (issues #9323, #9392).

Impact: producers that send extra tags (Telegraf, for example), empty tag values or a misspelled key see refusals where they used to succeed.

Action: fix the producers, or set the database setting `arcadedb.timeSeriesUndeclaredKeys=ignore` for schema-less producers. See Undeclared keys.

26.10.2 - Unbounded time-series reads stream

`SELECT FROM <timeseries type>` without `LIMIT`, and the gRPC time-series stream without a row limit, now stream rows instead of loading the whole range first (issues #9420, #9421).

Impact: a retention pass that runs during such a read gives a shorter answer, since the removed rows are skipped. A downsampling pass that runs during it fails the read with a retryable error (HTTP `503`, gRPC retry category), possibly after some rows were sent.

Action: retry a read that fails this way, and expect a client to have received partial rows before the error.

26.10.2 - PromQL refuses what Prometheus refuses

The PromQL engine now answers HTTP `400 bad_data` on `/prom/api/v1/query` and `/query_range`, and an error in SQL `promql()`, for expressions it used to answer with NaN, an empty result or an unchanged value (issues #9723, #9743, #9788, #9883, #9922, #9924): a comparison between two scalars without `bool` (`1 > 0`), `and`, `or` and `unless` with a scalar operand, a range vector or a string as an operator operand, a function or aggregation argument of the wrong type or count (`sum(m[5m])`, `abs(-1)`), and a range query over a range vector or a string. `bool` right after an operator is now the comparison modifier, so a metric named `bool` cannot be the right operand.

Impact: dashboards and alert rules that used these expressions get an error instead of a value.

Action: rewrite them as Prometheus requires, for example `1 > bool 0`. See Refused PromQL expressions.

26.10.2 - Time-series filters, Prometheus endpoints and `ts.movingAvg` change results

- The PromQL label matchers `=`, `!=`, `=~` and `!~` read a typed tag the same way on `/query`, remote read and `/series`, so `=` and `!=` select complementary sets, and `=~"007"` selects a `LONG` tag holding 7 (issues #9833, #9880).
- The JSON tag filter of `POST /api/v1/ts/{database}/query`, `GET /api/v1/ts/{database}/latest`, the Grafana query and the gRPC `TimeSeriesQuery` and `TimeSeriesLatest` calls reads a `DATE` or `DATETIME` value in the database time zone instead of the JVM one (issues #9879, #9929). Embedded API: `TimeSeriesGateway.andTag()` and `buildTagFilter()` now take the `Database` as an argument.
- Prometheus remote write and read answer `415` for a `proto=` content type they do not implement, so a remote-write 2.0 sender falls back to 1.0 instead of losing the batch, and `400` instead of `204` for a non-empty body with no remote-write 1.0 field (issues #9624, #9755).
- `ts.movingAvg()` keeps one element per row: a null sample keeps its slot instead of being dropped, which shifted every later average into an earlier row (issue #9958).

Impact: some matcher and tag filter results change, a misconfigured remote-write sender now gets an error, and `ts.movingAvg()` results line up with their rows.

Action: recompile embedded code that calls `andTag()` or `buildTagFilter()`; otherwise none.

26.10.2 - `GraphBatch.withLightEdges(true)` needs a `LIGHTWEIGHT` edge type

`GraphBatch` with the deprecated `withLightEdges(true)` flag now throws for a property-less edge into an edge type that does not declare `LIGHTWEIGHT`, instead of creating light edges in it (issues #9378, #9383). The batch HTTP endpoint answers `400`.

Action: declare the edge type `LIGHTWEIGHT`. See Lightweight edges.

26.10.2 - A Graph Analytical View over some of the vertex types no longer accelerates traversals

A view created with `VERTEX TYPES` naming only some vertex types was chosen by edge type alone, so SQL `out()`, SQL `MATCH` and Cypher `MATCH` answered 0 for edges leaving the uncovered types (issues #9301, #9320). Such a partial view no longer serves count push-downs, SQL traversal, shortest paths or `algo.*` procedures. A view covering every vertex type is unaffected.

Impact: queries that used a partial view now return correct results through the OLTP path, which can be slower.

Action: to keep the acceleration, create the view over every vertex type. See Graph analytical views.

26.10.2 - HA snapshot ship compresses at DEFLATE level 1

The leader compresses a database snapshot shipped to a follower at the new `arcadedb.ha.snapshotCompressionLevel`, default `1` (fastest), instead of the JDK default level 6, and serves two followers concurrently (issues #9295, #9296).

Impact: a full resync is faster on a LAN, but sends more bytes over a WAN or metered link.

Action: set `arcadedb.ha.snapshotCompressionLevel=6` (or `-1`) to restore the previous level. See SERVER settings.

26.10.2 - HA: a follower restarted in place is no longer reformatted automatically

A follower whose Raft layer was restarted in place, and that has since taken no replicated entry and seen no new leader, is no longer reformatted when it looks stuck at a stale term, because that signature also matches a dead replication path (issues #8901, #9001). A node that was never restarted in place keeps the automatic recovery of `arcadedb.ha.divergedFollowerRecovery`.

Impact: such a node keeps raising the `follower-stuck-at-stale-term` alert and logs a WARNING every 5 minutes instead of healing itself.

Action: restart the node by hand when the alert persists.

26.10.2 - HA rolling upgrade: type renames and database settings wait for every node

`ALTER TYPE … NAME` and `ALTER DATABASE` / `set database setting` now replicate to every node; before, they changed the node that ran them only (issues #9762, #9763). While any peer runs an older build or cannot be reached, a rename is refused up front with HTTP `409`, and a database setting change with HTTP `409` / gRPC `FAILED_PRECONDITION`; nothing changes (issues #9891, #9904, #9919). Path-valued settings (`arcadedb.externalPropertyBucketPath`, `arcadedb.pageSnapshotSpillPath`, `arcadedb.opencypher.loadCsv.importDirectory`) now apply on every node. No setting overrides the rename refusal; `arcadedb.ha.securityEntryCapabilityGate=false` skips the database setting check, at the risk of halting a peer that cannot decode the entry (see how-to/operations/ha.pdf).

Impact: during a rolling upgrade, and while a node is down, these commands fail. Once a rename has been replicated, the cluster cannot be downgraded past 26.10.2 on its retained Raft log.

Action: finish the rolling upgrade before renaming types or changing database settings. To rename a type while a peer is gone for good, remove that peer from the Raft configuration first. Make sure path-valued settings point to a directory that exists on every node. See HA cluster upgrade.

26.10.2 - HA vector indexes that lost entries on a follower must be rebuilt

A vector insert committed through a follower could reuse a vector id the leader had already allocated and replace that vector on every node, so `countEntries()` stayed flat and the replaced vectors dropped out of search (issues #9428, #9507). The cause is fixed, but the lost vectors are not restored.

Impact: an HA vector index that received inserts through followers before the upgrade can be missing entries.

Action: run `REBUILD INDEX <name>` on such vector indexes after every node runs 26.10.2. See `REBUILD INDEX`.

26.10.2 - Vector graphs are rebuilt after a snapshot install or a restore

A snapshot install now ships a vector index graph file (`.vecgraph`) empty, and the follower builds and verifies its own graph on the first search (issues #9758, #9889). A full backup archives the graph with its side files only when it can prove they belong to the pages it captured; otherwise it archives the graph empty, and the restored index builds it on the first search (issues #9888, #9917, #9918). See Vector indexes.

Impact: the first search on a vector index after a snapshot install, or after a restore whose graph could not be archived, pays one graph build. Archives taken before 26.10.2 restore the old graph unverified (counted in `unverifiedGraphReuses` and logged at `WARNING`).

Action: after restoring an archive taken with an earlier version, run `REBUILD INDEX` on its vector indexes. See Restore.

26.10.2 - A follower's write refusal is a redirect on every protocol

A write sent to an HA follower is refused with the leader address. Bolt, MongoDB and Redis reported it as a retryable conflict, so drivers retried against the same follower. A refusal that names the leader is now a redirect (issues #9845, #9868, #9869, #9930, #9931): Bolt `Neo.ClientError.Cluster.NotALeader` (was `DeadlockDetected`), MongoDB `10107 NotWritablePrimary` (was `112 WriteConflict`), Redis `READONLY` (was `TRYAGAIN`), the `/ws` insert endpoint code `NOT_LEADER` (was `DB_ERROR`) with a `leaderAddress` field, and the gRPC streaming inserts `InsertError.code = NOT_LEADER` (was `DB_ERROR` or `COMMIT_FAILED`) with `leader_address`. PostgreSQL keeps `40001`. During an election, with no leader known, the old retryable answers stay.

Impact: a `bolt://` (not `neo4j://`) or plain MongoDB client no longer retries a leader refusal, and clients that matched the old codes miss the new ones.

Action: send writes to the leader or use a routing driver, and handle the new codes. See `/ws` insert session.

26.10.2 - Gremlin Server in production mode refuses a channelizer that cannot hide error details

With `arcadedb.server.mode=production`, a Gremlin Server channelizer other than TinkerPop's default `WebSocketChannelizer` (`WsAndHttpChannelizer`, `HttpChannelizer`, `UnifiedChannelizer` or a custom class), set by `channelizer:` in `gremlin-server.yaml` or by `gremlin.channelizer`, now fails the server start, because it would send engine error text, including stored values, to clients (issues #9639, #9750). Outside production mode the server starts with a WARNING.

Impact: a production server configured with such a channelizer does not start after the upgrade.

Action: remove the `channelizer` entry or set it to `org.apache.tinkerpop.gremlin.server.channel.WebSocketChannelizer`. See Gremlin Server.

26.10.2 - Security checks that now apply to open sessions, GraphQL and LOAD CSV settings

- PostgreSQL, Bolt, MongoDB, Redis and `/ws` sessions re-check their user against the live security state on every request or frame (issues #9279, #9311, #9340). Deleting a user, changing a password or revoking a database grant now applies to open connections; a Bolt transaction or

`/ws` session whose authorization is gone is rolled back.

- A GraphQL type definition replaces the database GraphQL schema, so it needs `UPDATE_SCHEMA` (issue #9280).
- The settings that control what `LOAD CSV` may reach (remote URLs, local files, import directory, blocked IP ranges) need `updateSecurity` to change through `ALTER DATABASE` (issue #9371).

Impact: long-lived clients can be disconnected or refused after a security change, and users without these permissions are refused.

Action: grant `updateSchema` to the users that define GraphQL types and `updateSecurity` to the users that change the LOAD CSV settings. See Security.

26.10.2 - Rotate passwords set through `server.defaultDatabases` if diagnostics were uploaded

Before 26.10.2, support diagnostics masked secrets by setting name only, so a password inside `-Darcadedb.server.defaultDatabases=Universe[albert:einstein]` was written in clear to `diagnostics.json`, uploaded to the portal and put in the support bundle (issues #9625, #9678). It is now masked there and in logs.

Action: on a server that uploaded support diagnostics before 26.10.2, rotate every password configured through `arcadedb.server.defaultDatabases`. See Support.

26.10.2 - Subtype records keep inherited `EXTERNAL` values inline until `REBUILD TYPE`

A subtype that inherits an `EXTERNAL` property now stores its values in its own `_ext` bucket; before, they were written inline (issues #9683, #9815).

Impact: subtype records written before the upgrade keep those values inline. They read normally, but do not get the storage benefits.

Action: run `REBUILD TYPE <Subtype>` to move them. See External property storage.

26.10.2 - Creating a user or a database under a taken name answers `409`

Creating a user whose name exists used to answer `403` "Security error", and creating a database whose name exists `400` (issues #10073, #10074). Both now answer `409` over HTTP (REST routes and server commands, including `import database`) and `ALREADY_EXISTS` over gRPC. The Java remote clients raise `RemoteAlreadyExistsException` for a taken user, API token or database name (issue #10075). Restoring into a taken name keeps `400`.

Action: clients that matched `403` or `400` for these cases must handle `409`.

26.10.2 - `BACKUP DATABASE` applies the retention policy

The SQL `BACKUP DATABASE` statement now prunes the archive directory of the database after a successful backup, as the scheduler and the `trigger backup` command already did (issue #9844). With auto-backup disabled nothing is pruned.

Impact: with auto-backup enabled, archives beyond the `maxFiles` (or tiered) retention of the effective policy are deleted, also for a database absent from the auto-backup configuration. The

archive just written, and archives that do not follow the `<db>-backup-<timestamp>.zip` naming, are never deleted.

Action: give archives you want to keep a custom name, for example `BACKUP DATABASE 'before-the-upgrade.zip'`. See Backup.

26.10.2 - gRPC settings are read from the configuration file and environment

Every `arcadedb.grpc.*` setting is now a registered server setting, so the server configuration file and environment variables reach it (issue #9316). Before, only `arcadedb.grpc.port` was, and the others worked as JVM system properties only: a `grpc.tls.enabled=true` written in the configuration file was ignored and the endpoint stayed in cleartext. An invalid `grpc.mode` or `grpc.tls.enabled` value now makes the server refuse to start, `grpc.host` is applied to the standard server, and `grpc.compression.enabled=false` is honored (issue #9782).

Impact: `grpc.*` values that sat unused in the configuration file now take effect at the next start, which can turn on TLS or change the bind address.

Action: review the `grpc.*` entries of the configuration file and environment before upgrading. See gRPC settings.

26.10.2 - Docker images run on Java 25, and Studio is served with a Content-Security-Policy

The Docker images are built on `eclipse-temurin:25-alpine` instead of Java 21, which enables compact object headers, and `ARCADEDB_OPTS_GC` drops the obsolete `-XX:+ZGenerational` (issue #9602). The jars still run on Java 21 for embedded use. Studio and the Swagger UI page are served with a `Content-Security-Policy` that refuses inline script (issue #9681), set by `arcadedb.server.httpContentSecurityPolicy`.

Impact: custom images or scripts that set JVM flags removed in Java 25 fail to start. A reverse proxy or extension that injects inline script into Studio is blocked.

Action: check custom JVM options against Java 25. To try a custom policy without blocking, set `arcadedb.server.httpContentSecurityPolicyReportOnly=true`. See Docker / Kubernetes.

26.10.2 - New and rebuilt hash indexes use page layout 3, which older servers cannot open

A `NOTUNIQUE_HASH` key whose record ids would take more than a quarter of a page now keeps them in a chain of pages of their own, so adding a record to a key costs the same whatever the key already holds (issue #9228). Before, the cost of an insert grew with the record ids the key already held: 49 pages were written per insert into a key with 20,000 of them. Every hash index created or rebuilt by 26.10.2, `UNIQUE_HASH` included, is written at page layout version 3. Hash indexes created by older versions keep working at their layout and move to version 3 only on `REBUILD INDEX`.

Impact: none on upgrade. A server older than 26.10.2 knows layouts 1 and 2 only, and refuses a hash index at layout 3 (see Downgrading from 26.10.2 or later with hash indexes).

Action: to get the new layout on an existing `NOTUNIQUE_HASH` index with large keys, run `REBUILD INDEX` on it. See Hash index.

26.10.2 - Cypher paths are sent as one path object on every protocol

Every Cypher path is now one path value, whatever built it: a `MATCH p = ⋯` pattern, `shortestPath()`, `allShortestPaths()`, a path procedure, or `apoc.path.create`, `apoc.path.slice` and `apoc.path.combine` (issue #9913). Over HTTP (and `Result.toJSON()`), gRPC and PostgreSQL a path returned by a `MATCH` used to be sent as its text form; it is now the object `{"_type": "path", "nodes": [⋯], "relationships": [⋯], "length": n}`, with every node and relationship written whole whatever the JSON serializer settings. A Bolt driver receives a native `Path` for the paths of procedures and `shortestPath()` too, and Studio's graph view draws a returned path.

Impact: a client that parsed the text form of a `MATCH` path gets an object instead.

Action: read the `nodes`, `relationships` and `length` fields of the object, or return `nodes(p)` and `relationships(p)` explicitly. See reference/graph-algorithms/path-finding.pdf.

26.10.2 - `COMMIT RETRY` and `Database.transaction()` do not retry a leader refusal

A `NeedRetryException` now tells whether retrying on the same server can succeed (`isRetryableOnThisNode()`). An HA follower that refuses leader-only work (a schema change, a security mutation) and names the leader answers `false`, so `BEGIN ⋯ COMMIT RETRY <n>` and `Database.transaction(⋯, retries)` raise its `ServerIsNotTheLeaderException` on the first attempt (issue #9995). Before, they retried it `<n>` times with the retry delay in between, and `COMMIT RETRY ⋯ ELSE` then ran the `ELSE` block. A refusal raised during an election, which names no leader, is still retried.

Impact: the caller gets the leader refusal at once. A `COMMIT RETRY ⋯ ELSE { ⋯ } AND CONTINUE` block no longer runs its `ELSE` or hides the error for such a refusal.

Action: send leader-only work to the leader the error names. See reference/sql/sql-script.pdf.

26.10.2 - GraphQL introspection reports real types, and Cypher typed constraints keep a property's attributes

- GraphQL introspection of a type with no `.gql` declaration reports `MAP` and untyped `EMBEDDED` as the new `JSON` scalar, `EMBEDDED OF <T>` as an object, `LINK` as `ID`, `DECIMAL` as the new `BigDecimal` scalar and `BINARY` as a list of `Int`; before, they were all `String`. A sub-selection on a `LINK` property now resolves the linked record (issues #8756, #9467, #9468). See GraphQL type mapping.
- `CREATE CONSTRAINT ⋯ REQUIRE n.x IS :: <TYPE>` keeps the property's earlier attributes (`MANDATORY`, `READONLY`, `REGEXP`, `MIN`, `MAX`, `DEFAULT`) instead of dropping them, and checks the records already stored: when one fails, the constraint is refused and the old declaration stays (issue #9304). See reference/cypher/cypher-clauses.pdf.

Action: regenerate GraphQL client code built from introspection, and expect a typed constraint over existing data that does not fit to be refused.

26.10.2 - Embedded API changes

- `Schema.alterMaterializedView()` refuses a null mode and a negative interval (`IllegalArgumentException`) and an interval on a `MANUAL` or `INCREMENTAL` view (`SchemaException`), which it used to store (issues #8726, #9381). SQL `ALTER MATERIALIZED VIEW` is unaffected.

- `Schema.unregisterFunctionLibrary()` of a stored library is now saved to `schema.json` and replicated (issues #8879, #9423).
- `Iterator.remove()` on the iterators of `Vertex.getEdges()` and `Vertex.getVertices()` deletes the edge through the database, as `edge.delete()` does, and a refused delete reaches the caller with the entry left in place; a second `remove()` throws `IllegalStateException` (issue #9779). Before, an iterator over given edge types only unlinked the walked side, and one over every edge type logged a refusal and unlinked anyway. See reference/java-api/java-ref-database.pdf.
- On an HA follower, `unregisterFunctionLibrary()`, `Property` setters, `Schema.setExtension()`, the schema date settings and type setters such as `LocalEdgeType.setLightweight()` throw `ServerIsNotTheLeaderException` ("Changes to the schema must be executed on the leader server") instead of changing that follower alone (issues #9747, #9765). `Schema.setZoneId()` checks `UPDATE_DATABASE_SETTINGS`.

Action: call these methods on the leader, validate materialized view arguments before passing them, and expect `remove()` on an edge iterator to throw when the delete is refused.

26.10.2 - Client error handling: gRPC writes whose answer was lost, streamed error lines, Postgres transaction modes

- The gRPC Java client throws a non-retryable `RemoteException` when an admin write through `RemoteGrpcServer` or a streaming write (`ingestStream`, `ingestBidi`, `timeSeriesWriteStream`, graph batch loads) ends with a bare `UNAVAILABLE`, since the write may have been applied; reads and `validate_only` dry runs stay retryable. A client-side deadline is a `TimeoutException` whose message says the write may already have been applied (issues #8822, #9470).
- In production mode, the `message` of the `error` line of a streamed query is the classified label, like the `error` field the line now carries in every mode; the detail is in the server log (issues #8899, #9475).
- PostgreSQL: `SERIALIZABLE`, `READ ONLY` and `AND CHAIN` in `BEGIN` / `START TRANSACTION` / `COMMIT` / `ROLLBACK` are refused with SQLSTATE `0A000` instead of a syntax error (issues #9631, #9753).

Action: do not retry a write the client reports as possibly applied without checking its outcome, and branch on the `error` field or the status instead of the `message` text.

8.17.2. Single-server upgrade

1. Download and extract the new ArcadeDB release in a separate directory (do not overwrite the old install).
2. Stop the running server (or close every open database via the HTTP `close database` command).
3. Copy the directories listed in What to copy from the old install into the new install, preserving the original paths.
4. Start the new server.

When the server starts, every database is opened with the new code and any required format migration is applied in place.

8.17.3. What to copy

ArcadeDB's root directory contains a fixed set of folders. Only a few of them hold user-owned state; the rest are recreated from the release archive on every install.

Folder	Action	Notes
`databases/`	**Copy**	The actual database data. Every subdirectory is one database. This is the only folder strictly required for an upgrade.
`config/`	**Copy selected files**	The release ships a default `config/` directory. Carry over only the files you have actually modified — listing below.
`backups/`	Copy if you want history	Backup archives produced by manual or automatic backups. The server does not need them to start; copy them only if you want to keep the backup history reachable from the Studio Backup tab and from `RESTORE DATABASE` commands.
`log/`	Skip	Server log files. Safe to start with an empty `log/` directory; the release ships an empty one. Copy only if you want to keep historical logs alongside the new install.
`raft-storage-<peerId>/` (HA only)	**Copy**	Per-node Raft log segments, created at runtime when HA is enabled. This storage is **durable by default** (`arcadedb.ha.raftPersistStorage=true`) and must be preserved so a node can rejoin by replaying its log instead of a full snapshot resync. When `raftPersistStorage` is explicitly `false`, it is ephemeral and can be skipped. See HA cluster upgrade for details.
`replication/`	Skip	Legacy directory from the pre-Raft replication implementation. Not used by the current HA stack.
`lib/`, `bin/`	Use the new release as-is	Java artifacts and start scripts. Always take them from the new release — do not copy from the old install. The only exception is custom JARs you have dropped into `lib/` yourself; carry those over by hand.

Files to carry over from `config/`

The `config/` directory ships with a small set of defaults. You only need to copy the files that you have actually changed:

File	When to copy
`server-users.jsonl`	Always, unless you create users only via REST/SQL on every upgrade. This is where ArcadeDB persists local users and their per-database group assignments.
`server-groups.json`	If you have customised the security policy (custom groups, type-level permissions, result-set limits). See Security Policy.
`backup.json`	If you have configured the auto-backup scheduler (schedule, retention, target directory). See Automatic Backup.

File	When to copy
`arcadedb-log.properties`	If you have tuned `java.util.logging` settings (log levels, rolling file size, etc.).
`gremlin-server.yaml`, `gremlin-server.groovy`, `gremlin-server.properties`	Only if you use the embedded Gremlin Server and have edited any of these.

Everything else under `config/` (for example `arcadedb-log-all.properties`) ships unchanged with each release; take the new version.

ArcadeDB does **not** read a `config/server-configuration.json` file in normal operation. Server-wide settings come from JVM system properties, environment variables and the `arcadedb-server.sh` arguments — see Server Configuration.

8.17.4. HA cluster upgrade

An HA cluster (see High Availability) is upgraded one node at a time. The per-node procedure is the same as single-server above, with the following clarifications.

What to migrate on each node:

- **`databases/`** — copy, exactly as in the single-server case.
- **`config/`** — copy the same files as in the single-server case. The cluster's identity is derived from `arcadedb.ha.clusterName` and the `root` password (the inter-node token is computed from them at startup), so as long as every node keeps the same cluster name and the same root password, the upgraded cluster reattaches to itself without any extra step.
- **`raft-storage-<peerId>/`** — **copy**. The local Raft log is durable by default (`arcadedb.ha.raftPersistStorage=true`), so preserve this directory across the upgrade to let the node rejoin by replaying its own log. Whether or not it is preserved, after the upgraded node rejoins the cluster the leader replays any missing entries via Raft or, if the node has fallen behind the purge boundary, streams a full snapshot over HTTP automatically. With `raftPersistStorage=false` this directory is ephemeral and can be skipped.
- **`backups/` and `log/`** — same as single-server.

Recommended rolling upgrade flow (zero downtime, requires quorum at all times):

1. Pick a non-leader node first.
2. Stop the node, install the new release in a separate directory, copy the folders listed above, start the new binary.
3. Wait until `GET /api/v1/cluster` reports the node as healthy and caught up (replication lag near zero).
    a. The Studio Cluster tab shows the same information visually.
4. Repeat for every remaining replica.
5. Step the leader down with `POST /api/v1/cluster/stepdown` (or simply stop it — Raft will elect a new leader) and upgrade it last.

A node that receives replicated TIMESERIES data for a type it does not have — or for a type that exists locally but is not a TIMESERIES one — refuses the entry, which quarantines that database and triggers an automatic resync from the leader. Earlier releases logged the problem and carried on, which left the node's copy of that type permanently behind the leader's with nothing but a log line to say so.

This should not occur during an ordinary rolling upgrade, since the entry that creates a type is always applied before data for it arrives. It is a safeguard for the case where it does — for instance a mixed-version window in which one node's build cannot construct a type another node has. If you see a database quarantined with a message about a TimeSeries sealed store, the resync will restore it; complete the rollout so every node runs the same build.

For first-time HA bring-up with a pre-existing database (e.g. when you scale a single-node install into a cluster), see Offline Cluster Bootstrap.

8.17.5. Docker / Kubernetes

When you run ArcadeDB from the official container image, the release is replaced atomically by changing the image tag. The only data that survives across container restarts is what lives on a persistent volume, so:

- Mount `/home/arcadedb/databases` on a persistent volume — this is the equivalent of "copy the `databases/` folder".
- Mount `/home/arcadedb/config` only if you have edited the files listed in Files to carry over from `config/`; otherwise let the image ship its defaults.
- Set `arcadedb.ha.raftStorageDirectory` to a static path (e.g. `/home/arcadedb/raft-storage`) and mount **that** path on a persistent volume. The default per-node directory is named `raft-storage-<nodeName>` dynamically, which a container runtime or Kubernetes cannot mount as a volume (a wildcard like `/home/arcadedb/raft-storage-*` is created literally, not expanded) — this is exactly what `raftStorageDirectory` is for. The Raft log is durable by default (`arcadedb.ha.raftPersistStorage=true`) so a restarted pod can replay its own log instead of resyncing from peers. With `raftPersistStorage=false`, the log is ephemeral and this mount is unnecessary.
- Mount `/home/arcadedb/backups` if you want to retain backup archives across pod restarts.

For Kubernetes specifics (StatefulSet, headless service, init container for pre-staging a database), see Kubernetes.

8.17.6. Downgrade ArcadeDB

In case you need to downgrade to an older version of ArcadeDB, check the binary compatibility between the versions. ArcadeDB uses calendar versioning in the format `YY.M.S` (year, month, serial: for example `26.12.1` is the first release of December 2026), with 100% on-disk compatibility for migration of databases up or down between **serial** releases of the same year and month (the `S` in `YY.M.S`). For downgrades across months the safest path is to export the database with the newer

version and re-import it with the older version. See Versioning, Compatibility and Deprecation Policy for the full compatibility rules.

8.17.7. Downgrading a node of an HA cluster

A cluster replicates schema changes in a compact format that older builds cannot read. Those entries stay in the Raft log, so a node that is restarted on an older build replays them and ends up with a schema that no longer matches the rest of the cluster - silently.

A node of a cluster must not be rolled back to an older build by restarting it on its existing Raft log. If you have to go back, rebuild the node instead: stop it, delete its `raft-storage-<peerId>/` directory and its `databases/` folder, and start the older build so it rejoins empty and takes a fresh copy from the leader.

This applies only to the Raft log, not to your data. Database files themselves keep the usual serial-release compatibility described above.

8.17.8. Downgrading from 26.10.2 or later with hash indexes

Since v26.10.2 every hash index (`UNIQUE_HASH` or `NOTUNIQUE_HASH`) that is created or rebuilt is written at page layout version 3, which older servers do not know. A server that predates this layout refuses such an index with:

```
Hash index '<name>' has the page layout version 3, which this server does not support
(it knows 1 and 2). It was created by a newer version: open the database with that
version, or drop and recreate the index.
```

Hash indexes that 26.10.2 opened but did not rebuild keep their older layout and still open on an older server. Before going back, drop the hash indexes created or rebuilt by 26.10.2 (the layout version is part of the index file name) and recreate them once the database runs on the older server, or export the database and re-import it. Do not open a database holding a hash index created or rebuilt at layout 3 with a server that predates the layout check, because it does not verify the layout version: see Hash index.

8.18. Versioning, Compatibility and Deprecation Policy

This page states how ArcadeDB releases are numbered, what stays compatible from one release to the next, and how a public interface is deprecated before it is removed. For the list of changes that need action when you move to a given release, see Breaking changes.

8.18.1. Version numbers

ArcadeDB uses calendar versioning in the format `YY.M.S`:

Part	Meaning
`YY`	Two-digit year of the release (`26` = 2026).

Part	Meaning
`M`	Month of the release (`10` = October).
`S`	Serial number within that month. `1` is the monthly feature release; `2`, `3`, … are fix releases on the same line.

For example, `26.12.1` is the feature release of December 2026 and `26.12.2` is the first fix release on top of it. A new feature release ships every month.

8.18.2. On-disk compatibility

- **Upgrade**: a newer release opens a database created by any older release and migrates it to the newer on-disk format automatically, the first time it opens it. No export or import is needed. See Upgrade ArcadeDB.
- **Serial releases**: databases move up **and** down freely between serial releases of the same year and month (for example between `26.12.1` and `26.12.3`), with 100% on-disk compatibility.
- **Downgrade across months**: moving a database to an older monthly line is not guaranteed. Export the database with the newer release and import it with the older one. See Downgrade ArcadeDB.
- **HA clusters**: a cluster is upgraded node by node without a service outage. Any release-specific constraint on mixing versions or rolling back a node is listed in the upgrade guide (see HA cluster upgrade and Downgrading a node of an HA cluster).

8.18.3. Public interfaces covered

The compatibility and deprecation rules below apply to these public interfaces:

- The HTTP API under `/api/v1`. Within `v1`, changes are additive: new endpoints, new optional parameters and new response fields. Removing or renaming an endpoint, a parameter or a response field follows the deprecation process. An incompatible redesign would ship under a new version prefix, alongside `v1`.
- The public Java API (`com.arcadedb.database`, `com.arcadedb.graph`, `com.arcadedb.schema`, `com.arcadedb.query`, `com.arcadedb.remote` and the server plugin API).
- The query languages: SQL, openCypher, Gremlin, GraphQL, the MongoDB query language and the Redis commands, including their built-in functions and procedures.
- The wire protocols (PostgreSQL, Neo4j Bolt, MongoDB, Redis, gRPC). These follow the external protocol specification they implement; support for a protocol version is not removed without deprecation.
- Configuration settings (`arcadedb.*`) and their documented defaults.

Internal classes, undocumented behavior, log message text and the on-disk file layout are not public interfaces.

8.18.4. Deprecation process

When a public interface is going to be removed or changed incompatibly:

1. The deprecation is announced in the release notes of the release that introduces it, together with the replacement to use.
2. The interface is marked deprecated in this documentation and, for the Java API, with `@Deprecated` and a pointer to the replacement in its Javadoc. A deprecated setting keeps being accepted.
3. The deprecated interface keeps working for **at least 6 months**, that is at least six monthly feature releases after the one that announced the deprecation.
4. The removal is listed under Breaking changes for the release that performs it.

8.18.5. Exceptions: security and correctness fixes

A fix for a security vulnerability, a data-integrity problem, or a result that does not conform to the specification of a query language (for example a Cypher query returning a different answer than the openCypher specification requires) can change behavior without a deprecation period. Every such change that can require action is listed under Breaking changes for the release that ships it, with what changed and what to do. Security fixes are also published as GitHub security advisories.

8.18.6. Which releases receive fixes

Fixes ship as serial releases (`YY.M.2`, `YY.M.3`, …) on the current monthly line and are included in the next monthly feature release. Customers with an ArcadeDB commercial support subscription can request backports to an older monthly line, according to the terms of their subscription.

8.18.7. License commitment

ArcadeDB is and will always be licensed under the Apache License 2.0, as stated in the project's governance and license guarantee. Every release made under Apache 2.0 stays available under Apache 2.0.

Chapter 9. Security

9.1. Users

Users are stored in the file `config/server-users.jsonl` file. The JSONL format means one json per line. When the server starts always checks if there are any users configured. If the user file is empty, the `root` user is created with a password the user must enter in the console where the server is starting. Example:

```
+--------------------------------------------------------------------+
|                WARNING: FIRST RUN CONFIGURATION                    |
+--------------------------------------------------------------------+
| This is the first time the server is running. Please type a        |
| password of your choice for the 'root' user or leave it blank      |
| to auto-generate it.                                               |
|                                                                    |
| To avoid this message set the environment variable or JVM          |
| setting `arcadedb.server.rootPassword` to the root password to use.|
+--------------------------------------------------------------------+

Root password [BLANK=auto generate it]: ************
Please type the root password for confirmation (copy and paste will not work):
************
```

Example of `config/server-users.jsonl` file:

```
{"name":"root","password":"PBKDF2WithHmacSHA256$65536$hcv0joKV/o/q+KOVmcwNUqhEq1w2/j8O
VnEkkVjzkeg=$2q2u4rjUlJjgoKBX9sG0rV0bOh6aHo+RhHsOkXneGkM=","databases":{"*":["admin"]}
}
```

In the users file the following information are stored per user:

- Name, mandatory
- Password. It is always saved hashed by using the algorithm `PBKDF2` with a configurable salt (default = 32). The password is mandatory for all the users, but root. In the case root has no password, then ArcadeDB server asks to insert a password at startup (see above).
- Databases, as the map database name and set of groups for that database. "*" is a special wildcard and means any. The configuration `"databases":{"*":["admin"]}` means use the "admin" group for any database.

Replication of user changes (clusters). Creating, updating and deleting a user is replicated to every server of the cluster whichever way it is done: the `/api/v1/server/users` endpoints (which Studio uses), the `create user` / `drop user` server commands, and the Cypher `CREATE USER`, `ALTER USER … SET PASSWORD` and `DROP USER` statements.

> Deleting a user or changing its password also revokes that user's active login tokens immediately, on every node - see Token-Based Authentication.
>
> Each server also keeps `config/server-security-cluster.json` beside the user, group and API-token files. It holds only a checksum of the last security document that server received from the cluster, which is what lets it tell a change it has already applied from one built on an out-of-date view. Keep it with the rest of the configuration directory; if it is lost the server rebuilds it on the next security change. Since 26.11.1, if the server cannot write it, it keeps the previous checksum and the change stays in the Raft log until the write succeeds, so a restart replays the change instead of judging the next one against an out-of-date checksum.
>
> If, after upgrading from an earlier version, a user exists on only some nodes, reissue the change once and the cluster converges.
>
> If a server cannot write `config/server-users.jsonl` - a full or read-only volume - it logs a SEVERE error and keeps running, applying the change in memory so it takes effect there straight away. Since 26.11.1 the save is retried while the server keeps applying cluster changes, and until it succeeds the change stays in the Raft log, so a restart replays it rather than reverting to the previous users. The same holds for groups and API tokens. The Raft log is not compacted past such a change while the volume stays unwritable, so fix the volume. Before 26.11.1 the save was not retried: after fixing the volume the user change had to be reissued, or that server reverted to the previous users on its next restart.

Reloading the security files. *(Since v26.11.1)* The users (`server-users.jsonl`), groups (`server-groups.json`) and API-token files share one reload policy and one watcher implementation, polled every `arcadedb.server.reloadEvery` milliseconds (the users watcher is a daemon thread named `arcadedb-security-users-watcher[<path>]`). Setting `reloadEvery` to 0 or less starts no watcher.

- When a running server cannot load one of the files (a malformed line, a record that is not a valid user, an empty, missing or unreadable file, a wrong shape), it keeps the users, groups or tokens already in force, logs a SEVERE message, and saves a readable bad file as an `-error` copy (`server-users-error.jsonl`, `server-groups-error.json`, or the API-token file's error copy). It never writes over the operator's file. The watcher applies the next corrected edit as usual. Before, a bad users file was replaced by a passwordless `root` user, other users' sessions were revoked, and a bad groups file was replaced by the default groups.
- At startup there are no previous users to keep: a parse or I/O error in `server-users.jsonl` keeps the file as `server-users-error.jsonl` and installs the default `root` user, so a server restarted before the file is fixed starts with only `root`. A `server-groups.json` without `version` or without a `databases` object is kept as `server-groups-error.json` before the default groups are installed. An API-token file without a `tokens` key is refused and kept as an error copy.
- The API-token file has its own watcher, and a users-file reload no longer clears the tokens.
- When a replicated change (a dropped user, a rotated password, a deleted group) could not be written to its file, the watcher keeps the replicated state in force and logs a WARNING instead of re-reading the older file. A reload that read the users file just before a concurrent replicated change was applied is dropped with a WARNING telling the operator to redo a hand edit

through the server.

ArcadeDB allows each user to belong to zero or multiple groups. If no groups are defined, the default setting for the group * are used.

The following configuration defines "Jay" user to belong to "BlogWriters" in "Blog" database and "Editors" in the "Library" database:

```
{
  "databases": {
    "Blog": ["BlogWriters"],
    "Library": ["Editors"]
  }
}
```

The declaration above implicitly assigns Jay to the default group for any other database configured. The default configuration for the default group is no access, where the user cannot read or write in the database.

9.1.1. Via Console

Alternatively, users can be created via the console, see `Create User` Console Command and Server Command.

9.1.2. Via Cypher

Users can also be managed using Cypher commands when running in server mode:

```
SHOW USERS
CREATE USER bob SET PASSWORD 'SecurePass123!'
ALTER USER bob SET PASSWORD 'NewPass456!'
DROP USER bob
```

See Cypher User Management for the full reference.

9.1.3. Password hash format

The recommended way to create or update a user's password is through the HTTP API (`POST /api/v1/server/users`), the Console (`CREATE USER`), or Cypher (`CREATE USER … SET PASSWORD …`). All three paths produce a correctly-formatted hash for you. Hand-editing `config/server-users.jsonl` is supported but error-prone; the rest of this section only matters if you are generating the hash yourself (for example from a provisioning script).

Each password is stored as a single string with four dollar-separated fields:

```
<algorithm>$<iterations>$<salt-base64>$<derived-key-base64>
```

Example:

```
PBKDF2WithHmacSHA256$65536$hcv0joKV/o/q+KOVmcwNUqhEq1w2/j8OVnEkkVjzkeg=$2q2u4rjUlJjgoK
BX9sG0rV0bOh6aHo+RhHsOkXneGkM=
```

Field meaning:

Field	Description
`algorithm`	Default `PBKDF2WithHmacSHA256`. Configured by `arcadedb.server.securityAlgorithm`.
`iterations`	Default `65536`. Configured by `arcadedb.server.saltIterations`.
`salt-base64`	32 random bytes, Base64-encoded. See the note below for the subtle detail.
`derived-key-base64`	The 256-bit (32-byte) PBKDF2 output, Base64-encoded.

When verifying a password, the server feeds PBKDF2 the **UTF-8 bytes of the Base64 salt string**, not the raw 32 salt bytes. In other words, the text that appears between the second and third `$` characters is taken verbatim and hashed with the user-supplied password. Any external tool that generates this hash must do the same, otherwise the stored hash will never match at login time and the user will be rejected with `401 Authentication failed` (before v26.11.1: `403 User/Password not valid`).

This is an implementation detail of the current storage format, preserved for compatibility with existing user files. It is not a security weakness, only a thing to be aware of when generating hashes outside the server.

The following Python snippet produces a hash that the server accepts:

```python
import base64, hashlib, secrets

def hash_password(plaintext: str, iterations: int = 65536) -> str:
    raw = secrets.token_bytes(32)
    salt_b64 = base64.b64encode(raw).decode("ascii")
    derived = hashlib.pbkdf2_hmac(
        "sha256",
        plaintext.encode("utf-8"),
        salt_b64.encode("utf-8"),   # the Base64 string, NOT the raw bytes
        iterations,
        dklen=32,
    )
    derived_b64 = base64.b64encode(derived).decode("ascii")
```

```
    return f"PBKDF2WithHmacSHA256${iterations}${salt_b64}${derived_b64}"
```

9.2. Groups

If a user has not assigned group in a database, the default group "*" is taken. The wildcard "*" represents all the groups that are not defined in this configuration. By default, such a group has no access to the database in read and write. Below you can find the default configuration for the default group "*".

```
{ "*": {
  "types": {"*": {"access": []}},
  "access": [],
  "readTimeout": -1,
  "resultSetLimit": -1
} }
```

Where:

- `types` is the map of type and access level. The wildcard "*" represents all the types that are not defined in this configuration.
- `access` is the array containing the allowed permissions for the group. The supported permission at group level are:
  - `updateSecurity`: to update the security settings (create, modify and delete users, groups, etc.). This also gates defining a JavaScript function via `DEFINE FUNCTION ⋯ LANGUAGE js` — see User Functions. `BACKUP DATABASE`, `EXPORT DATABASE` and `IMPORT DATABASE` require it too, because they work on the whole database regardless of the per-type permissions and read or write files on the server, so the `crud` and `readonly` groups cannot run them.
  - `updateSchema`: to update the database schema (create, modify and drop buckets, types and indexes, including every `ALTER TYPE` form and the `REBUILD TYPE` and `REBUILD INDEX` maintenance commands). *Since v26.12.1* it also gates the repairs of `CHECK DATABASE`, `FIX` and `COMPRESS`, so the `crud` and `readonly` groups can run `CHECK DATABASE` only to report
  - `updateDatabaseSettings`: to change per-database settings — e.g. `ALTER DATABASE`, `ALTER TYPE ⋯ BUCKETSELECTIONSTRATEGY`, materialized-view / continuous-aggregate refresh and downsampling policies. The LOAD CSV access settings (`arcadedb.opencypher.loadCsv.allowRemoteUrls`, `allowFileUrls`, `importDirectory` and `blockedIpRanges`) are security controls, so changing them with `ALTER DATABASE` also requires `updateSecurity`.
- `readTimeout` if present, specify the maximum timeout for read operations. -1 means no limits. If set, all the read operations (lookups and queries) will be limited to maximum `<readTimeout>` milliseconds. This is useful to limit users to execute expensive commands and queries impacting the performance of the server and therefore other connected users.
- `resultSetLimit` if present, specify the maximum number of entries in the result set returning from a command or query. -1 means no limits. If set, any query or command will be interrupted when this limit is reached. This is useful to limit users to retrieve huge result sets impacting the

performance of the server and therefore other connected users. A `LIMIT` (or `TIMEOUT`) in the statement that is lower than the group's limit is kept, also when the `LIMIT` is a parameter (`LIMIT :n`, `LIMIT ?`). Both limits apply only to the users of the group: other users running the same statement are not affected.

You can profile the access of each group up to the type level.

- `createRecord`, allows creating new records
- `readRecord`, allows reading records
- `updateRecord`, allows updating records
- `deleteRecord`, allows deleting records

creating an edge is technically 2 operations: (1) create a new edge record and (2) update the vertices with the reference. For this reason, if you want to allow a user to create edges, you have to grant the `createRecord` permission on the edge type and `updateRecord` on the vertex type.

(Since v26.10.2) the same applies to a `LIGHTWEIGHT` edge type (see Lightweight Edges), even though it writes no edge record: creating such an edge needs `createRecord` on its edge type, deleting one needs `deleteRecord`, and deleting a vertex that holds lightweight edges needs `deleteRecord` on their edge type.

Example of the definition of the group for a Blog writer, where he can only read from the "Blog" type and have full access to the "Post" type:

```
{
  "types": {
    "*": {
      "access": []
    },
    "Blog": {
      "access": [
        "readRecord"
      ]
    },
    "Post": {
      "access": [
        "createRecord",
        "readRecord",
        "updateRecord",
        "deleteRecord"
      ]
    }
  }
}
```

The default settings for the `admin` group are:

```
{
  "access": [
    "updateSecurity",
    "updateSchema",
    "updateDatabaseSettings"
  ],
  "resultSetLimit": -1,
  "readTimeout": -1,
  "types": {
    "*": {
      "access": [
        "createRecord",
        "readRecord",
        "updateRecord",
        "deleteRecord"
      ]
    }
  }
}
```

Which allows to execute any operation against the security, the schema, the database settings and records.

When ArcadeDB opens a `server-groups.json` written before the `updateDatabaseSettings` permission existed, it automatically migrates existing `admin` groups to include it, so upgraded clusters keep full administrative access without a manual edit.

Here is an example for an append-only group:

```
"appendonly": {
  "access": [],
  "resultSetLimit": -1,
  "readTimeout": -1,
  "types": {
    "*": {
      "access": [
        "createRecord",
        "readRecord"
      ]
    }
  }
}
```

Which allows the group members to read and create, but not to update or delete records. Such a group can be useful for ledgers, block chains, or data provenance.

You can use any JSON editor to edit the file `config/server-groups.json`. It's recommended to keep a

copy of the current file before editing the groups. In this way if there are any errors, it's easy to restore the previous file.

A change to a group takes effect immediately for users that are already connected, whether it is made through the `/api/v1/server/groups` endpoints or by editing `config/server-groups.json`. This covers the database-level permissions (`updateSchema`, `updateSecurity`, `updateDatabaseSettings`) and the `resultSetLimit` / `readTimeout` limits too: no server restart is needed.

`GET /api/v1/server/groups` (root only) answers `{"result": <groups document>}`, with `databases` mapping each database name to its `groups`. *(Since v26.11.1)* Every group in the listing carries `resultSetLimit`, `readTimeout`, `access` and `types`: a group stored without one of them, such as the default `crud` group of older installs, is listed with `resultSetLimit` and `readTimeout` at `-1`, `access` as an empty array and `types` as an empty object, so strict clients such as the Python driver's `list_groups` can parse it. The stored document is not changed, and the shipped `server-groups.json` now sets `readTimeout: -1` on `crud`.

Since version 26.10.2, the same applies to the user itself on the PostgreSQL, Bolt, MongoDB and Redis protocols: if a user is deleted, its database access is removed or its password is changed, the connections it already had open are refused on their next request, as on HTTP. Open Bolt transactions of that user are rolled back.

9.3. Security Policy

The default rules of security are pretty basic. The username must be between 4 and 256 characters. The password length must be between 8 and 256 characters. You can implement your own security policy by providing your own implementation:

```
server.getSecurity().setCredentialsValidator( new DefaultCredentialsValidator(){
  @Override
  public void validateCredentials(final String userName, final String userPassword)
throws ServerSecurityException {
    if( userPassword.equals("12345678") )
      throw new ServerSecurityException("Guess who was not attending security
lesson!");
  }
});
```

9.4. Handling Secrets

Passing secrets, like passwords, safely to the server, especially in containers, can be tricky. For example, the settings:

```
-Darcadedb.server.rootPassword=password
```

for the root password, or:

```
-Darcadedb.server.defaultDatabases=database[username:password]
```

for database and user generation are comfortable, but present the issue that passwords passed this way are readable in plaint text in applications like `htop` or `docker compose top`. This is the case because these settings are passed ultimately via command-line to the JVM (even if using `JAVA_OPTS` or `ARCADEDB_SETTINGS`).

In the following, an approach is presented to avoid this problem. The idea is to use the special environment variable `JAVA_TOOL_OPTIONS` to pass the settings directly into the JVM (alternatively `JDK_JAVA_OPTIONS` or the undocumented `_JAVA_OPTIONS` may be used). However, just using this particular environment variable would mean that the JVM writes the contents of this variable to the standard error stream, which means the secrets would appear in logs.

For the root password this can be avoided using the setting:

```
-Darcadedb.server.rootPasswordPath=/path/to/root_secret
```

which passes the secret via a plain text file which is mounted to a known path inside the container. This is a common practice to pass secrets to containers, for example with Docker compose secrets or a Kubernetes `Secret` mounted as a volume. A trailing newline, which most editors and `echo` append, is not treated as part of the password, and an empty file is rejected with an error naming it rather than silently accepted.

The same file-based approach is available for the High Availability cluster token using:

```
-Darcadedb.ha.clusterTokenPath=/path/to/cluster_token
```

It is read only when `arcadedb.ha.clusterToken` is not set, and the file content is trimmed of surrounding whitespace, so a trailing newline added by most editors or `echo` does not break the token comparison between nodes.

To suppress this logging for non-root secrets, a workaround can be used by providing the secret also file-based and wrapping the `server.sh` script:

```
#!/bin/sh

export JAVA_TOOL_OPTIONS="-Darcadedb.server.defaultDatabases=database1[user1:`cat
/path/to/user_secret`]"
./bin/server.sh 2> >(grep -v "^Picked up JAVA_TOOL_OPTIONS:") &

PID="$!"

unset JAVA_TOOL_OPTIONS

wait $PID
```

This way the secrets are passed directly into the JVM without being resolved before, and the `stderr` logging is filtered. The server is send to the background and its process ID saved. Then, by unsetting the `JAVA_TOOL_OPTIONS` variable immediately after using it, ensures the secrets will not linger in the environment.

9.5. Gremlin Security

ArcadeDB provides two Gremlin execution engines with different security profiles.

CRITICAL SECURITY NOTICE: The Groovy Gremlin engine is **NOT SECURE** and should **NOT be used in production environments** or with untrusted users. Despite security restrictions, it remains vulnerable to remote code execution (RCE) attacks. Always use the Java engine (default) for production deployments.

9.5.1. Execution Engines

Java Engine (Default - Secure)

The native Java engine parses Gremlin queries using a restricted grammar and generates bytecode. This engine:

- Only accepts valid Gremlin traversal syntax
- Cannot execute arbitrary Java code
- Blocks `Runtime.getRuntime().exec()` and similar attacks
- **Recommended for ALL deployments**
- **Required for production and security-critical environments**

Groovy Engine (Legacy - INSECURE)

SECURITY VULNERABILITY: The Groovy engine is vulnerable to Remote Code Execution (RCE) attacks. Authenticated users can execute arbitrary operating system commands. **DO NOT USE IN PRODUCTION.**

The Groovy engine provides dynamic scripting capabilities but has critical security limitations:

- Supports Groovy-specific syntax and dynamic features
- **Cannot be adequately secured** - `SecureASTCustomizer` has known bypasses
- Vulnerable to: `Runtime.exec()`, `ProcessBuilder`, file system access, reflection
- **Only use for development/testing in fully trusted environments**
- Will display a security warning at startup

9.5.2. Who Can Evaluate Groovy

Evaluating Groovy is running arbitrary code on the server host, so its reach is the whole server, not one database. ArcadeDB reserves it to the server administrator (`root`) on every entry point. A per-

database grant, including a database administrator holding `updateSecurity`, is not enough:

- **HTTP command endpoint** (`POST /api/v1/command/{database}` with `"language": "gremlin"`): when the database runs the `groovy` engine, or the `auto` engine falls back to Groovy, any other user gets `403 Forbidden`. Under the default `java` engine nothing changes: every authorized user keeps running traversals.
- **Gremlin Server** (port 8182): a string script submitted in a language other than `gremlin-lang` is refused for any other user. A script with no language is evaluated by TinkerPop as `gremlin-groovy`, so it is refused too. The same applies to a bytecode traversal carrying a lambda (e.g. `Lambda.function(…)`), whose body is Groovy source. Bytecode traversals without lambdas, and string scripts submitted with language `gremlin-lang`, stay available to every authorized user:

```
client.submit("g.V().count()", RequestOptions.build().language("gremlin-lang"
).create());
```

The `io()` step (`g.io('/path/file.xml').read()` / `.write()`) reads or writes a file at a caller-chosen path on the server host, so it is reserved to `root` as well, on every engine including `java`. The restriction cannot be lifted from the request: a `withoutStrategies(…)` naming the strategy that enforces it is ignored. Use EXPORT DATABASE and IMPORT DATABASE to move data in and out of a database.

In embedded mode no user is bound to the request, so none of these checks apply.

9.5.3. Groovy Scripts Are Always Treated as Writes

A Groovy script is code: only running it tells what it does, and it can write through any API it reaches (`g.getGraph().addVertex(…)`, the database itself), not only through traversal steps. Since 26.11.1 ArcadeDB never runs a script to find out whether it is read-only: a script the `gremlin-lang` grammar cannot read (so one the `auto` or `groovy` engine runs as Groovy) is treated as one that may do anything, a schema change or an administrative operation included. In practice:

- the read-only query entry points (`GET`/`POST /api/v1/query`, `database.query("gremlin", …)`, the MCP `query` tool, a chart drawn by the Studio AI assistant) refuse it as not idempotent, without running it. Send it as a command (`POST /api/v1/command`) instead;
- the MCP `execute_command` tool allows it only when the MCP configuration allows every operation, schema changes and administrative operations included;
- on an HA follower it is always forwarded to the leader;
- the streaming encoding refuses it, as it refuses any statement that may write.

A script written as a plain traversal (`g.V().has('name', 'a').values('age')`) is read by the grammar on every engine and is classified exactly, as before. Note that a traversal can fall back to Groovy only because of where it places a parameter: `__.constant(p1)` is not a position the `gremlin-lang` grammar accepts a variable in, so on the `auto` engine such a read runs as Groovy and has to be sent as a command.

9.5.4. Configuration

Set the engine via database configuration:

```
-- Secure (default)
ALTER DATABASE `arcadedb.gremlin.engine` 'java';

-- Legacy with security restrictions (use only if needed)
ALTER DATABASE `arcadedb.gremlin.engine` 'groovy';

-- Auto-detect (not recommended for production)
ALTER DATABASE `arcadedb.gremlin.engine` 'auto';
```

Or via JVM startup parameters:

```
# Set to Java engine (secure)
java -Darcadedb.gremlin.engine=java ...

# Set to Groovy engine (use only if needed)
java -Darcadedb.gremlin.engine=groovy ...
```

9.5.5. Security Best Practices

1. **Use Java Engine**: **ALWAYS** use `'java'` engine in production - this is non-negotiable for security
2. **Never Use Groovy in Production**: The Groovy engine is vulnerable to RCE and cannot be secured
3. **Least Privilege**: Grant Gremlin query permissions only to fully trusted administrators
4. **Input Validation**: Never accept Gremlin queries from untrusted sources
5. **Audit Logging**: Monitor all Gremlin query execution for suspicious patterns
6. **Network Isolation**: Run ArcadeDB in isolated network segments
7. **Authentication Required**: Always require authentication for Gremlin access

9.5.6. Security Limitations (Groovy Engine)

Despite attempts to restrict dangerous operations, the Groovy engine remains vulnerable to:

- **Process execution**: `Runtime.exec()`, `ProcessBuilder` - **NOT BLOCKED**
- **File system access**: `File`, `FileInputStream`, `Files`, `Paths` - **NOT BLOCKED**
- **Reflection**: `Method`, `Field`, `Constructor`, `Class.forName()` - **NOT BLOCKED**
- **Class loading**: `ClassLoader`, `GroovyClassLoader` - **NOT BLOCKED**
- **Network operations**: `URL`, `URLConnection`, `Socket` - **NOT BLOCKED**
- **System operations**: `System.exit()`, `System.load()` - **NOT BLOCKED**

The `SecureASTCustomizer` has inherent limitations that prevent effective sandboxing of Groovy scripts.

9.5.7. Migration from Groovy to Java Engine

If you are currently using the Groovy engine and want to migrate to the more secure Java engine:

1. **Test your queries**: Run your existing Gremlin queries with the Java engine to verify compatibility
2. **Update configuration**: Change the `arcadedb.gremlin.engine` setting to `'java'`
3. **Refactor if needed**: If you have Groovy-specific syntax, refactor to standard Gremlin traversal syntax
4. **Verify results**: Ensure query results are consistent between engines

Most standard Gremlin queries work identically on both engines. The Java engine is faster and more secure.

9.5.8. Security Considerations

CRITICAL: Remote Code Execution Vulnerability

The Groovy engine has a **CRITICAL SECURITY VULNERABILITY** that allows Remote Code Execution (RCE):

- **Vulnerability**: Authenticated users can execute arbitrary operating system commands
- **Impact**: Complete server compromise - attackers can read files, execute processes, access network
- **Mitigation**: The `SecureASTCustomizer` security restrictions are **INSUFFICIENT** and can be bypassed
- **Recommendation**: **NEVER use Groovy engine in production or with untrusted users**

Example of vulnerable code that executes successfully:

```
Runtime.getRuntime().exec("touch /tmp/owned")
new File("/etc/passwd").text
```

The Java engine (default) is **NOT vulnerable** to these attacks.

Default Engine: the default Gremlin engine is `'java'`, not `'auto'`, for security reasons. This protects users from RCE attacks.

If you explicitly enable the Groovy engine (`arcadedb.gremlin.engine=groovy`), you will receive a **security warning** at startup, and your system will be vulnerable to

RCE attacks.

Error messages in production mode: when the server runs with `arcadedb.server.mode=production`, the Gremlin Server (WebSocket, default channelizer) answers engine failures with the generic message "The request failed. Check the server log for the details" and no stack trace, so stored key values are not sent to the client. The real message is written to the server log.

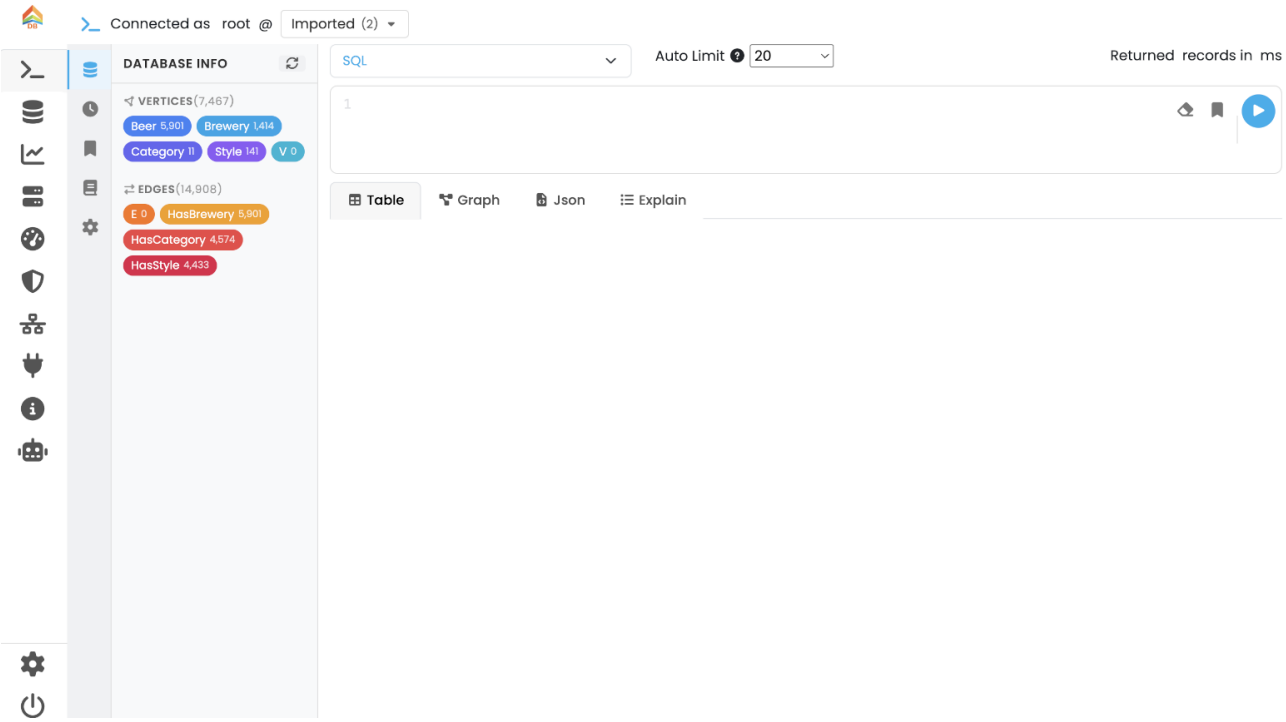
Other channelizers (TinkerPop's `HttpChannelizer`, `WsAndHttpChannelizer`, `UnifiedChannelizer`, or a custom class) cannot conceal the engine's text. Since v26.10.2, with `arcadedb.server.mode=production` a non-default channelizer, selected by `channelizer:` in `gremlin-server.yaml` or by the `gremlin.channelizer` setting, makes the server fail to start, and a runtime switch of `arcadedb.server.mode` to `production` is refused while the Gremlin Server runs with one. Outside production mode the channelizer is used as configured and the server logs a WARNING at startup. To fix a refused start, remove the `channelizer` entry or set it to `org.apache.tinkerpop.gremlin.server.channel.WebSocketChannelizer`. See also Error details in production mode.

Chapter 10. Tools

10.1. Studio

ArcadeDB Studio is the web-based workbench that ships with every ArcadeDB server. You can open it by pointing a browser at the server and port that host the ArcadeDB server (in "development" or "test" mode); for a default local install this is http://localhost:2480 .

(Since v26.10.2) Studio's pages are served with a `Content-Security-Policy` that refuses inline script, so an HTML-injection bug cannot run script with the signed-in user's token. The policy is set by `arcadedb.server.httpContentSecurityPolicy`; the default also allows Google Fonts and `raw.githubusercontent.com` (the sample datasets list), and an empty value sends no policy. A reverse proxy or a browser extension that injects inline script into Studio's pages is blocked by it: allow that script in a custom policy, or try the policy first with `arcadedb.server.httpContentSecurityPolicyReportOnly=true`, which only reports violations in the browser console.



10.1.1. Main menu

The vertical icon bar on the left is the main navigation. Each icon opens one tab in the main canvas:

Icon	Tab	Needs a selected database
□	Query — write and execute commands in SQL, SQL Script, Open Cypher, Apache TinkerPop Gremlin, GraphQL, MongoDB, Redis or PromQL, or search a vector, hybrid or full-text index; visualise results as Table, Graph, Chart, JSON or Explain	Yes
□	Database — manage schema, buckets, indexes, dictionary, time series (Time Series sub-tab), metrics, settings and per-database backups	Yes
□	Server — server health summary, metrics, sessions, events, server-wide settings, auto-backup and MCP configuration	No
□	Profiler — record and analyse query execution across protocols, save runs, optionally analyse with AI	No
□	Security — manage users, user groups (roles) and API tokens	No
□	Cluster — replication topology, leader election, peer add/remove	No
□	API — browse and try the HTTP API endpoints from an interactive playground	No
□	Info — quick links into this documentation	No
□	AI Assistant — natural-language assistant for schema, queries and data modelling (paid add-on)	Yes
□	Settings — Studio appearance (light/dark/system theme)	No

The current user and database selector are shown at the top of every tab that operates on a database. Tabs that do not need a database (Server, Profiler, Security, Cluster, API, Info, AI, Settings) are enabled as soon as you sign in.

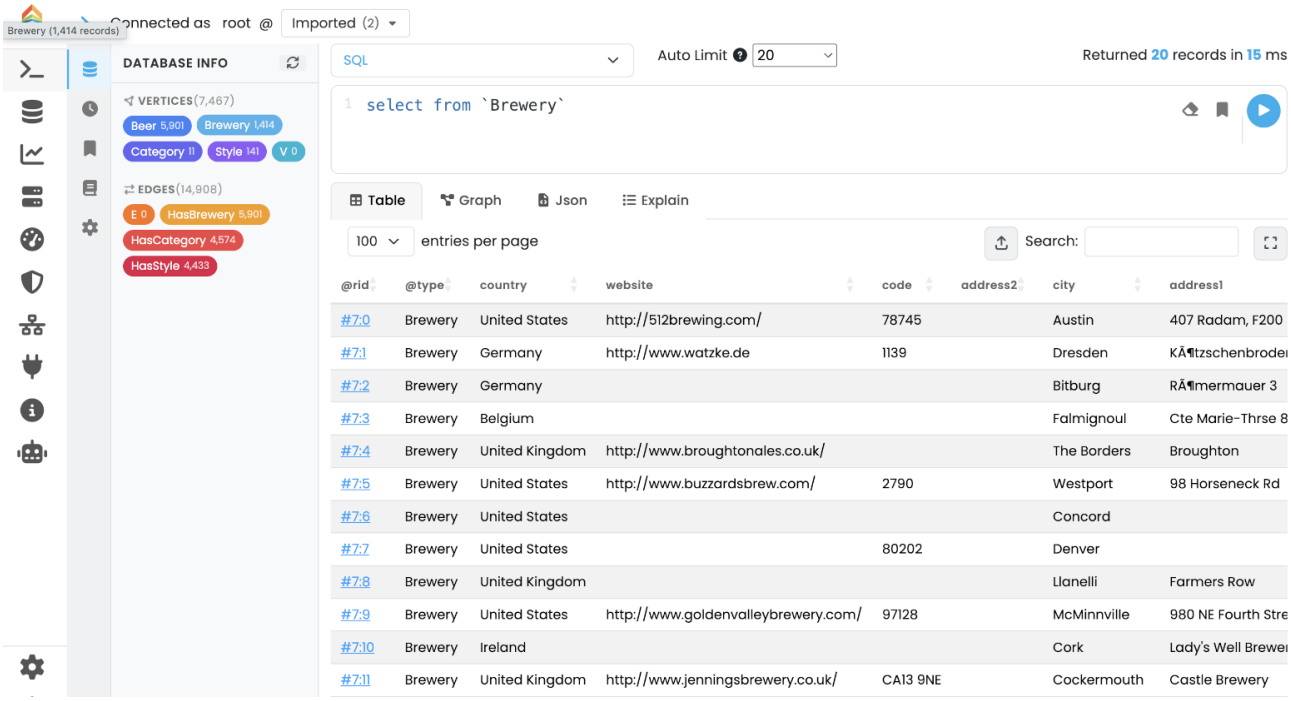
10.1.2. Login and themes

Studio uses the same users as the server (see Security). The theme (Light, Dark or System) is set from the Settings tab and persists across sessions.

A **Logout** button sits at the very bottom of the icon sidebar.

10.2. Query Panel

The **Query** panel is the everyday workspace: write a command in any supported language, run it against the selected database, and inspect the result as a table, a graph, a chart, a JSON document, or an execution plan.

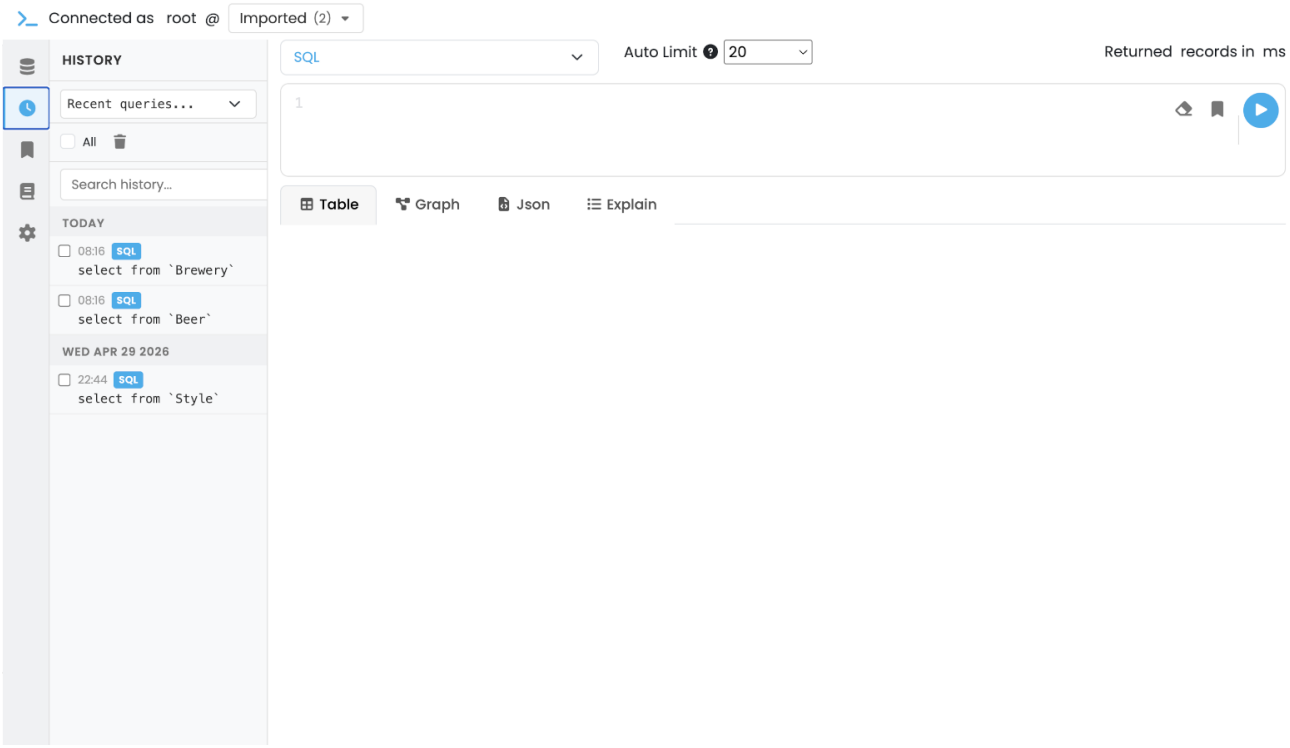


The header shows the current user and the **Database Selector**; pick the database to run against here. The status bar at the top right reports `Returned X records in Y ms` after every execution.

10.2.1. Left icon sidebar

The narrow icon column on the left of the editor switches between auxiliary panes; the main editor and results stay in place.

- **Overview** — summary information about the selected database (default view). It has the same **Filter types...** box as the Schema sub-tab, to find one type among many *(since v26.12.1)*.
- **History** — list of recently executed commands. Click an entry to load it back into the editor.
- **Saved** — saved queries. Use the **Save Query** button in the editor toolbar to add the current command to this list.
- **Reference** — searchable reference of SQL functions and methods plus quick links to the other query languages.
- **Settings** — editor preferences (theme, font size, key bindings).

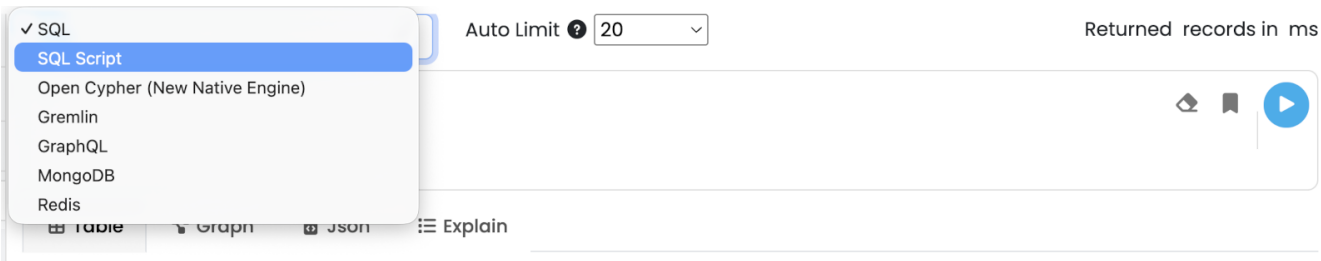


10.2.2. Editor and toolbar

The editor is a CodeMirror instance with syntax highlighting that follows the selected language.

The toolbar above the editor exposes:

- **Language** selector with: **SQL**, **SQL Script** (multi-statement), **Open Cypher** (native engine), **Apache TinkerPop Gremlin**, **GraphQL**, **MongoDB**, **Redis**, **PromQL (Time Series)**, and under **Index search**: **Vector search**, **Hybrid search**, **Full-text search** (see Index search).
- **Auto Limit** dropdown — caps the number of rows returned by interactive runs (`20`, `50`, `100`, `500`, **No Limits**). Useful to stay safe on large datasets; the cap is applied as a `LIMIT` when the language supports it.
- **Clear** — empties the editor.
- **Save Query** — adds the current command to the Saved pane in the sidebar.
- **Run** (or kbd:[Ctrl+Enter] / kbd:[Cmd+Enter]) — executes the command.



10.2.3. Index search (Vector, Hybrid, Full-text)

The last three entries of the **Language** selector query an index instead of running a command. Selecting one replaces the editor with a small form, while **Auto Limit** is hidden because the form carries its own `k` or limit. **Run**, **Clear**, **Save Query**, the history and the result views (Table, Json,

Explain) work as for any other language.

- **Vector search** - pick an `LSM_VECTOR` or `LSM_SPARSE_VECTOR` index, enter the query vector (`[0.12, -0.5, 0.33]` or `0.12, -0.5, 0.33`), `k`, an optional SQL `WHERE` filter, and for a dense index `efSearch`. For a sparse index you can also enter the sparse dimension ids.
- **Hybrid search** - the vector inputs above plus a full-text leg (index and Lucene query), the fusion strategy, per-leg weights and an optional graph expansion (edge types, direction, max depth).
- **Full-text search** - pick a `FULL_TEXT` index, enter a Lucene query (for example `+java -python`) and a limit.

The form checks what it can before sending: a vector whose length differs from the index dimensions is refused without a round trip, and the limits for `k`, `efSearch`, `limit` and `maxDepth` are read from the server, so they always match the running version. The index list is read from the database schema; a database without a vector or full-text index shows a note instead.

Each hit is one table row: `@rid`, `@type`, then the ranking value (`distance`, `score` or `fusedScore`, and for hybrid the `sources`), then the record properties. The **Json** tab holds the raw response, and the **Explain** tab summarizes which index answered, how it ranks, and whether the result window was filled. When the count reads **(truncated)**, more matches exist: raise `k`.

The form runs the same operations as the HTTP endpoints `POST /api/v1/vector/{database}/search`, `/hybrid` and `/fulltext`. These searches used to live in a separate **Vector** tab, which was removed.

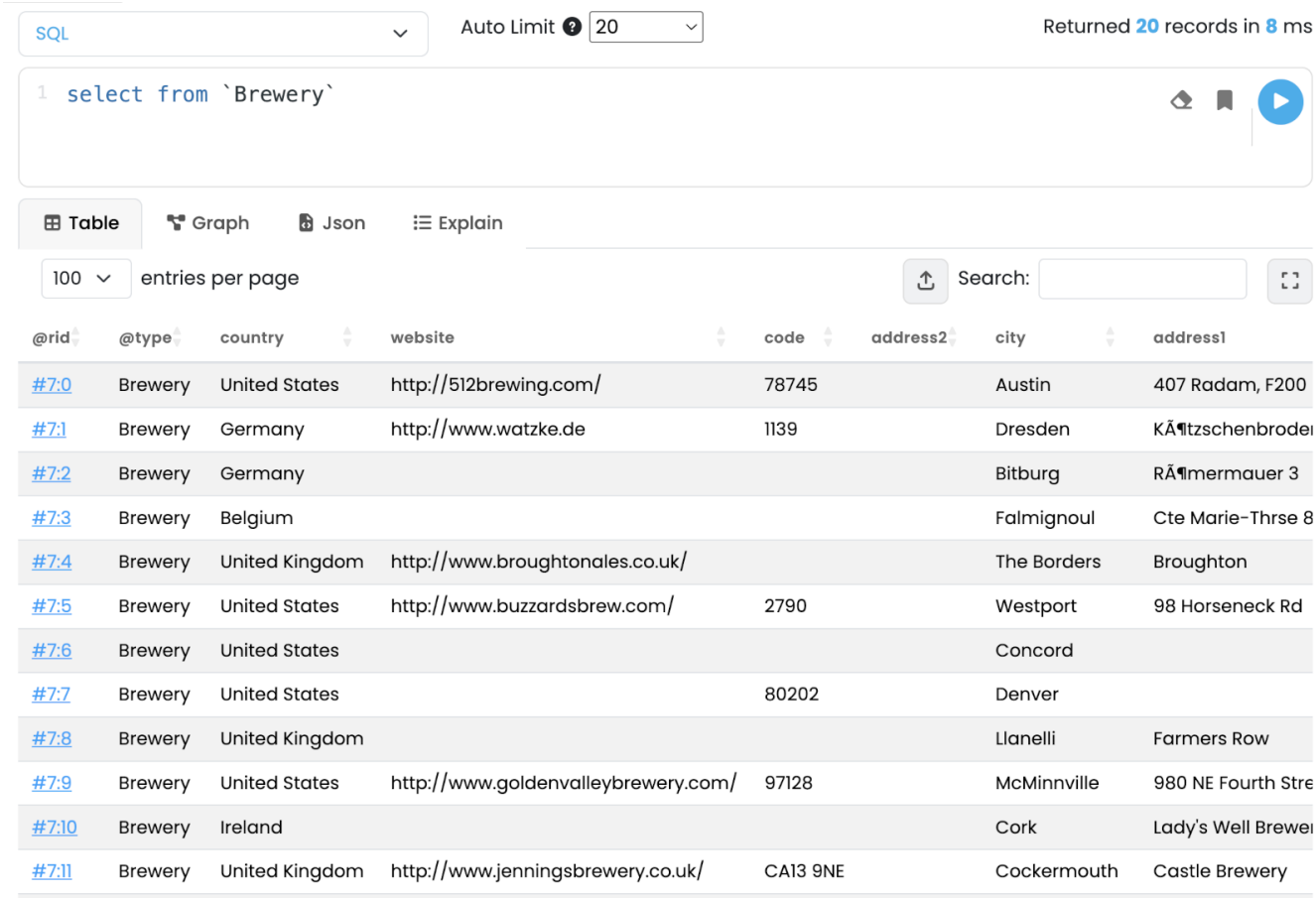
10.2.4. Result views

The result area sits below the editor and exposes four tabs. Studio automatically picks the most appropriate one for the result type, but you can switch at any time.

Table

Tabular view backed by DataTables with sorting, paging and per-column search.

- **Click a cell** to open the **Record Editor** in a modal overlay, where you can edit fields and save the record back to the database.
  - **Save** writes only what you changed: saving without an edit sends nothing, and a property removed with the **x** button is removed from the record, whatever else you added, removed or edited before it.
  - A text property stays text when you edit it (`01234` keeps its leading zero). A number or boolean property takes what you type. An empty field, or the text `null`, sets the property to null.
- The header tools export the current result: **Copy** (to the clipboard), **Excel** (`.xlsx`), **CSV** and **Print**. For a PDF, use **Print** and choose the browser's "Save as PDF" *(since v26.11.1 Studio has no PDF button of its own)*.



@rid	@type	country	website	code	address2	city	address1
#7:0	Brewery	United States	http://512brewing.com/	78745		Austin	407 Radam, F200
#7:1	Brewery	Germany	http://www.watzke.de	1139		Dresden	KÃ¶tzschenbroder
#7:2	Brewery	Germany				Bitburg	RÃ¶mermauer 3
#7:3	Brewery	Belgium				Falmignoul	Cte Marie-Thrse 8
#7:4	Brewery	United Kingdom	http://www.broughtonales.co.uk/			The Borders	Broughton
#7:5	Brewery	United States	http://www.buzzardsbrew.com/	2790		Westport	98 Horseneck Rd
#7:6	Brewery	United States				Concord	
#7:7	Brewery	United States		80202		Denver	
#7:8	Brewery	United Kingdom				Llanelli	Farmers Row
#7:9	Brewery	United States	http://www.goldenvalleybrewery.com/	97128		McMinnville	980 NE Fourth Stre
#7:10	Brewery	Ireland				Cork	Lady's Well Brewe
#7:11	Brewery	United Kingdom	http://www.jenningsbrewery.co.uk/	CA13 9NE		Cockermouth	Castle Brewery

Graph

Force-directed graph view powered by Cytoscape.js. Vertices and edges returned by the command are added to the canvas; existing nodes are kept so you can incrementally explore the graph.

The graph keeps moving with live physics (node repulsion, edge springs and centre gravity), as Neo4j Browser does. *(Since v26.11.1)*

Toolbar actions:

- **Search** — locate a node by `@rid` or a property value.
- **Redraw** — reapply the layout.
- **Select** — selection helpers: **Direct Neighbors**, **Orphans**, **Invert Selection**, **Shortest Path**.
- **Cut** / **Crop** — remove the selected nodes / keep only the selected nodes.
- **Import** / **Export** — JSON, GraphML, PNG, JPEG.
- **Settings** — per-type style overrides (label, color, size, icon).
- **Fullscreen** — expand the canvas to the full browser window.

The GraphML export writes the vertices and edges on the canvas in the same layout as the GraphML database export (see value fidelity of the graph exports): each element's `@rid` as its id, its type under the `labelV` / `labelE` keys, and every property of the record under a key declared with the GraphML type of its values. A map or a list is written as its JSON text, and a value whose type differs from its key's carries an `arcadedb:type` attribute, so the file imports back into ArcadeDB with its types, while yEd, Gephi or NetworkX read it as plain GraphML. What the canvas computes to draw

an element (label, size, color, position) is not exported.

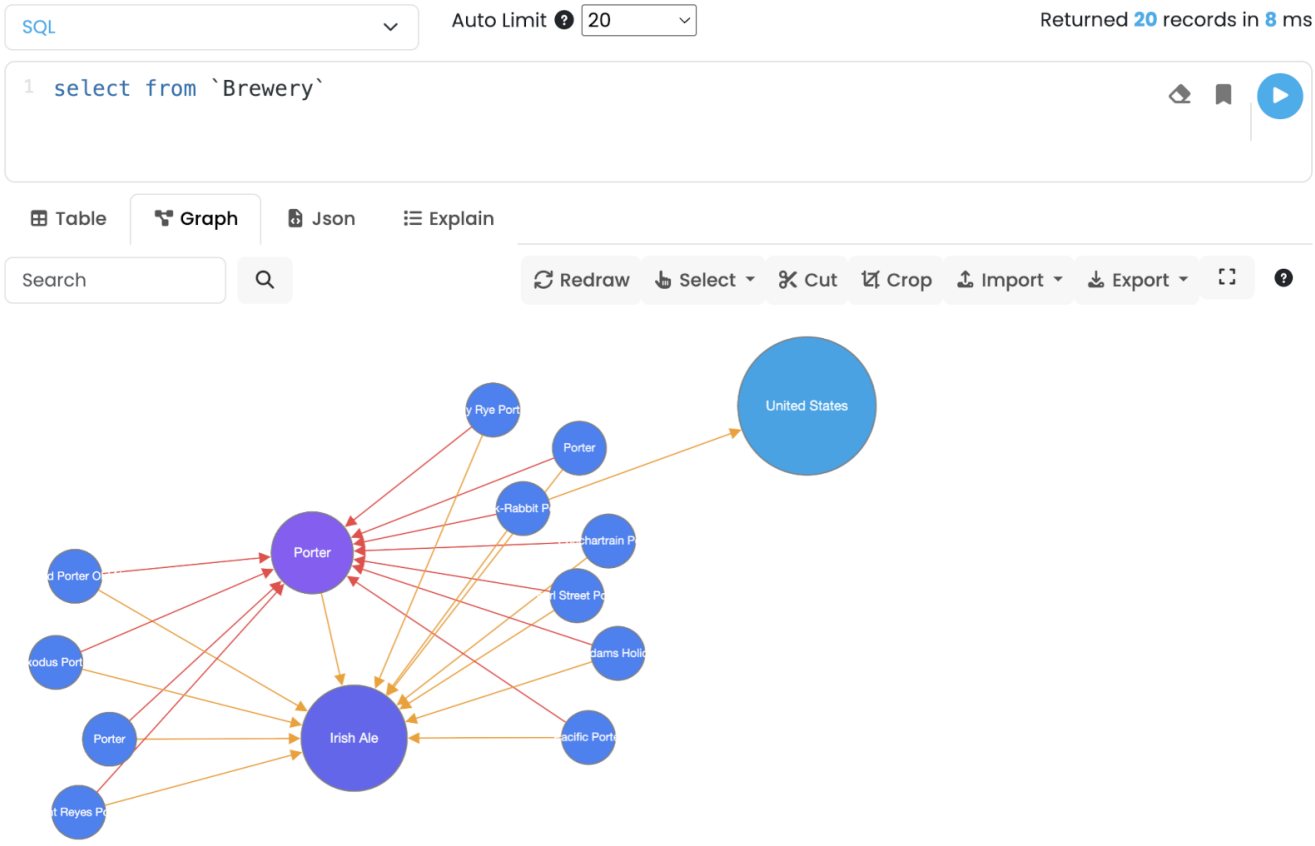
Interactions:

- **Click** on a node selects it; kbd:[Cmd]/kbd:[Ctrl]-drag rubber-bands a block selection.
- **Hold the mouse** on a node opens its context menu (load neighbors, edit, delete).
- **Double-click** on a node loads its direct neighbors.

The node context menu offers four ways to expand:

- the three arrows load **all** neighbours - both directions, outgoing only, incoming only;
- the **funnel** opens the relationship picker, where you choose exactly what to add.

The picker lists one row per relationship type and direction with its edge count, largest first, so you can add only the `Bought` edges going out and the `Knows` edges coming in, rather than everything at once. A **Max elements per direction** box caps how much a single expansion may add. This is what keeps a node with thousands of connections from filling the canvas with a hairball; the counts are computed on the server, so opening the picker does not download those edges just to count them.



Chart

The **Chart** tab plots the rows of the last result, whatever language produced them: SQL, Open Cypher, Gremlin, PromQL and so on. *(Since v26.11.1)* Studio draws all its charts (Query > Chart, the AI Assistant charts, Server operations, the Database CRUD donut, Profiler steps and Time Series) with Apache ECharts, because ApexCharts stopped being open source in 5.1.0. Charts resize with their panel and follow the light or dark theme. The role of each column is inferred from its values, and every choice can be changed in the tab:

- **X axis** - the first time column (epoch milliseconds or an ISO date string), otherwise the first text column, otherwise the row number. A time axis is sorted; a text axis keeps the row order, so a `GROUP BY` result becomes a bar-friendly category chart.
- **Measures** - the numeric columns to plot, one line, area or bar series each.
- **Split by** - a text column to turn into one series per value. On a time chart it defaults to the first text column with at most 12 distinct values (a host, a metric label), so a result with `ts, host, usage` draws one line per host.
- **Line**, **Area**, **Bar** - the chart type.

Record metadata (`@rid`, `@type`) and nested values are not plotted, and a value that is not a number leaves a gap rather than a zero. Chart does not run anything by itself: run a query as usual, then open the tab.

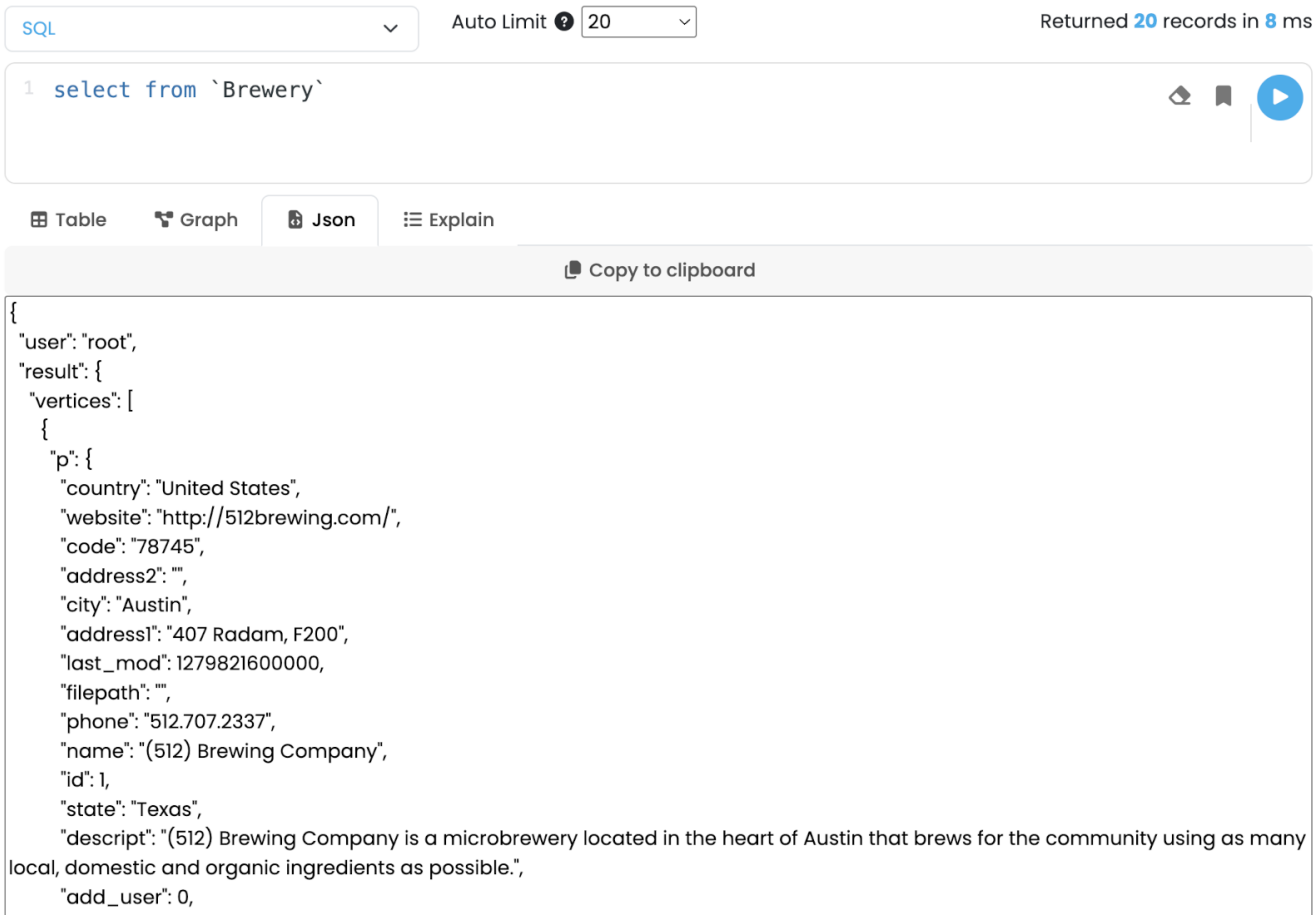
```
SELECT ts, host, usage FROM Cpu
```

PromQL

Selecting **PromQL (Time Series)** in the language selector lets you type a PromQL expression, for example `Cpu{host="a"}`. **Auto Limit** is replaced by a **Range** (last 5 minutes to 7 days) and a **Step** (1 second to 1 hour). The expression runs as a range query against the database's Prometheus-compatible endpoint, and the points arrive as rows `timestamp`, `metric`, `value`, so they can be read in the Table and Json tabs; the Chart tab opens automatically, with one series per `metric`.

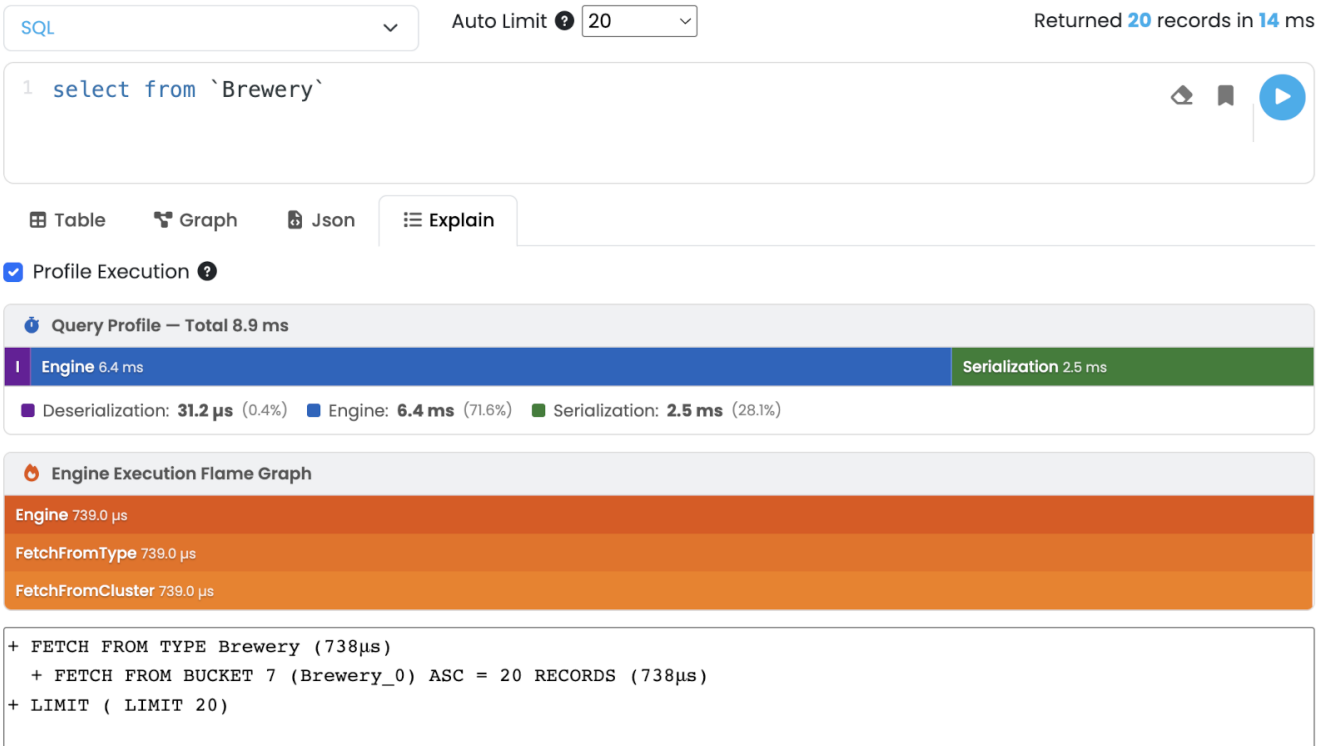
JSON

The raw response is rendered as pretty-printed JSON in a read-only viewer. A **Copy to Clipboard** button copies the full payload.



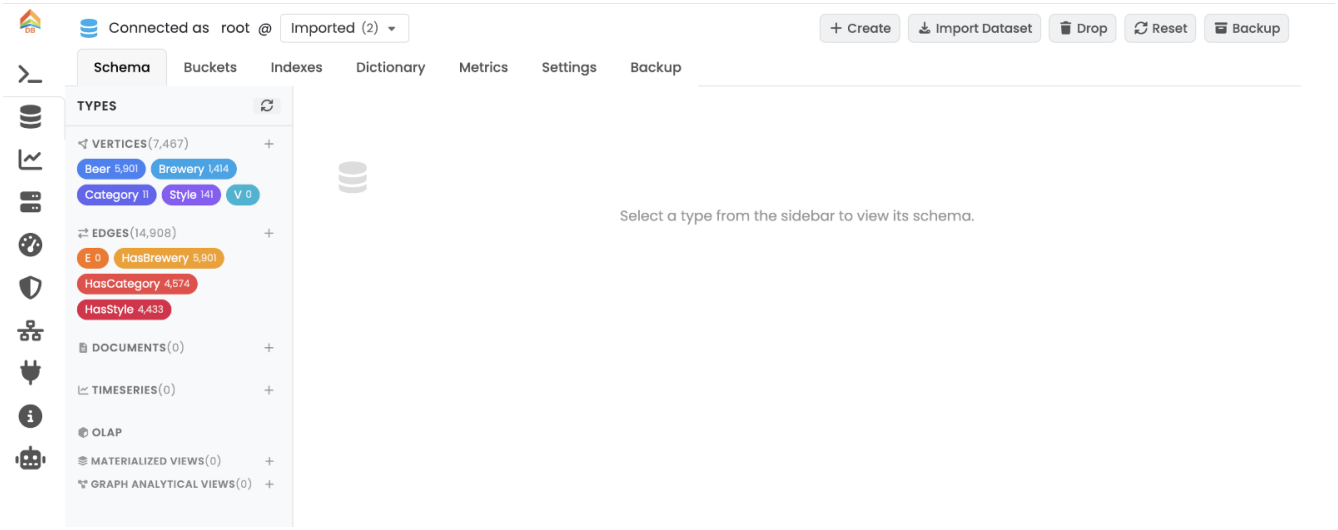
Explain

Shows the execution plan for the current command. Tick **Profile Execution** before running to collect per-step timings; the result is rendered as a flame graph on top of the textual plan, so you can spot the heaviest steps at a glance. Profiling times the query as it normally runs, without changing how it is executed, so the numbers describe the same work the query does when profiling is off.



10.3. Database Panel

The **Database** panel manages everything that lives inside one database: schema, storage layout, indexes, dictionary, time series, metrics, settings and per-database backups.



The header shows the current user, the database selector, and the database-level actions:

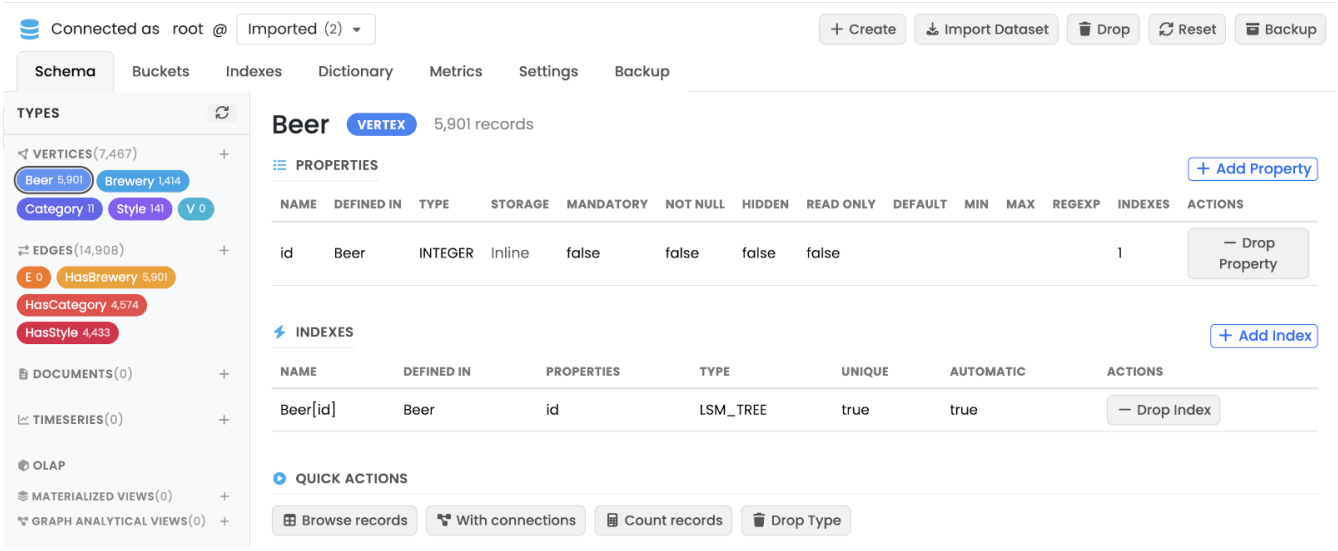
- **Create Database** — create a new database from a name.
- **Import Dataset** — open a modal to import a dataset (e.g. importer formats, JSON, OrientDB or Neo4j export).
- **Drop Database** — ***irreversible*** removal of the current database.
- **Reset Database** — drop all data and schema while keeping the database file.
- **Backup Database** — trigger an immediate backup of the current database. The resulting file is written under `backups/<db-name>/<db-name>-backup-<timestamp>.zip`. See Backup.

The panel is split into seven sub-tabs.

10.3.1. Schema

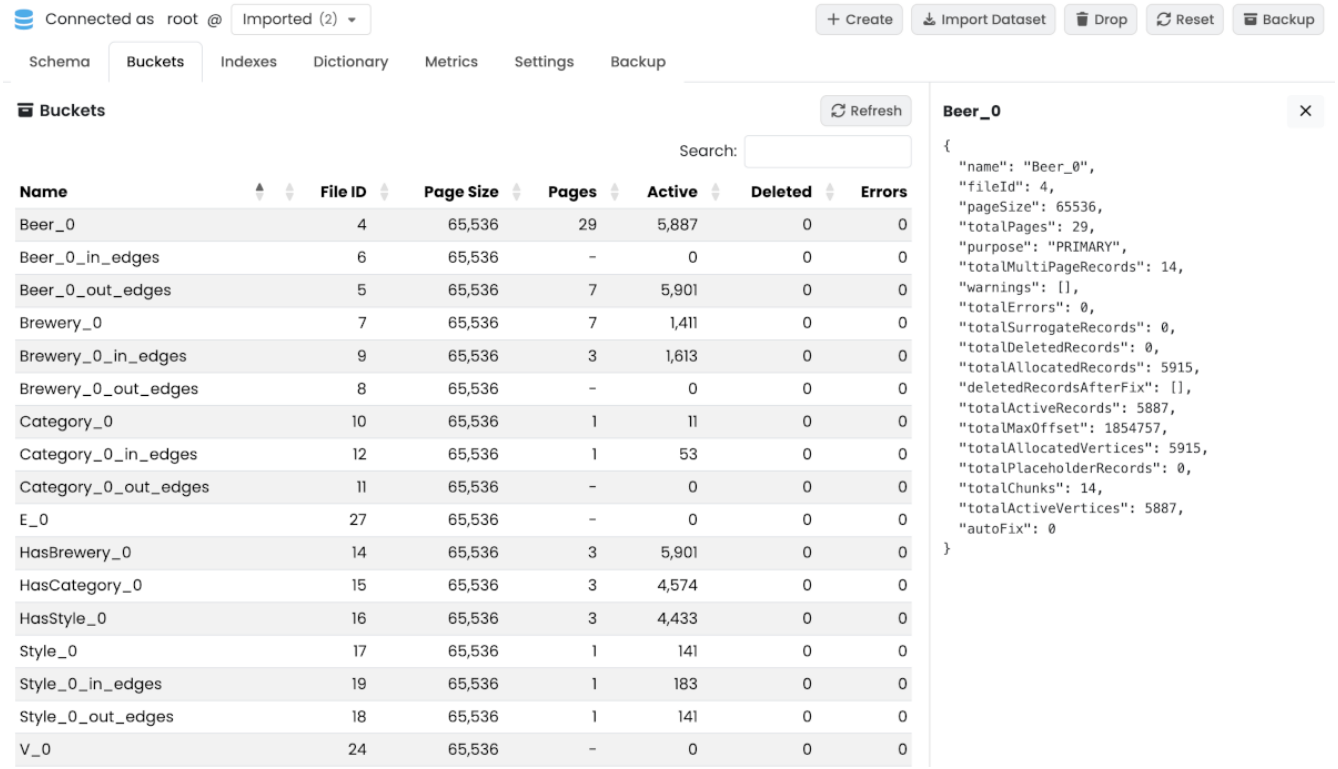
Lists every type in the database (document types, vertex types and edge types) in a scrollable left sidebar. Selecting a type loads its detail on the right: declared properties (name, type, constraints), indexes built on the type, parent / child types, and quick-action buttons.

The **Filter types…** box above the list narrows it to the types whose name contains the typed text, ignoring case *(since v26.12.1)*. Only the name is matched, not the record count shown on each badge. Sections left without a match are hidden, a line under the box says how many types match, and clearing the box shows everything again. The filter stays in place across a refresh, a type created or dropped, or a change of database.



10.3.2. Buckets

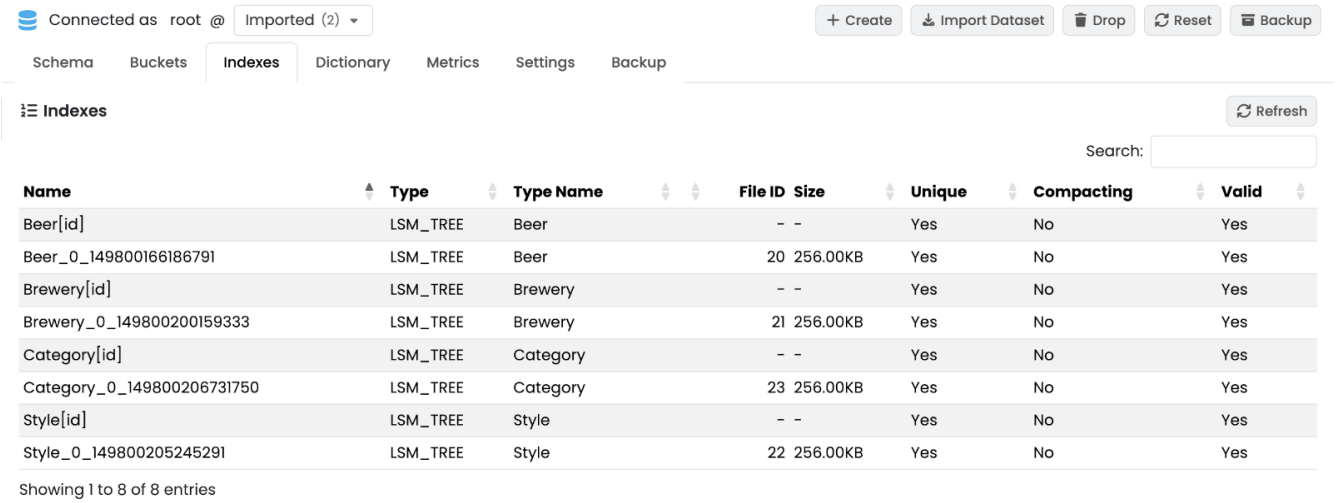
Lists the storage buckets that back the database. Selecting a row opens a detail panel on the right with file path, record count, page count and per-bucket counters.



10.3.3. Indexes

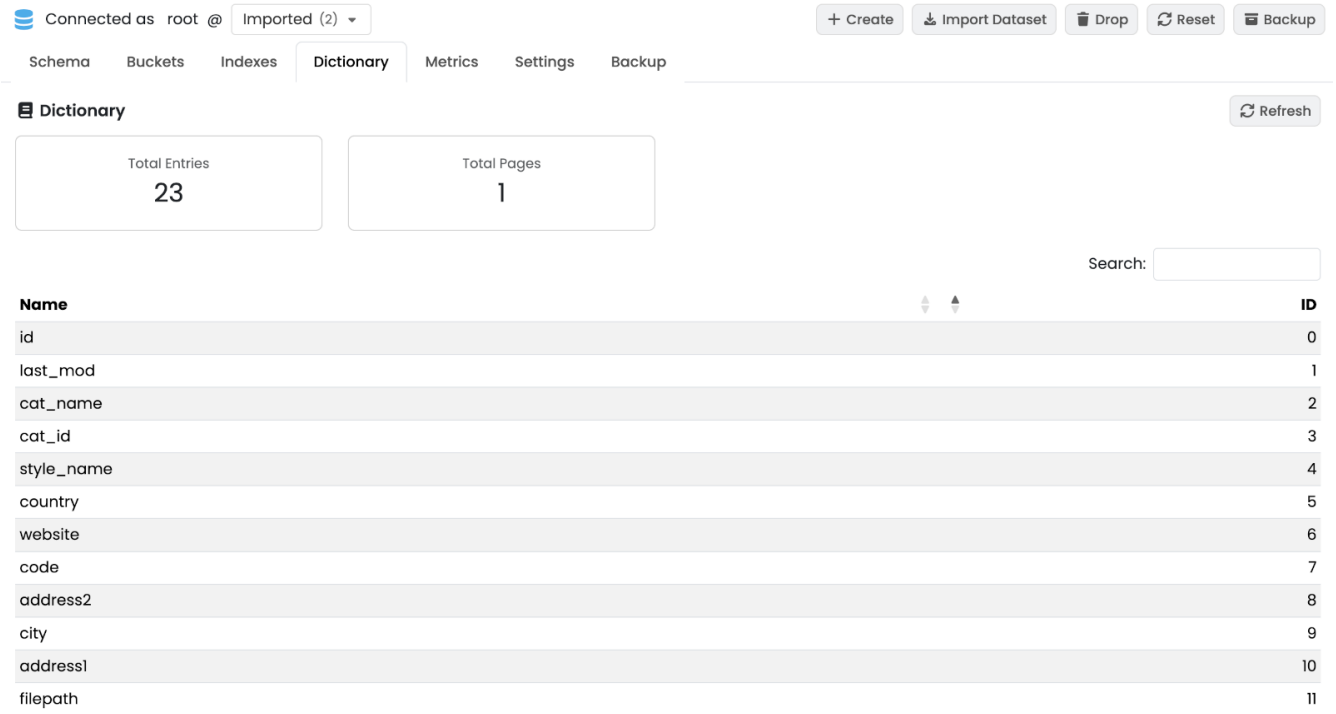
Lists every index defined on the database with type, properties, status and entry count. Selecting a row opens a detail panel with the index definition and statistics.

The **Add Index** dialog has a **Build Mode** option, default or `SORTED`, for `LSM_TREE` indexes *(since v26.11.1)*. `SORTED` adds `METADATA {"buildMode":"SORTED"}` to the generated `CREATE INDEX`, which builds a large index much faster. The option is hidden for the other index algorithms, which do not support it, and the sorted build is refused on replicated databases.



10.3.4. Dictionary

The dictionary deduplicates property names and string values across the database. The sub-tab shows summary cards (total entries, total pages) and a paginated table of the dictionary entries.



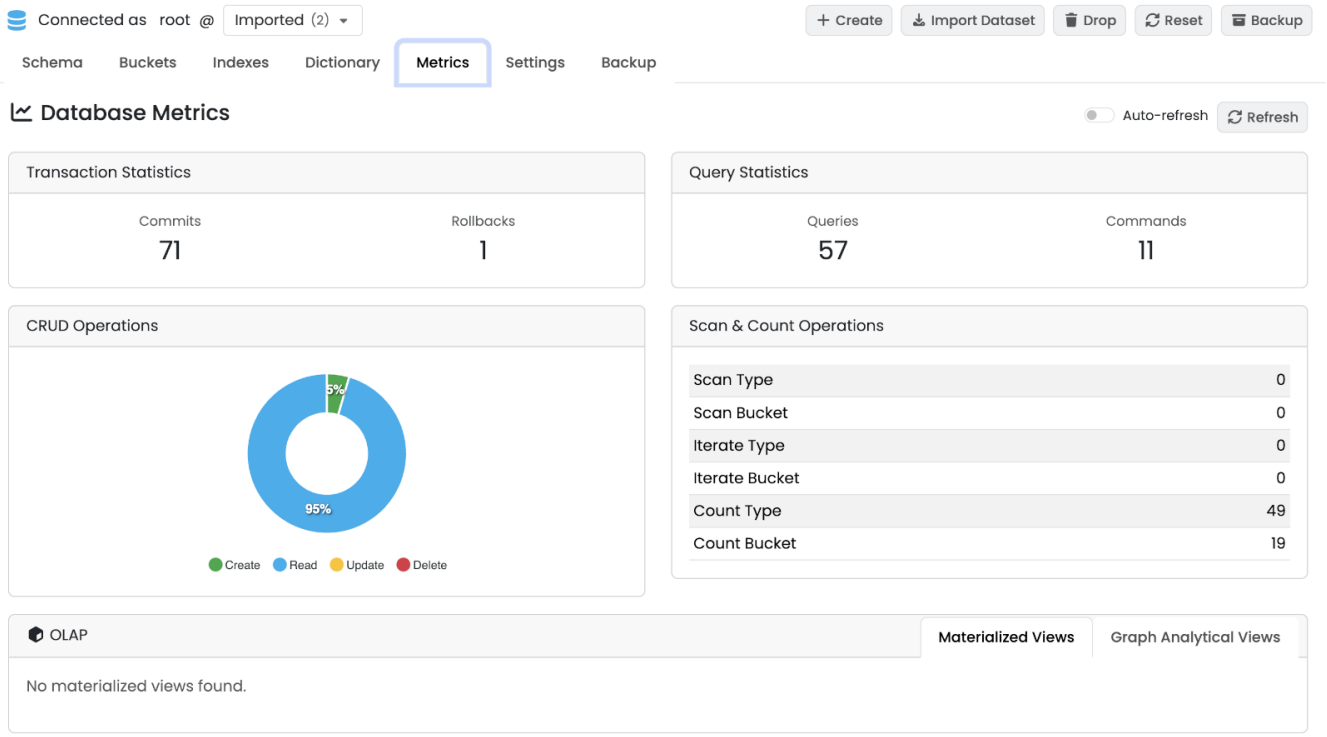
10.3.5. Time Series

The **Time Series** sub-tab holds everything about the database's time-series types: create and drop a type, query it with the builder or PromQL, and find ingestion snippets. It is described in Time Series sub-tab. Time series types also appear in the Schema sidebar, and any time-series result can be charted from the Query panel (see Chart).

10.3.6. Metrics

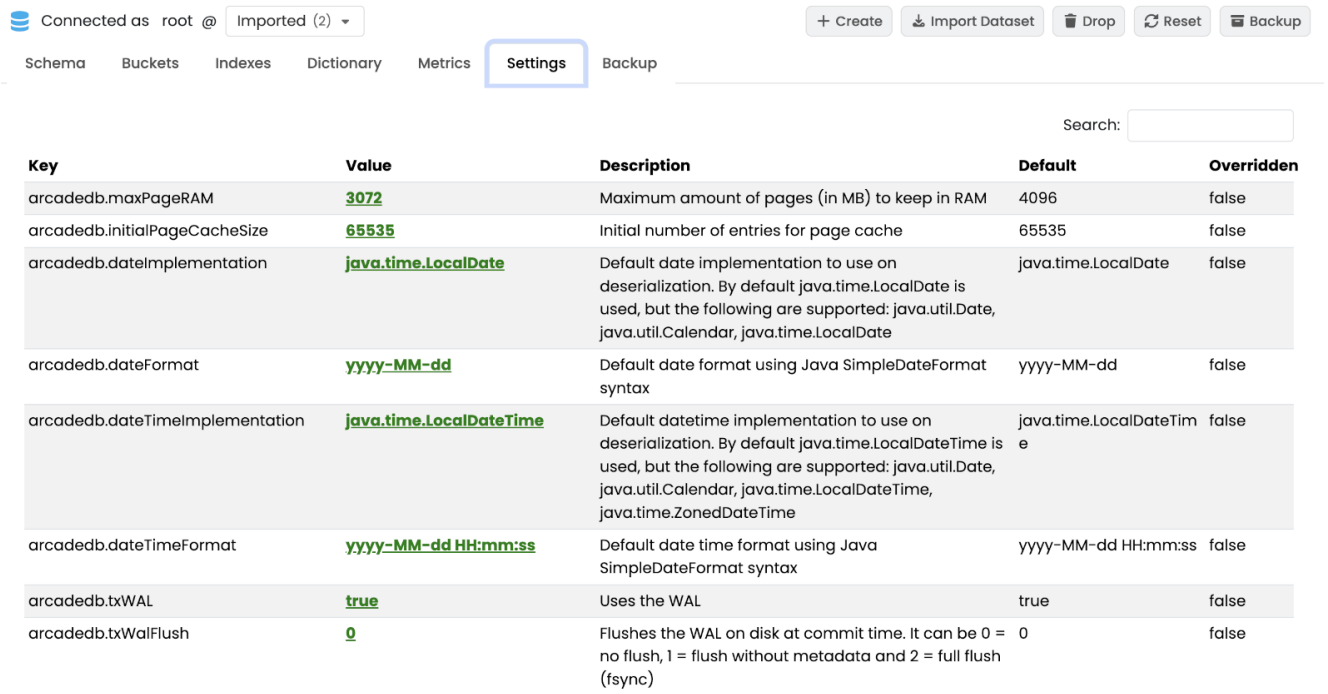
Live counters and charts for the selected database. **Auto Refresh** keeps the values updated at a fixed interval.

- **Transaction Statistics** — commits and rollbacks since startup.
- **Query Statistics** — queries and commands executed.
- **CRUD Chart** — running chart of creates, reads, updates and deletes.
- **Scan & Count** — full-scan and count operations per type.
- **OLAP** — nested tabs for Materialized Views and Graph Analytical Views: refresh status, last computation, row counts.



10.3.7. Database Settings

Key-value table of the database-scoped configuration settings. Values are editable by users with database-settings permission; others see a read-only view. See Settings for the full setting reference.



10.3.8. Backup

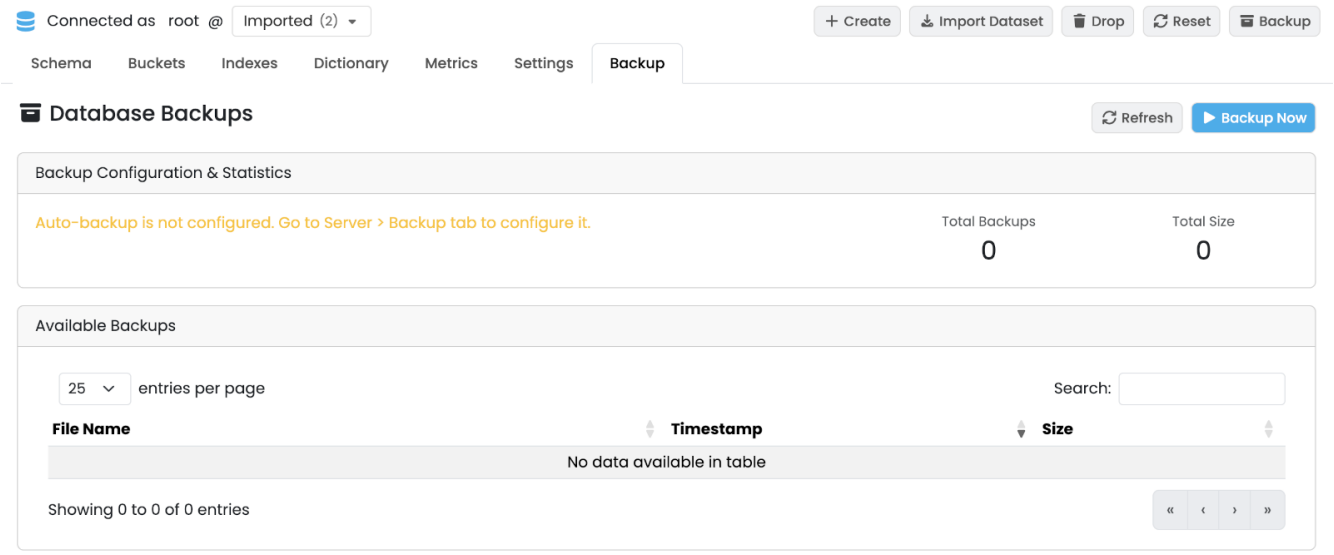
Per-database backup management.

The sub-tab displays:

- **Backup Configuration** — current schedule and retention settings for this database (read-only here; configured server-wide in the Server › Backup panel).
- **Statistics** — total number of backups and disk space used.
- **Available Backups** — paginated list of backup files with timestamps and sizes.

From this sub-tab you can:

- Click **Refresh** to reload the list.
- Click **Backup Now** to trigger an immediate backup of the current database.
- Restore or delete an individual backup from the row actions.



10.4. Time Series Sub-tab

The **Time Series** sub-tab of the Database panel is the UI for ArcadeDB's native time-series types: create and drop types, query them with a builder or PromQL, see the result as a chart and a raw table, and find ready-made ingestion snippets for InfluxDB Line Protocol, SQL, Java and Grafana.

See Time Series for the underlying concepts.

Under the sub-tab bar, a toolbar shows the **Type Selector**, which lists every TimeSeries type in the selected database along with its record count. Two buttons sit next to it:

- **Create TimeSeries Type** — open a modal to define a new type (timestamp column, tags, fields, sharding, retention, downsampling).
- **Drop Type** — ***irreversibly*** delete the selected type.

The sub-tab has three inner tabs: Query, Schema and Quick Tour.

To chart a time series from the Query panel instead, pick the **PromQL (Time Series)** language there, or run a SQL query against the type and open the **Chart** tab (see Chart).

10.4.1. Query

The default sub-tab. It has two modes, toggled at the top:

Builder mode offers structured controls — no need to write a query by hand:

- **Time Range** — `Last 5 min`, `1h`, `24h`, `7d`, `All`, or `Custom` (which exposes **From** and **To** datetime

inputs).

- **Aggregation** — `None`, `AVG`, `SUM`, `MIN`, `MAX`, `COUNT`.
- **Bucket Interval** — `1s`, `10s`, `1m`, `5m`, `1h`, `1d`.
- **Tag Filters** — one input per indexed tag of the selected type.
- **Fields** — checkboxes to pick which numeric measures to plot.
- **Query** — run with the current filters.
- **Latest** — fetch the most recent point per series.
- **Auto-Refresh** — `Off`, `10s`, `30s`, `1m`.

PromQL mode exposes a textarea where you can type a PromQL expression and execute it with the **Run** button or kbd:[Ctrl+Enter] / kbd:[Cmd+Enter].

Below the controls the panel shows the **Query Stats** label (`record count` + `elapsed ms`), the **Chart** and the **Data Table**. Each of the two visualisations can be toggled on or off independently.

The **Chart** is an Apache ECharts line/area chart; toggle the style from the chart toolbar. The **Data Table** is a paginated grid of the raw (or aggregated) values returned by the query.

10.4.2. Schema

Shows the schema of the selected type: timestamp column, indexed tag dimensions, numeric fields, shard count, retention policy and downsampling policies.

10.4.3. Quick Tour

Inline documentation cards with copy-ready snippets for common tasks:

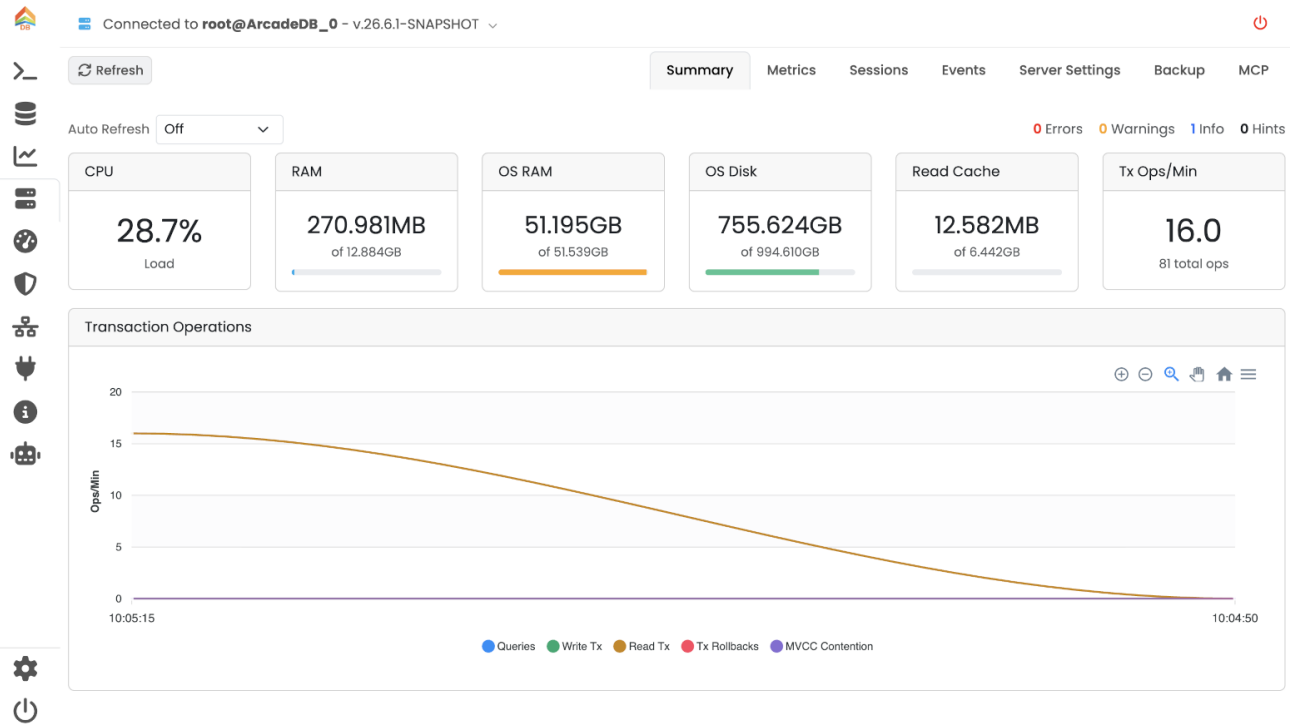
- Creating a TimeSeries type in SQL.
- Ingesting data via InfluxDB Line Protocol (`curl` example).
- Inserting points with SQL `INSERT`.

- Using the Java API to write points.
- Analytical SQL functions (`ts.rate()`, `ts.percentile()`, etc.).
- Configuring downsampling and retention.

A second section covers the **Grafana integration**: how to point the Infinity datasource at ArcadeDB, the available HTTP endpoints (`/health`, `/metadata`, `/query`), metadata discovery, the DataFrame format and a `curl` example.

10.5. Server Panel

The **Server** panel is the operational console for the ArcadeDB server itself: health summary, live metrics, active sessions, running queries, events, server-wide settings, automated backups and MCP integration.



The header shows a **server info** label (click it to open a popover with version, JVM details, host, uptime, etc.) and a **Refresh** button. A **Shutdown** button is available under the **Cluster** tab — see Cluster Panel.

The panel has eight sub-tabs.

10.5.1. Summary

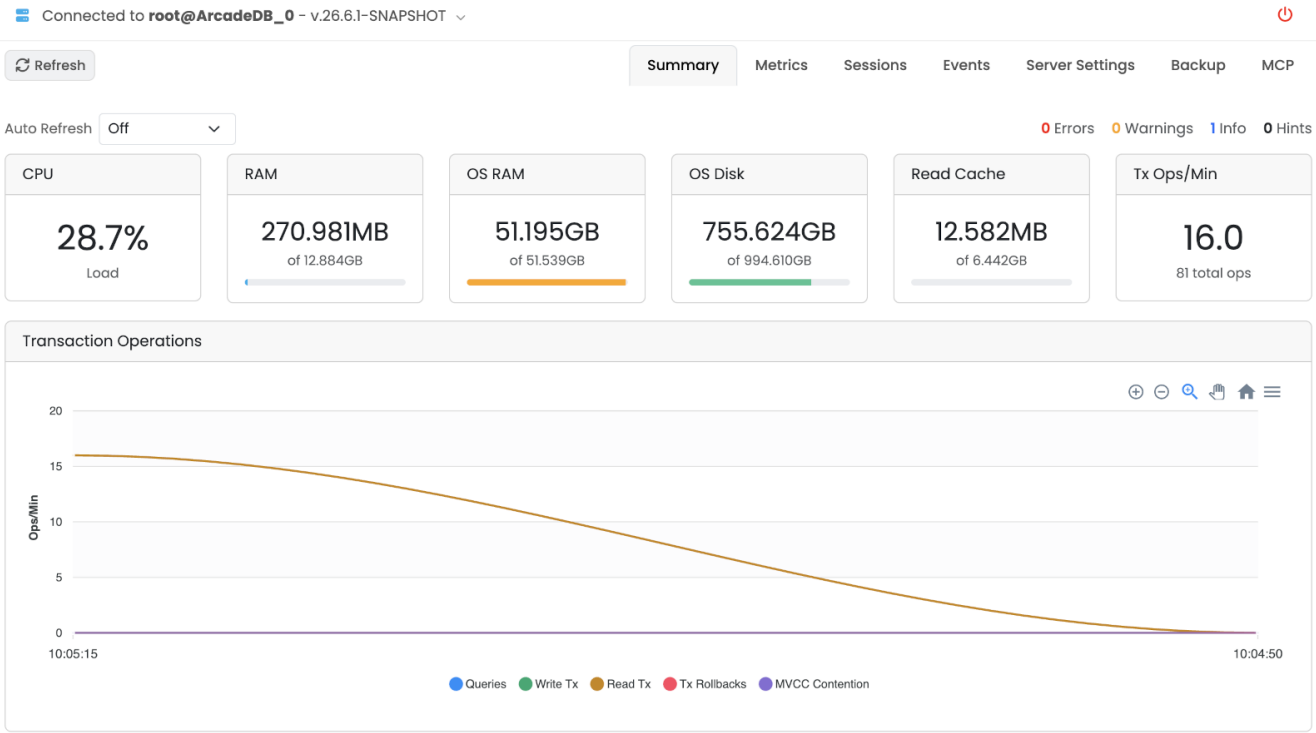
The default sub-tab. **Auto Refresh** keeps the page in sync at a fixed interval (`Off`, `10s`, `30s`, `1m`, `5m`).

The top row shows clickable event counters — Errors, Warnings, Info and Hints — that jump straight to the Events sub-tab filtered on that level.

Below them, system-resource cards report:

- **CPU Load** of the server process.
- **RAM (used / max)** of the JVM, with a progress bar.
- **OS RAM** available on the host.
- **OS Disk** free on the data directory.
- **Read Cache** hit ratio and size.
- **Transaction Ops/Min** throughput.

A **Transaction Operations** chart at the bottom plots ops/min over a rolling window.



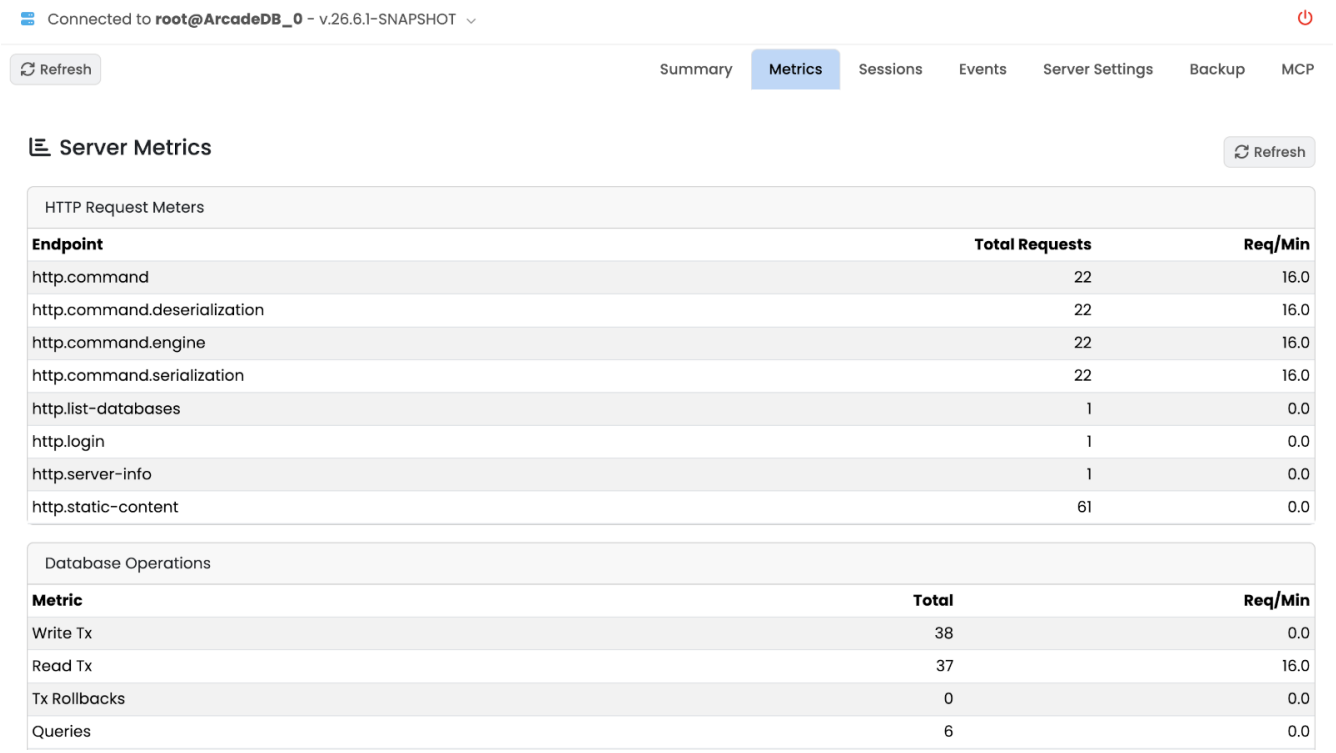
10.5.2. Metrics

Detailed numeric metrics for the server:

- **HTTP Request Meters** — per endpoint, total requests and req/min.
- **Database Operations** — per metric, total and req/min.
- **Profiler Details** — internal profiler counters.
- **Executor Pools** — thread pools (active vs. allocated threads, queue depth, completed tasks, fallbacks, reclaimed).

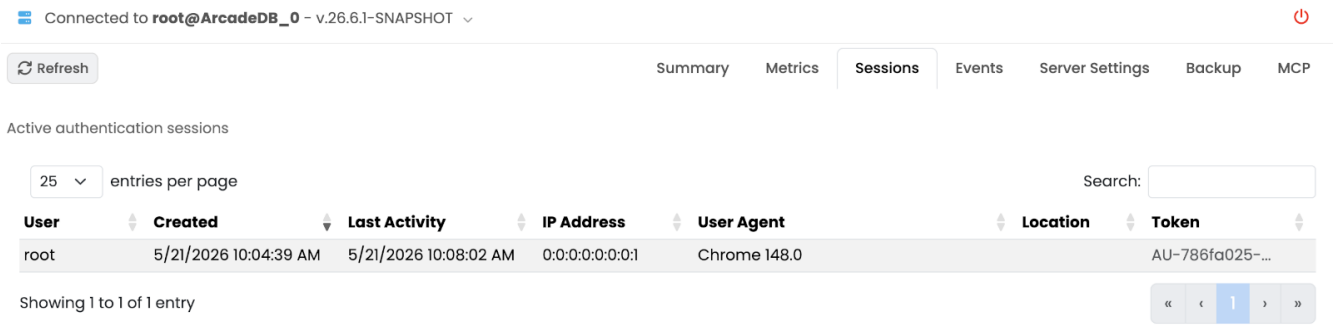
Two of those columns look similar and mean opposite things. **Caller-Run Fallbacks** counts tasks that ran on the submitting thread because the pool's queue was **full**; sustained growth there means the pool is undersized for your workload and is the signal to raise its size. **Reclaimed** counts tasks a waiting caller took back and ran itself because the pool was **busy** rather than full: that is normal, healthy behaviour and needs no action.

- **Sparse Vector Indexes** — counters for vector indexes when present.



10.5.3. Sessions

Table of active authentication sessions: user, source, started, last-seen. Use it to spot stale or unexpected connections.



10.5.4. Running Queries

Table of the statements the server is running, from every protocol (HTTP, PostgreSQL, Bolt, gRPC, MongoDB, Redis, Gremlin): id, database, user, protocol, language, statement, tag, elapsed time and status (`running`, or `terminating` once it was asked to stop). The **root** user sees every statement, any other user only their own. Available since 26.10.2.

Auto Refresh reloads the table every 2, 5 or 10 seconds (`Off` by default).

The **Terminate** button at the end of a row stops that statement after a confirmation, and rolls back what it wrote. A notification then says what happened: the statement was terminated, it completed before it could be terminated (what it did stands), it was asked to stop and has not stopped yet, or it was no longer running.

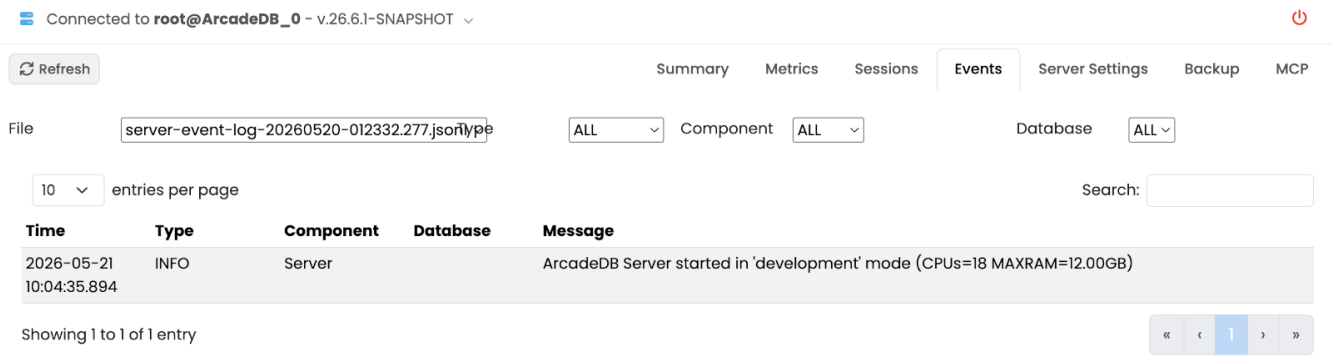
The tab is built on the `list queries` and `terminate query` server commands; see there for how a statement is labelled with a tag and how each protocol reports the termination to its client.

10.5.5. Events

Browsable server event log. Filter dropdowns above the table:

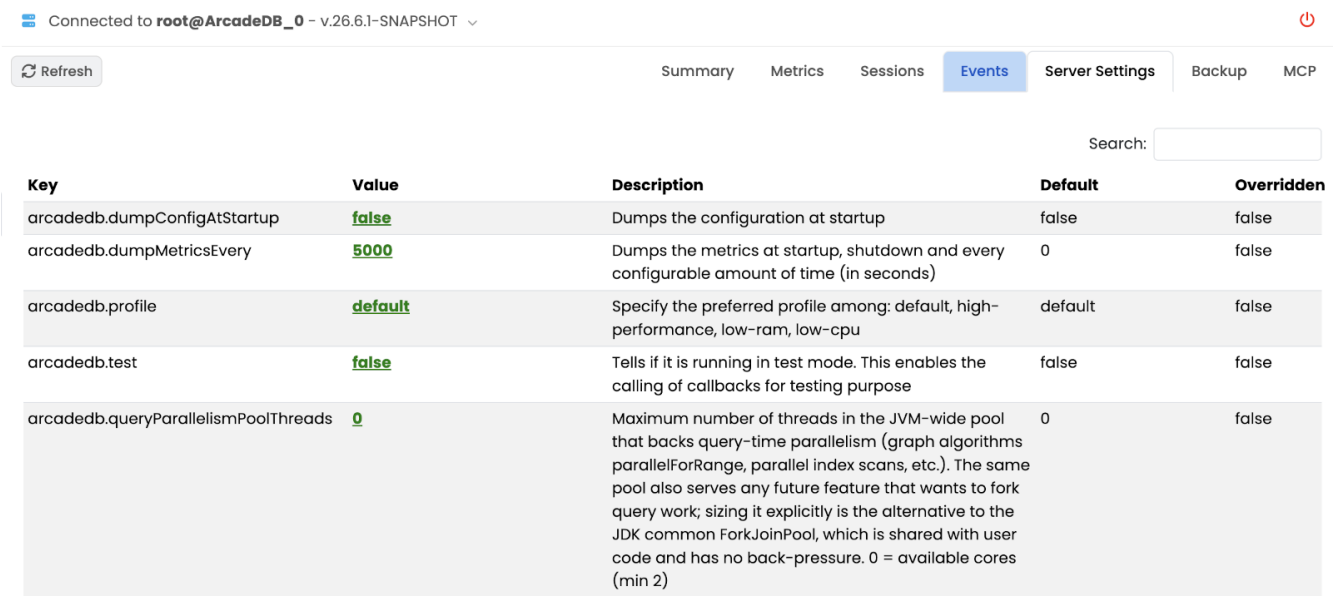
- **File** — pick a log file.
- **Type** — `ALL`, `CRITICAL`, `WARNING`, `INFO`, `HINT`.
- **Component** — server subsystem (HTTP, transactions, indexes, ...).
- **Database** — limit to events scoped to one database.

Each row shows timestamp, level and message.



10.5.6. Server Settings

Key-value table of the server-scoped configuration. See Settings for the full setting reference.

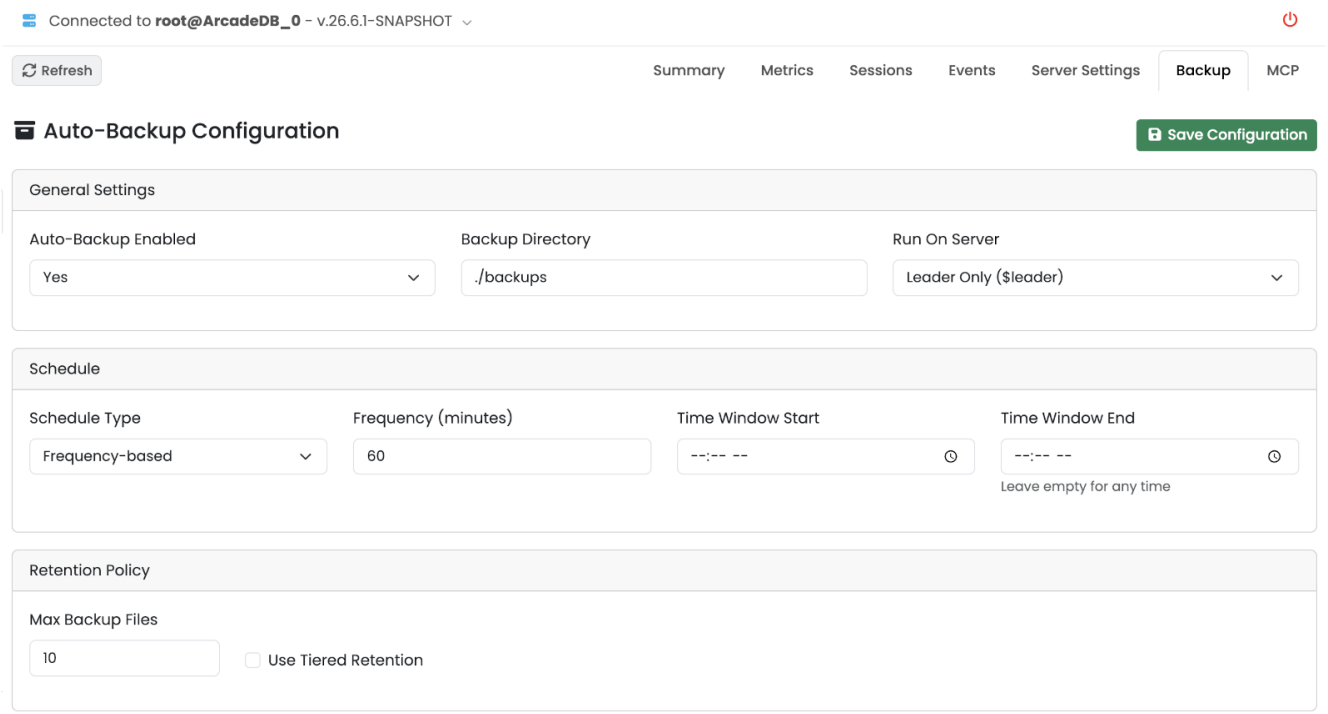


10.5.7. Backup

Server-wide auto-backup configuration.

- **Auto-Backup Configuration** — toggle to enable or disable automatic backups, set the backup directory, and choose whether backups run on the leader only or on every server in a cluster.
- **Backup Schedule** — pick **Frequency** (every N minutes/hours/days) or a full **CRON** expression.
- **Retention Policy** — **Max files** (simple ring) or **Tiered** (per-bucket keep counts: Hourly, Daily, Weekly, Monthly, Yearly).
- **Backup Status** — last run, next run, files produced.

See Automatic Backup for the underlying configuration and CLI equivalents. Per-database backup history is managed from the Database › Backup sub-tab.



10.5.8. MCP

Configures the embedded Model Context Protocol server so AI tools (Claude Desktop, Claude Code, Cursor, …) can talk to ArcadeDB.

The left column controls **Settings**:

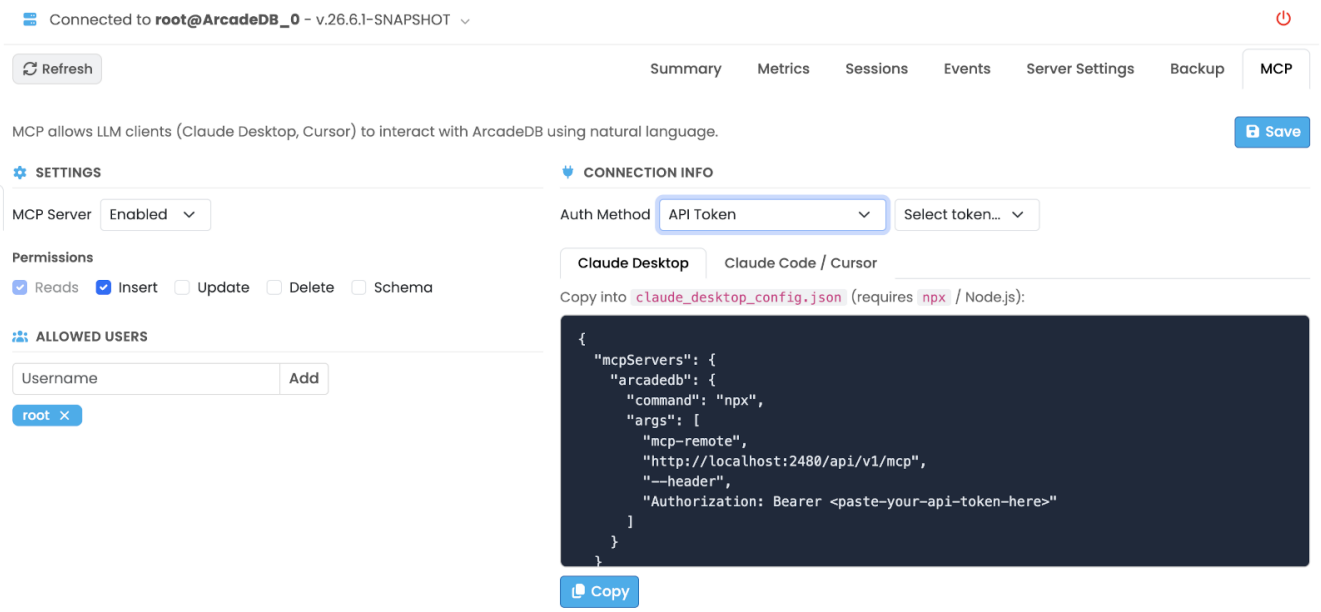
- Enable or disable the MCP endpoint.
- Per-operation permissions: **Reads**, **Inserts**, **Updates**, **Deletes**, **Schema**.
- **Allowed Users** — add or remove users that may authenticate against MCP.

The right column shows **Connection Info**:

- **Auth Method** — Basic auth or API token.
- Ready-to-paste configuration snippets for **Claude Desktop** and for **Claude Code** / **Cursor**, each

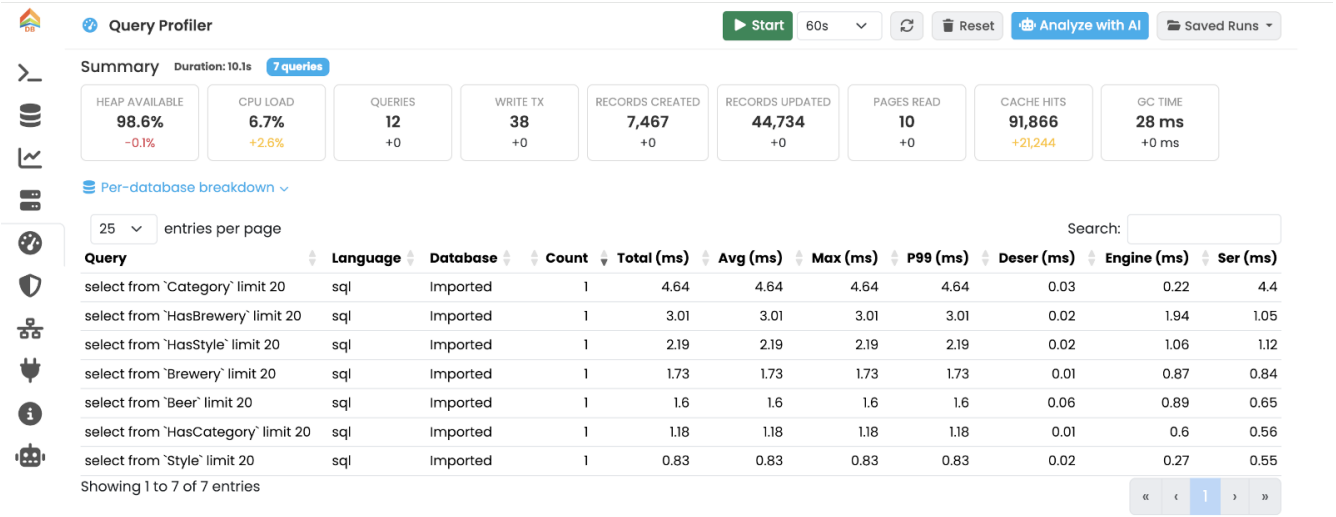
with a **Copy** button.

See MCP Server for the protocol reference.



10.6. Profiler Panel

The **Profiler** panel records the queries executed on the server across every protocol (HTTP, Java embedded, BOLT, PostgreSQL, MongoDB, Redis, MCP), aggregates them by canonical form, and lets you drill into per-step timings to find slow paths. Runs can be saved for later comparison, and an **Analyze with AI** button (requires the AI Assistant subscription) produces a natural-language commentary on the captured run.



10.6.1. Toolbar

- **Start** — begin a new recording. A **Timeout** selector next to it caps the run length (`30s` to `10m`, default `60s`); recording stops automatically when the timeout elapses.
- **Stop** — stop the recording before the timeout (only visible while recording).
- **Refresh** — re-pull the current run from the server.

- **Reset** (trash icon) — clear the current run.
- **Analyze with AI** — open the AI commentary panel (only shown when results are available and the subscription is active).
- **Saved Runs** — dropdown of previously saved sessions; pick one to load it back into the view.

A run saved by an older build does not record how many executions each step was timed in, so Studio flags it at the top of the view and marks such steps `not timed` rather than showing a figure it cannot vouch for. Record the run again on a current build to get complete step timings.

10.6.2. Summary

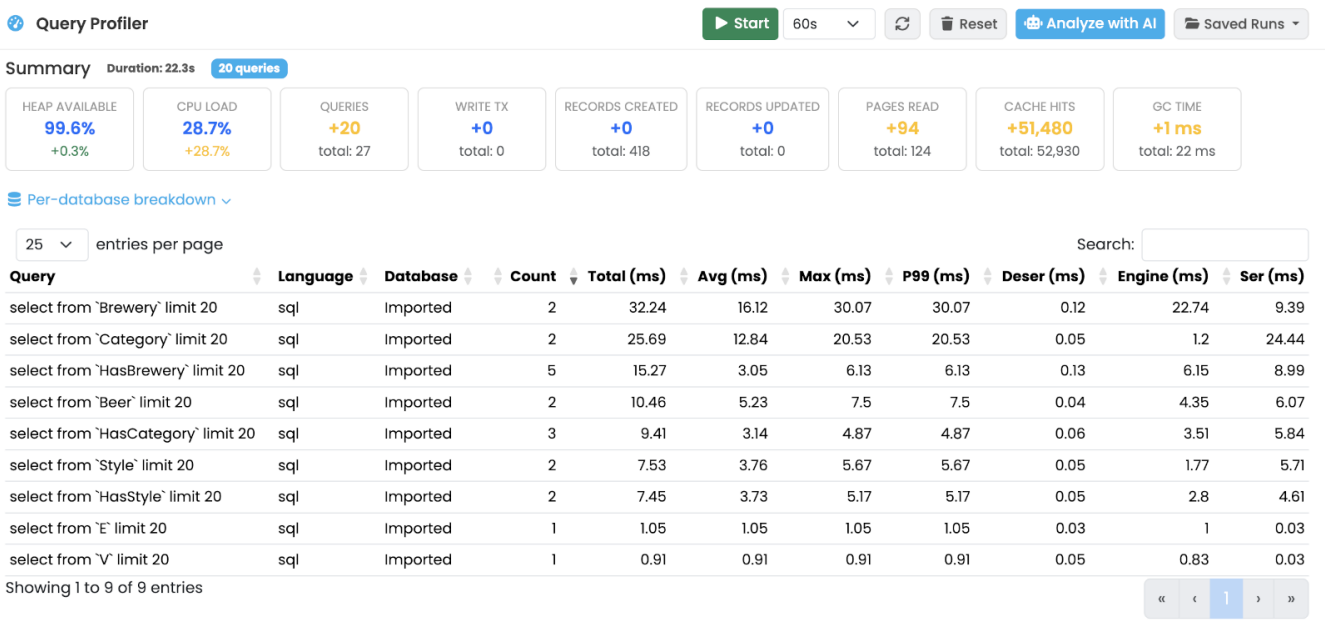
Once a recording exists, a summary header appears at the top:

- **Duration** of the recording.
- **Total Queries** captured.
- **Metric Delta** cards comparing this run to the baseline (where applicable).
- Per-database breakdown, collapsible.

10.6.3. Query list

The default view is a table of distinct queries collected during the run. Columns: **Query** (canonical text), **Language**, **Database**, **Count** (executions), **Total**, **Avg**, **Max**, **P99** (ms), and per-stage timings **Deser**, **Engine**, **Ser** (ms).

Click a row to drill into that query.



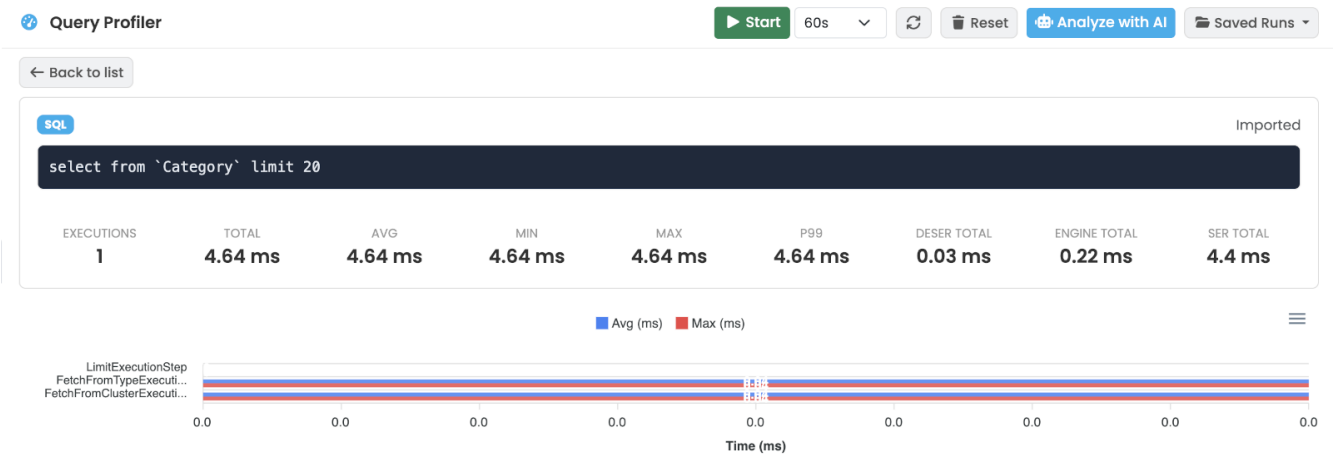
10.6.4. Query detail

The detail view exposes:

- **Back to List** button to return.
- Query card with the **Language** badge, **Database**, the canonical query text and stat cards (**Count**, **Total**, **Avg**, **Max**, **P99**).
- **Step Chart** - Apache ECharts breakdown of execution-step timings. Only steps that were actually timed are plotted.
- **Step table** — for each execution step: name, count, **measured**, total, min, avg, max, p99 (ms). **Measured** is how many of those executions the step was actually timed in, so an average drawn from 3 of 100 executions says so instead of reading as if it covered all 100.

Step timings are the time each step spent in its **own** work - never the time the steps below it spent in theirs - summed over every occurrence of that step across all the executions of the query. Because no step counts another step's time, the rows add up to at most the query's **Engine** total and never more, and the largest row is the step actually worth optimising. A step the engine did not time is shown as `not timed` rather than as `0 ms`, so it is not mistaken for work that completed instantly. A step whose only job is to hand work to the steps below it therefore appears with a count but without timings: the time is reported on the step that did the work.

Gremlin traversals are timed on the same run that returns their results, so recording does not execute a traversal twice, and a write traversal (`addV`, `addE`, ...) is still applied only once. Their steps (`GraphStep`, `HasStep`, ...) appear in the step table and chart like SQL and Cypher steps.



Step	Count	Total (ms)	Min (ms)	Avg (ms)	Max (ms)	P99 (ms)
LimitExecutionStep	1	0	0	0	0	0
FetchFromTypeExecutionStep	1	0.04	0.04	0.04	0.04	0.04
FetchFromClusterExecutionStep	1	0.04	0.04	0.04	0.04	0.04

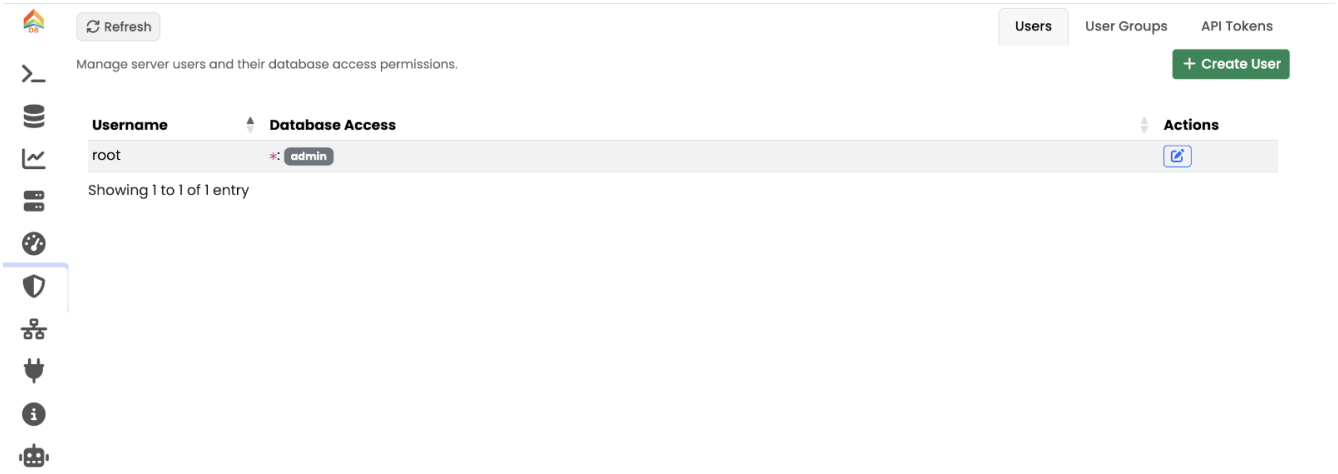
10.6.5. AI analysis

When you click **Analyze with AI**, a panel slides in with AI-generated insights — likely bottlenecks, suggested rewrites, indexes that may help. Closing the panel keeps the underlying run intact.

The analysis is rendered in the same Markdown subset as the AI Assistant's answers (see AI Assistant): raw HTML and links that are not `http` or `https` are shown as text. Code blocks in the analysis have no copy button. *(Since v26.10.2.)*

10.7. Security Panel

The **Security** panel manages who can log into the server, with which permissions, and which automated clients have API tokens. It is the UI front-end of the ArcadeDB security model — see Users, Groups and Security Policy for the underlying concepts.



The header has a **Refresh** button. The panel is split into three sub-tabs.

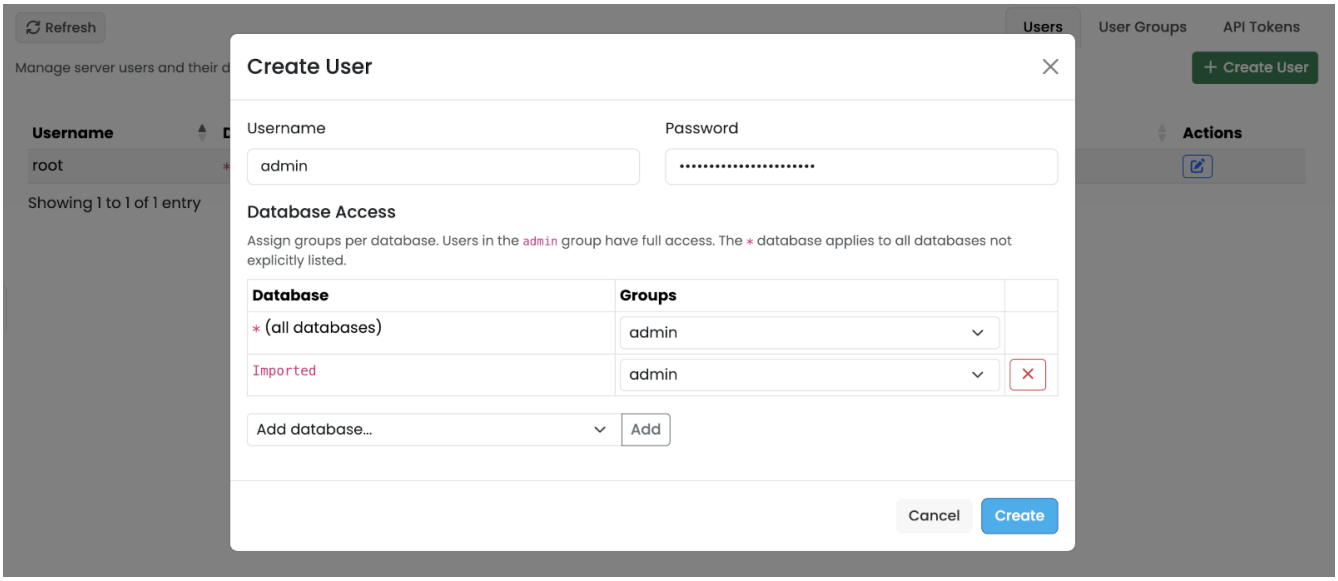
10.7.1. Users

Lists every user known to the server: username, e-mail, and the groups they belong to in each database.

- **Create User** — open the **Create User** modal.
- Per-row actions: **Edit**, **Delete**.

The **Create User** modal asks for:

- **Username** and **Password**.
- **Database Access** — a table mapping each database to a group. The wildcard * row represents the default group used for any database not listed. Use the **Add Database** dropdown to grant access to additional databases with a specific group.



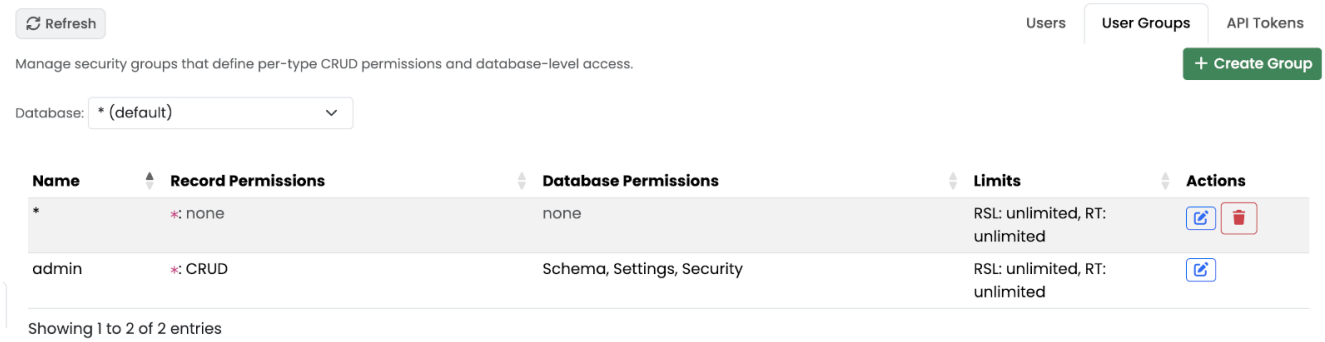
10.7.2. User Groups

Groups (aka roles) bundle the permissions granted to the users that belong to them.

- **Database** filter at the top — limit the table to groups defined for a specific database.
- **Create Group** — open the **Create Group** modal.
- Table columns: **Name**, **Result Set Limit**, **Read Timeout**, type-level CRUD checkboxes, database-level permission checkboxes.

The **Create / Edit Group** modal exposes:

- **Database** and **Name**.
- **Result Set Limit** — cap the number of rows returned per query.
- **Read Timeout** — cap on the duration of read queries (ms).
- **Type Permissions** — per-type **Create**, **Read**, **Update**, **Delete** checkboxes. The * row is the default applied to types not explicitly listed; use the **Add Type** dropdown to override the default for specific types.
- **Database Permissions** — **Schema Changes**, **Database Settings**, **Security**.



10.7.3. API Tokens

Long-lived bearer tokens you can issue to scripts and services in place of username/password

authentication.

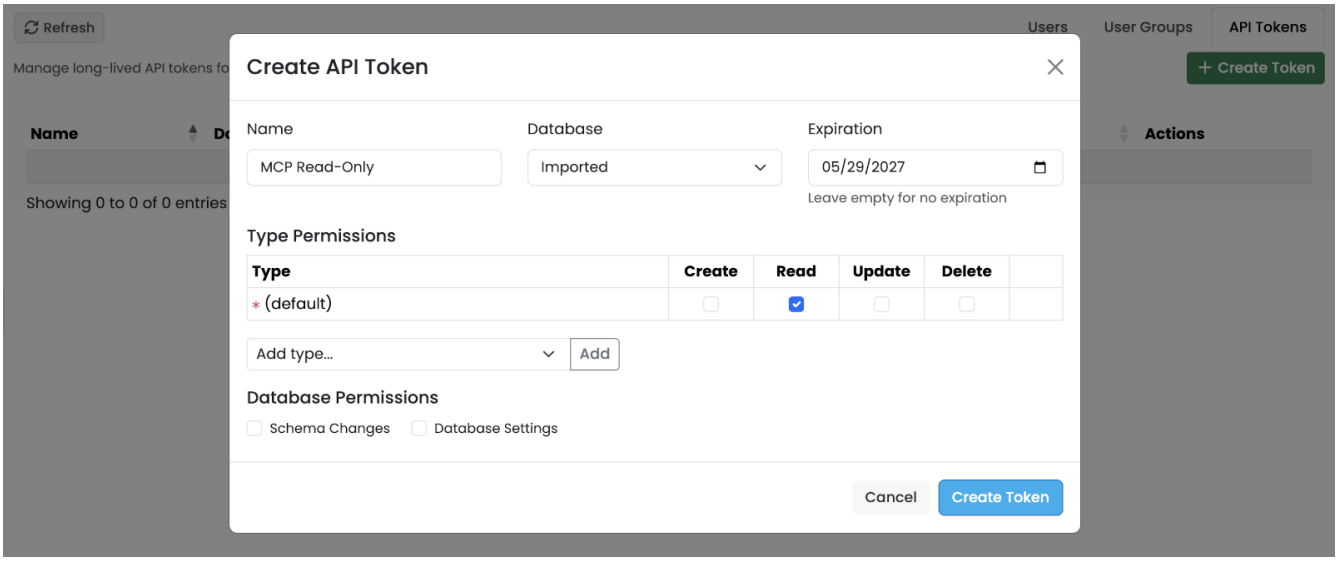
- **Create Token** — open the **Create Token** modal.
- Table columns: **Name**, **Database**, **Expiration**, **Status** (Active / Expired), **Actions** (View / Revoke).

The **Create Token** modal asks for:

- **Name** and **Database**.
- **Expiration** date.
- **Type Permissions** and **Database Permissions** — same structure as the Groups modal.

When you submit the form the new token is shown once in the **Token Created** modal, with a **Copy** button. The token is **not** shown again, so copy and store it before closing the dialog.

An expired token authenticates nobody from the moment it expires, but it stays in the list, marked as expired, until the next token is created or revoked - that is when expired entries are cleared out. You can reuse the name of an expired token for its replacement.



10.7.4. Group and Token Changes During a Rolling Upgrade

On a cluster, a group or API-token change is replicated to every node, and the leader refuses it while any node cannot yet understand it — including a node that is simply unreachable. Nothing is submitted when it is refused, so nothing changes anywhere and the same request succeeds unaltered once the cluster is whole again.

Studio shows this before you act instead of after: **Create Group** and **Create Token** are disabled while the cluster is not ready, with a notice above the table naming the nodes that are holding it back and why. The Create / Edit Group dialog carries the same notice, because editing an existing group is replicated the same way. Revoking a token is refused by the same rule — the control stays available, and the refusal explains itself.

See Cluster Readiness on the Cluster panel for the same information per node. Nothing of this appears on a single server or on a fully upgraded cluster.

10.8. Cluster Panel

The **Cluster** panel shows the live topology and health of an ArcadeDB High Availability cluster, the current leader, and exposes membership-management actions on the leader.

The header shows:

- **Cluster Name**.
- A **Role** badge for the local node (**Leader** or **Follower**).
- A **Health** badge (color-coded).
- **Refresh** button.
- **Auto-Refresh** dropdown (`Off`, `5s`, `10s`, `30s`, `1m`).

The panel has three sub-tabs.

10.8.1. Overview

Top info row: **Leader** name, **Leader HTTP** address, **Local Peer** identifier, **Server count**.

Below it, a grid of **Node Cards** — one per node — each showing status, uptime, hosted databases and key metrics. Each card also lists what that node's build supports, or, when the leader has not been able to ask it, the reason why nothing is known about it.

A **Cluster Readiness** banner appears above the cards when a group or API-token change would be refused right now. Those changes are replicated to every node, and a node that cannot understand the change stops rather than skip it, so the leader refuses the whole change until the last node is upgraded and reachable. The banner names which nodes are holding the cluster back and why, so a rolling upgrade that is not finished is visible before you attempt an administrative change rather than after. It is hidden entirely on a single server, on a fully upgraded cluster, and on a follower - only the leader polls the other nodes, so only the leader can answer for them.

10.8.2. Metrics

Summary cards:

- **Elections** — number of leader elections since startup.
- **Last Election** — timestamp of the most recent election.
- **Uptime** — cluster uptime.

10.8.3. Management

Membership and lifecycle actions.

The left column exposes:

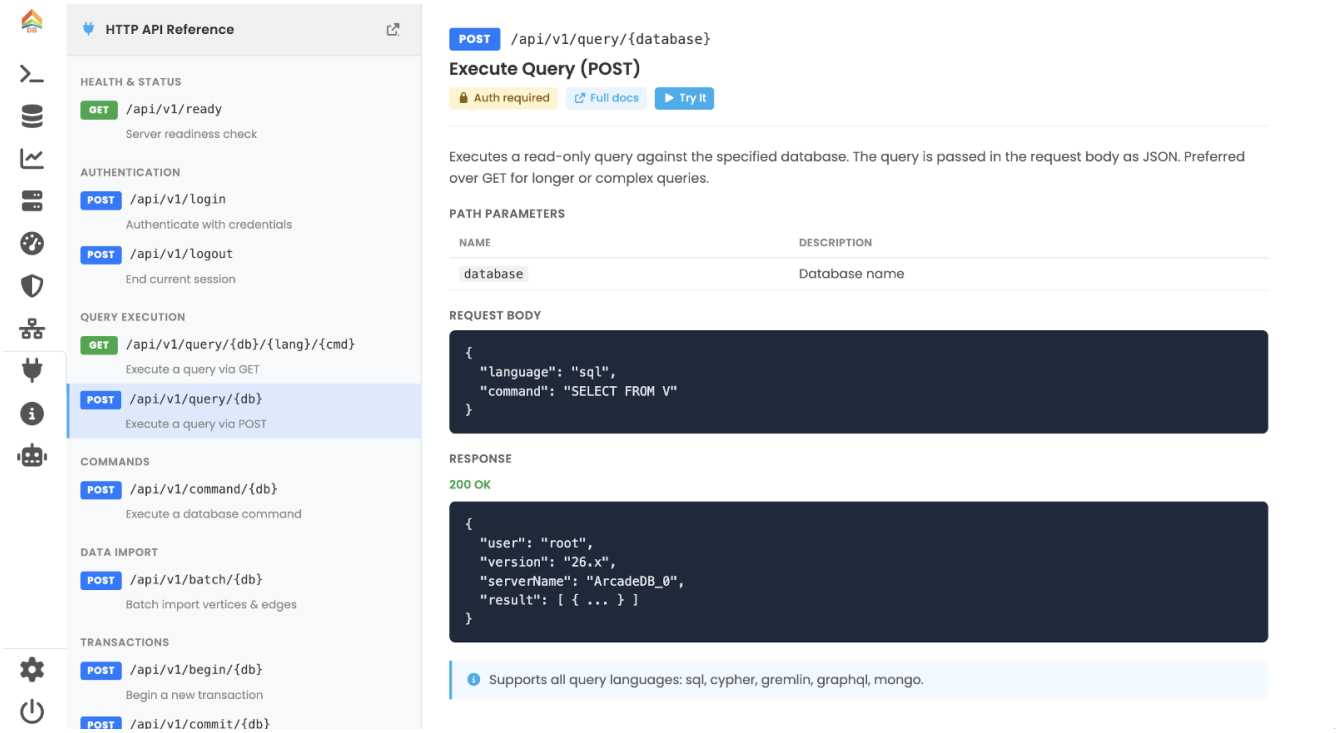
- **Cluster Membership** — list the current peers and add or remove them. This action is only available on the leader; on a follower the controls are read-only.
- **Bootstrap Baselines** — when applicable, shows the per-database fingerprints used to validate new peers.

The right column contains a **Danger Zone** with the **Shutdown This Server** button. Shutting a server down is irreversible from the UI; in HA the rest of the cluster keeps running.

10.9. API Panel

The **API** panel is an interactive browser for the HTTP API exposed by the current ArcadeDB server. Pick an endpoint from the left sidebar to see its contract on the right; flip the **Try It** toggle to send a real request from the built-in playground.

See HTTP API for the full reference.



10.9.1. Endpoint sidebar

The left sidebar groups the API surface by topic:

- **Health & Status**
- **Authentication**
- **Query Execution**
- **Commands**
- **Data Import**
- **Transactions**
- **Server Management**
- **Database Management**
- **TimeSeries**
- **Documentation**

Each entry shows the HTTP verb badge (`GET`, `POST`, `PUT`, `DELETE`), the path, and a one-line description. Clicking an entry loads its detail in the right pane.

10.9.2. Detail pane

For the selected endpoint the right pane shows:

- The verb and path, the human-readable title and the long description.
- Badges for **Auth required** / **No auth**, and a **Full Docs** link that jumps to the relevant page in this manual.
- **Path Parameters**, **Query Parameters** and **Request Headers** tables.

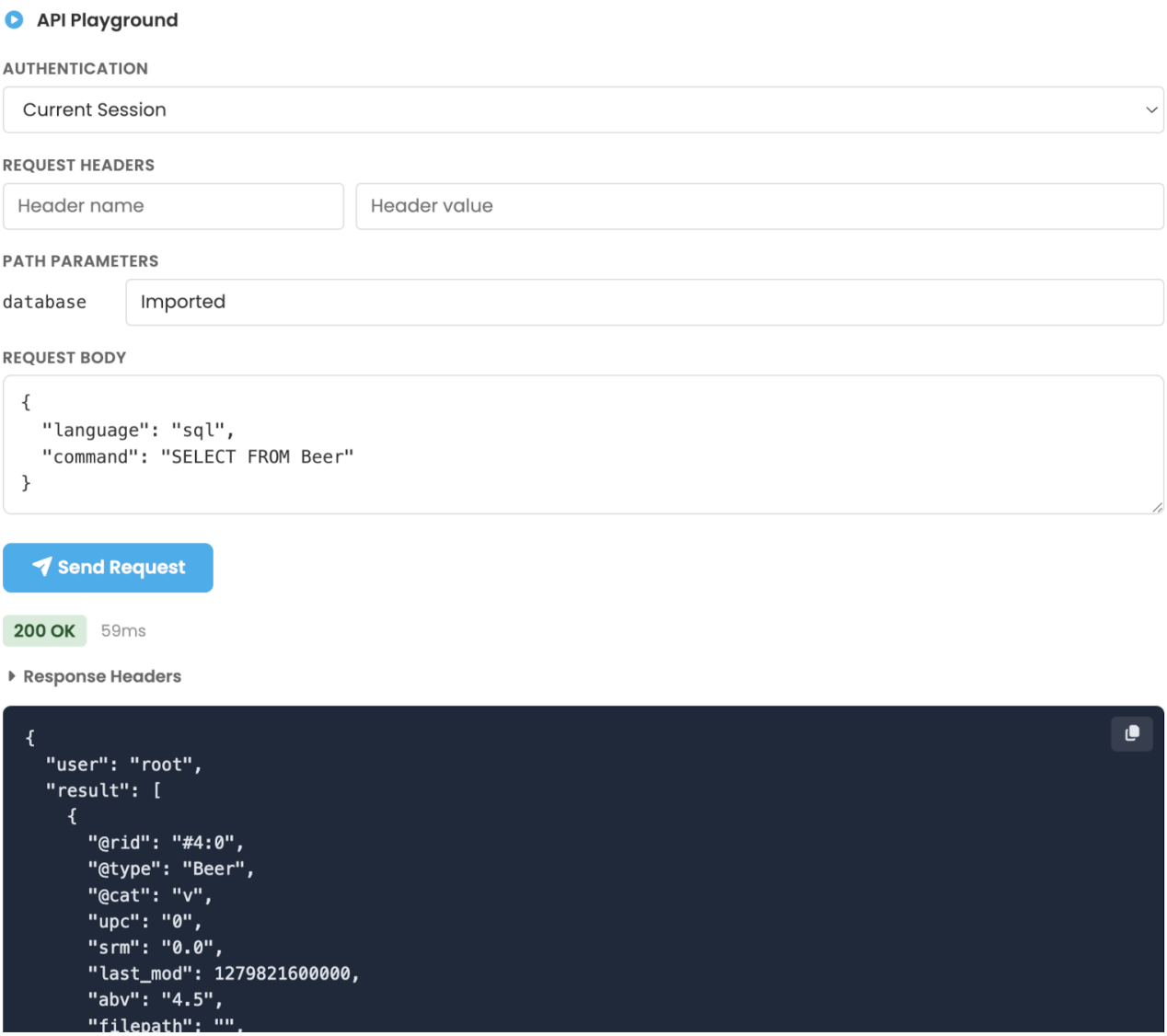
- The **Request Body** example (pretty-printed).
- The **Response** example with the typical status code, body and optional headers.
- Implementation **Notes** — quirks, defaults, references to related endpoints.

10.9.3. Try It playground

Toggle **Try It** to reveal a request playground for the selected endpoint:

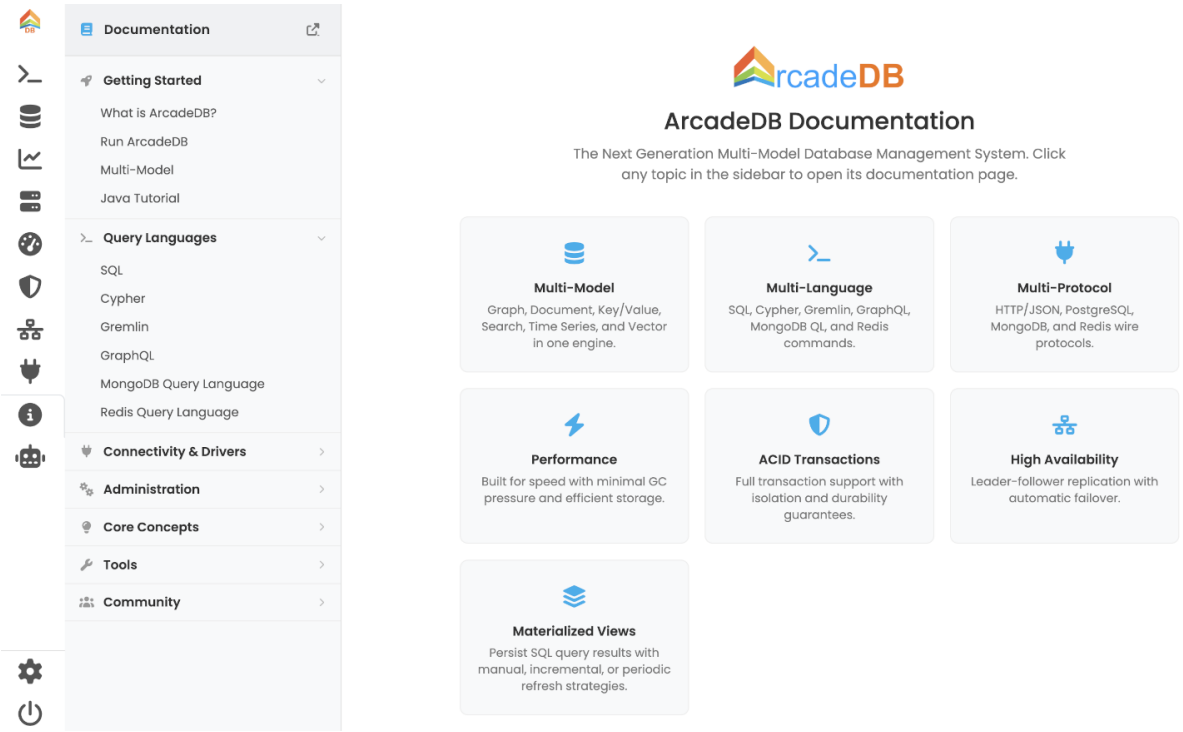
- **Auth** — pick **Current Session** (reuse the Studio login), **Basic** (username + password), **Bearer** (API token) or **None**.
- **Request Headers** — add custom headers.
- **Path Parameters** and **Query Parameters** — inputs are pre-filled with sensible defaults where applicable.
- **Request Body** editor with syntax highlighting.
- **Send Request** — execute and view the response status, headers (collapsible) and body, with a **Copy** button.

For transactional flows the playground automatically captures the `arcadedb-session-id` returned by `POST /api/v1/begin/{db}` and re-injects it in the following `command`, `commit` and `rollback` calls — so you can walk through a complete transaction without copy-pasting the session header.



10.10. Info Panel

The **Info** panel is a quick-reference launcher into this documentation and a short feature overview, all without leaving Studio.



10.10.1. Documentation sidebar

The left sidebar is a collapsible table of contents grouped by topic:

- **Getting Started** — What is ArcadeDB, Run, Multi-Model overview, Java Tutorial.
- **Query Languages** — SQL, Cypher, Gremlin, GraphQL, MongoDB, Redis.
- **Connectivity & Drivers** — HTTP, Java, JDBC, Python, PostgreSQL wire, MongoDB wire, Redis wire.
- **Administration** — Installation, Docker, Kubernetes, Backup, Security, High Availability, Settings.
- **Core Concepts** — Graph, Schema, Indexes, Transactions, Materialized Views.
- **Tools** — Console, Importer.
- **Community** — GitHub, Issues, Discussions.

Each entry opens the corresponding page on `docs.arcadedb.com` in a new browser tab.

10.10.2. Feature overview

The right pane shows a welcome card and a grid of feature cards (Multi-Model, Multi-Language, Multi-Protocol, Performance, ACID Transactions, High Availability, Materialized Views) plus **Quick Links** to the full documentation, the GitHub repository and the community discussions.

10.11. AI Assistant Panel

The AI Assistant Panel turns Studio into a chat-driven workspace where you can ask questions about your database in natural language and get answers, schema insights, and executable queries back. The assistant can read your schema, run read-only queries on your behalf, and propose

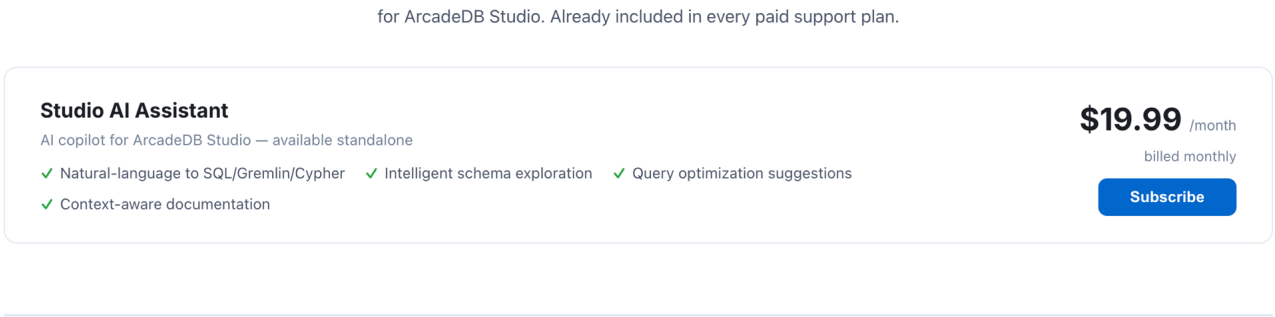
commands you can execute with one click directly from the conversation.

Typical use cases include:

- Schema review and optimization
- Query optimization and troubleshooting
- Data modeling recommendations
- Server profiling analysis
- Synthetic data generation
- Translating natural language into SQL, Apache Tinkerpop Gremlin or Open Cypher

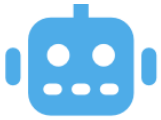
10.11.1. Subscription

The AI Assistant is a paid add-on for ArcadeDB Studio. You can subscribe to it standalone or get it bundled with any ArcadeDB paid support plan from the Pricing page.



10.11.2. Activation

The first time you open the **AI Assistant** tab from the Studio main menu, you are asked to activate your subscription. Enter the subscription key you received after purchase and click **Activate**.



AI Assistant

Get AI-powered database assistance:

- Schema review and optimization
- Query optimization and troubleshooting
- Data modeling recommendations
- Server profiling analysis
- Synthetic data generation
- And more...

Activate your subscription

Enter your subscription key to enable the AI assistant.

Enter your subscription key | Activate

Don't have a key? Get one here.

If you do not have a key yet, click **Get one here** to be redirected to the ArcadeDB pricing page.

When you activate, you are shown the AI Assistant Terms of Use. You must accept them to start using the service. The key points are:

- **Data transmission.** To provide AI-powered assistance, information from your database — such as schema definitions, metadata, query structures and small data excerpts — is transmitted to a third-party Large Language Model (LLM) provider for processing.
- **No training on your data.** Your data is never used to train any AI or machine learning model and is not retained by the LLM provider beyond the duration of the request.
- **Use of test data recommended.** For maximum privacy protection, use the AI Assistant with non-production, test, or anonymized datasets whenever possible.
- **User responsibility.** You are responsible for ensuring that your use of the AI Assistant complies with all applicable data protection regulations (GDPR, CCPA, and equivalent legislation).

Once activated, the subscription is stored on the ArcadeDB server and is shared by all Studio users connected to it.

10.11.3. Using the assistant

After activation, the panel switches to chat mode. The layout is composed of:

- **Chat history** on the left, grouping past conversations as **Today**, **Yesterday**, **This Week** and **Older**. You can create a new chat with **New Chat** or delete a previous one with the trash icon.
- **Database selector** at the top of the chat area. The assistant only operates on the database you select here, so make sure to pick the right one before sending a message.
- **Mode toggle** at the top right, see Modes.
- **Messages area** with the conversation between you and the assistant.
- **Input box** at the bottom: type your question and press **Enter** (use **Shift+Enter** to insert a newline) or click **Send**. While a response is being generated, the **Send** button turns into a **Stop** button you can use to cancel the request.

The assistant renders answers in a fixed subset of Markdown. *(Since v26.10.2.)* Supported are paragraphs and line breaks, headings (`#` to `######`), bullet and numbered lists, block quotes, horizontal rules, bold, italic and inline code, fenced code blocks (shown with their language as a label and a copy button, without syntax highlighting), pipe tables, and links whose URL is `http` or `https`. Everything else is shown as text: raw HTML, links with any other scheme, and autolinks. An image is shown as a plain link, so nothing is loaded from its URL. Nested lists and nested quotes render one level deep. An answer quotes type names and values from your database, so it is never rendered as HTML.

10.11.4. Modes

The AI Assistant supports two modes, switchable at any time from the toggle at the top right of the panel. The selected mode is remembered across sessions.

- **Auto Run Read Only Queries** — the assistant can autonomously execute read-only queries (for example to inspect the schema or sample data) and use the results to compose its answer. Every tool call performed during a turn is shown in the message as a transparent log, so you can always see which queries were run.
- **Review First** — the assistant never runs anything on its own. It can still propose commands, but it is up to you to inspect and run them.

In both modes, **write** operations (inserts, updates, deletes, schema changes) are never executed automatically: they are always returned as command blocks for you to review.

10.11.5. Executing suggested commands

When the assistant proposes one or more commands, each is rendered in a dedicated block showing:

- A short **purpose** describing what the command does

- A language badge (`SQL`, `SQL Script`, `GREMLIN`, `CYPHER`, `MONGO`, `GRAPHQL`)
- The command itself
- Action buttons:
  - **Copy** to copy the command to the clipboard
  - **Open in Query** to switch to the Query panel with the command and language preloaded, ready to be tweaked and executed
  - **Execute** to run the command in place against the currently selected database; the outcome (success with result count, or error) is shown right below the block

If the assistant returns multiple commands, an additional **Execute All** button runs them sequentially in the order they were proposed, stopping at the first error.

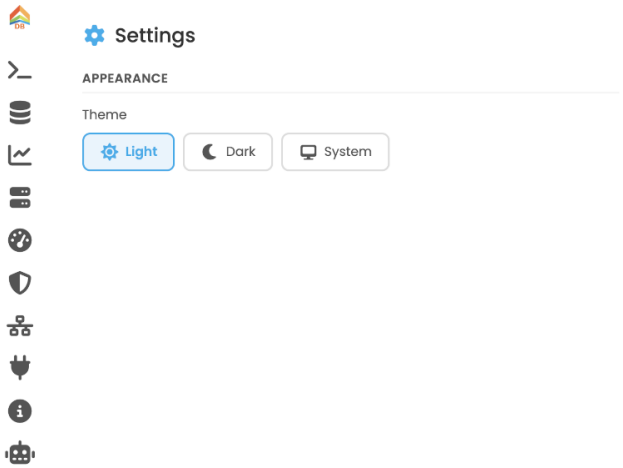
Multi-statement SQL blocks are automatically executed as `sqlscript` so that several statements separated by `;` can run in a single request.

10.11.6. Privacy and security notes

- The assistant only operates on the database explicitly selected in the panel.
- All requests are sent over the same authenticated connection used by the rest of Studio.
- The subscription key is validated on every request; if the key is invalid, expired or has been disabled, the panel switches back to the activation screen.
- For sensitive workloads, prefer the **Review First** mode and use a dedicated read-only Studio user when granting access to the AI Assistant.

10.12. Settings Panel

The **Settings** panel controls Studio's own appearance. It is pinned at the bottom of the icon sidebar.



10.12.1. Appearance

The **Theme** control offers three options:

- **Light** — light theme.

- **Dark** — dark theme.
- **System** — follow the operating-system theme; Studio listens for OS changes and switches on the fly.

The choice is stored in browser local storage (`studio-theme`) and persists across sessions.

Server- and database-scoped configuration lives elsewhere: see Server › Server Settings and Database › Database Settings.

10.13. Console

Run the console by executing `console.sh` under `bin` directory from the `arcadedb` directory:

```
$ bin/console.sh

ArcadeDB Console v26.11.1 - Copyrights (c) 2021 Arcade Data Ltd (https://arcadedb.com)

>
```

A warning `WARNING: Using incubator modules: jdk.incubator.vector` in the log can be safely ignored.

The following command-line arguments are supported by the console:

- `-D` allows to pass settings,
- `-b` enables batch mode which exits after executing all commands passed via the command-lines.
- `-fae` fail-at-end if error occurs during batch (normally batch mode breaks execution at an error).

In batch mode the console exits with code `1` when any command failed, and `0` only when everything succeeded, so a script or CI step can rely on the exit code. This includes a failure raised when the write is committed - a unique-index violation, a full disk - also with `SET asyncMode = true`.

The console supports these direct commands (not evaluated by the query engine):

Console	HELP / ?	QUIT / EXIT	PWD
Server	CONNECT	CLOSE	LIST DATABASES
User	CREATE USER	DROP USER	
Database	CREATE DATABASE	DROP DATABASE	CHECK DATABASE
Schema	INFO TYPES	INFO TYPE	INFO TRANSACTION
Transaction	BEGIN	COMMIT	ROLLBACK
Scripting	SET LANGUAGE	LOAD	`--` (Comment)

for which you can retrieve the following help by typing `HELP` or just `?`:

```
begin                                                          -> begins a new
transaction
check database                                                 -> check database
integrity
close |<path>|remote:<url> <user> <pw>                         -> closes the database
commit                                                         -> commits current
transaction
connect <path>|local:<url>|remote:<url> <user> [<pw>]          -> connects to a
database
create database <path>|remote:<url> <user> [<pw>]              -> creates a new
database
create user <user> identified by [<pw>] [grant connect to <db>[:<group>]*] -> creates
a user
drop database <path>|remote:<url> <user> [<pw>]                -> deletes a database
drop user <user>                                               -> deletes a user
help|?                                                         -> ask for this help
info types                                                     -> prints available
types
info type <type>                                               -> prints details about
type
info transaction                                               -> prints current
transaction
list databases |remote:<url> <user> [<pw>]                     -> lists databases
load <path>                                                    -> runs local script
pwd                                                            -> returns current
directory
rollback                                                       -> rolls back current
transaction
set language = sql|sqlscript|cypher|gremlin|mongo|graphql      -> sets console query
language
-- <comment>                                                   -> comment (no
operation)
quit|exit                                                      -> exits from the
console

Omit <pw> to be asked for the password with the echo hidden, so it is never written to
the .history file nor echoed to the output in batch mode.
```

The `close` command can be used without path or url, then it closes the currently connected database.

The password can be left out of `connect`, `create database`, `drop database`, `list databases` and `create user`. The console then asks for it and hides what you type. This is the recommended form: a password written on the command line ends up in the console's `.history` file, and in the build log when the console runs with `-b`. The console masks passwords in both places, but a password you never type in full cannot leak at all. A password containing spaces is accepted.

Long-running maintenance commands (`CHECK DATABASE`, `REBUILD INDEX`, `COMPACT INDEX`, `BACKUP DATABASE`, `IMPORT DATABASE`) render a live progress line while they run, showing the current step and percentage, e.g. `check database fix [step 3/9] Checking vertices 'Account' |======== | 42%`. The same progress is available in Studio and via the HTTP progress endpoint.

It can be useful to wrap longer scripts called via `LOAD myscript.sql` in a transaction via `BEGIN; LOAD myscript.sql; COMMIT;` to ensure completion of write operations.

Every statement in a script run with `LOAD` must end with `;`, except optionally the very last one. A `;` is what tells `LOAD` a statement is complete, so it can also span multiple lines.

With `SET LANGUAGE = sqlscript`, `LOAD` sends the whole file as a single script instead of one statement at a time, so `LET` variables, `FOREACH`, `IF` and `WHILE` work across statements exactly as in Studio's SQL Script language.

The comment command `--` is useful for scripts.

The console does line-by-line processing, so commands and queries need to be in a single line.

The console passes backslashes to the query language untouched, so a statement means the same typed at the prompt, passed with `-b`, or replayed with `LOAD` as it does over the HTTP API. Escaping is the query language's business: in SQL a string literal needs two backslashes for one, so a Windows path is written `'C:\\Users\\bob'` and a regular expression `'\\d+'`.

The multi-line comment syntax `/* ··· */` is not supported in the console, since the console reads from the terminal and script files line by line. It is required and recommended to use the single-line comment syntax `--` for inline code docu or headers.

10.13.1. Tutorial

Let's create our first database, located in our local filesystem. To identify the database you specify in a connection string. It could be either absolute, or relative to the database directory. The database directory can be explicitly set, when you start the console, like

```
$ ./bin/console.sh -Darcadedb.server.databaseDirectory=/databases
```

If no database directory is given at startup, it is the `databases` directory under the root path of the arcadedb server by default. To create a new database under the database directory, you would run the following command in the console

```
> CREATE DATABASE mydb
```

If you want to create your database somewhere else in the local filesystem, you can specify a different path to the database:

```
> CREATE DATABASE data/mydb
```

Once the database has been created, you can simply connect to it:

```
> CONNECT mydb
```

Or if on a different path, specify the path where the database is located:

```
> CONNECT data/mydb
```

If you're using the ArcadeDB Server, you can create the database through the server:

```
> CREATE DATABASE remote:localhost/mydb root arcadedb-password
```

Once created, you can always connect to the server with the following command:

```
> CONNECT remote:localhost/mydb root arcadedb-password
```

Let's also create a user with access to "mydb":

```
> CREATE USER albert IDENTIFIED BY einstein GRANT CONNECT TO mydb
```

Now let's create a "Profile" type:

```
{mydb}> CREATE DOCUMENT TYPE Profile
```

```
+---------+--------------------+
|NAME     |VALUE               |
+---------+--------------------+
|operation|create document type|
|typeName |Profile             |
+---------+--------------------+
Command executed in 99ms
```

Check your new type is there:

```
{mydb}> INFO TYPES
```

```
AVAILABLE TYPES
+-------+--------+-----------+-------------------------------------+-----------
+-------------+
|NAME   |TYPE    |SUPER TYPES|BUCKETS                              |PROPERTIES |SYNC
STRATEGY|
+-------+--------+-----------+-------------------------------------+-----------
+-------------+
|Profile|Document|0[]        |8[Profile_0,...,Profile_7]|0[]       |round-robin|
|
+-------+--------+-----------+-------------------------------------+-----------
+-------------+
```

You can also query the defined types by executing the following SQL query: `select from schema:types`.

Finally, let's create a document of type "Profile":

```
{mydb}> INSERT INTO Profile SET name = 'Jay', lastName = 'Miner'
```

```
DOCUMENT @type:Profile @rid:#1:0
+--------+-----+
|NAME    |VALUE|
+--------+-----+
|name    |Jay  |
|lastName|Miner|
+--------+-----+
Command executed in 17ms
```

You can see your brand new record with RID `#1:0`. Now let's query the database to see if our new document can be found:

```
{mydb}> SELECT FROM Profile
```

```
DOCUMENT @type:Profile @rid:#1:0
+--------+-----+
|NAME    |VALUE|
+--------+-----+
|name    |Jay  |
|lastName|Miner|
+--------+-----+
Command executed in 37ms
```

Here we go: our document is there.

Remember that a transaction is automatically started. In order to make changes persistent, execute a `COMMIT` command. When the console exists (`exit` or `quit` command), the pending transaction is committed automatically.

10.13.2. Settings

The console provides specific settings accessible via:

```
> SET <setting> = <value>
```

The following console settings are available:

Setting	Type	Default
asyncmode	boolean	false
expandresultset	boolean	false
language	string	sql
limit	integer	20
maxmultivalueentries	integer	10
maxwidth	integer	150
transactionbatchsize	integer	0
verbose	integer	3

The `language` setting can be `sql`, `sqlscript`, `cypher`, `gremlin`, `mongo`, or `graphql`.

Other settings can be changed this way, too.

Here is an example using `SQLscript`:

```
{mydb}> SET language = sqlscript
```

```
{mydb}> IF (1 = 1) { \ SELECT 1; \ }
```

10.13.3. Scripting

The console can also run local SQL scripts using the `LOAD` command:

```
$ bin/console.sh -b "LOAD myscript.sql"
```

or passing the commands as string argument:

```
$ bin/console.sh "CREATE DATABASE test; CREATE DOCUMENT TYPE doc; BACKUP DATABASE;
exit;"
```

Make sure to `create database` or `connect` to a database first in the script before using SQL commands.

All commands (of a script) are executed, disregarding if a previous command failed.

10.13.4. Console-Server Interaction

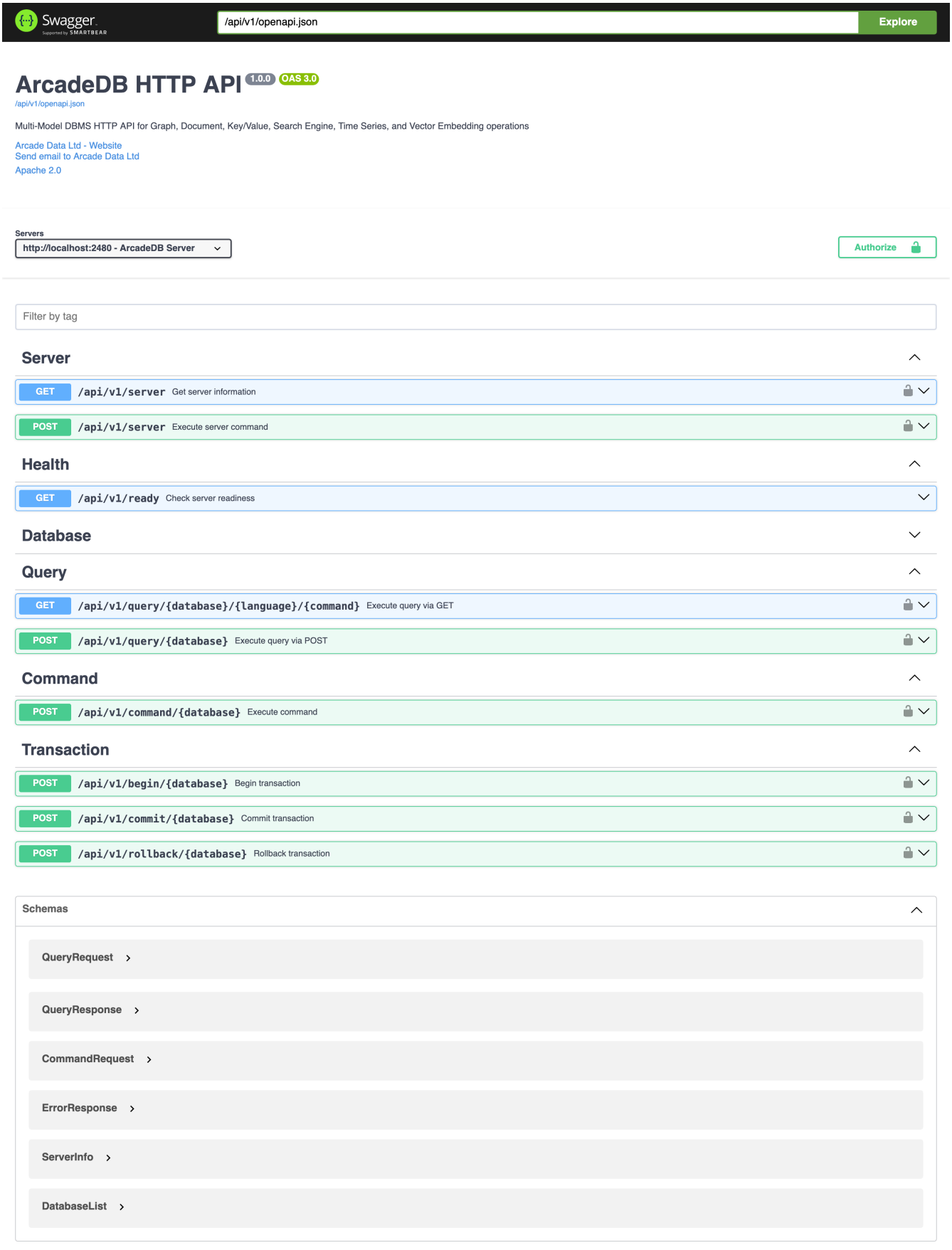
The console cannot access a database via local connection when a server is running.

When the server is running it locks all (opened) databases, hence the console cannot access these databases via local connection which utilizes the file system. Nonetheless, the console can still connect to these databases via a remote connection, particularly, using `localhost` if the console is running on the same machine as the server:

```
> CONNECT remote:localhost/mydb root arcadedb-password
```

10.14. Swagger-UI

To use the Swagger-UI, point your web browser to the server and port that hosts the ArcadeDB server at the `/api/v1/docs` endpoint. For instance, when running ArcadeDB locally with the default port this means: http://localhost:2480/api/v1/docs .



With the Swagger UI you can hands-on explore the HTTP API and test requests.

The page is served with the Content-Security-Policy of `arcadedb.server.httpContentSecurityPolicy`, which refuses inline script by default. The page adds the hash of its own bootstrap script to the directive that governs

> script (`script-src-elem`, else `script-src`, else `default-src`), so a custom policy does not need to allow it. If your policy refuses every script there (`'none'`), the page cannot start: the server logs a warning once and leaves the policy as configured. *(Since v26.10.2.)*

10.15. Compatible Tools

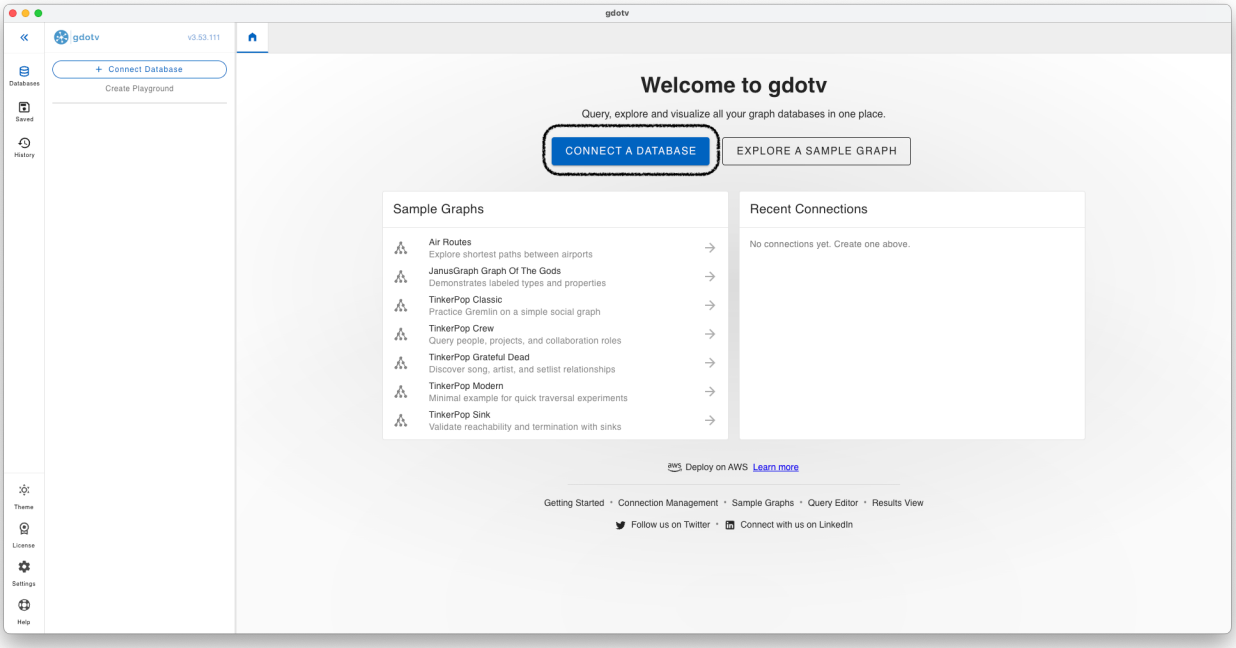
10.15.1. gdotv

ArcadeDB is fully compatible with gdotv, so no particular configurations need to be made. The only requirement is loading the Gremlin plugin via the server setting:

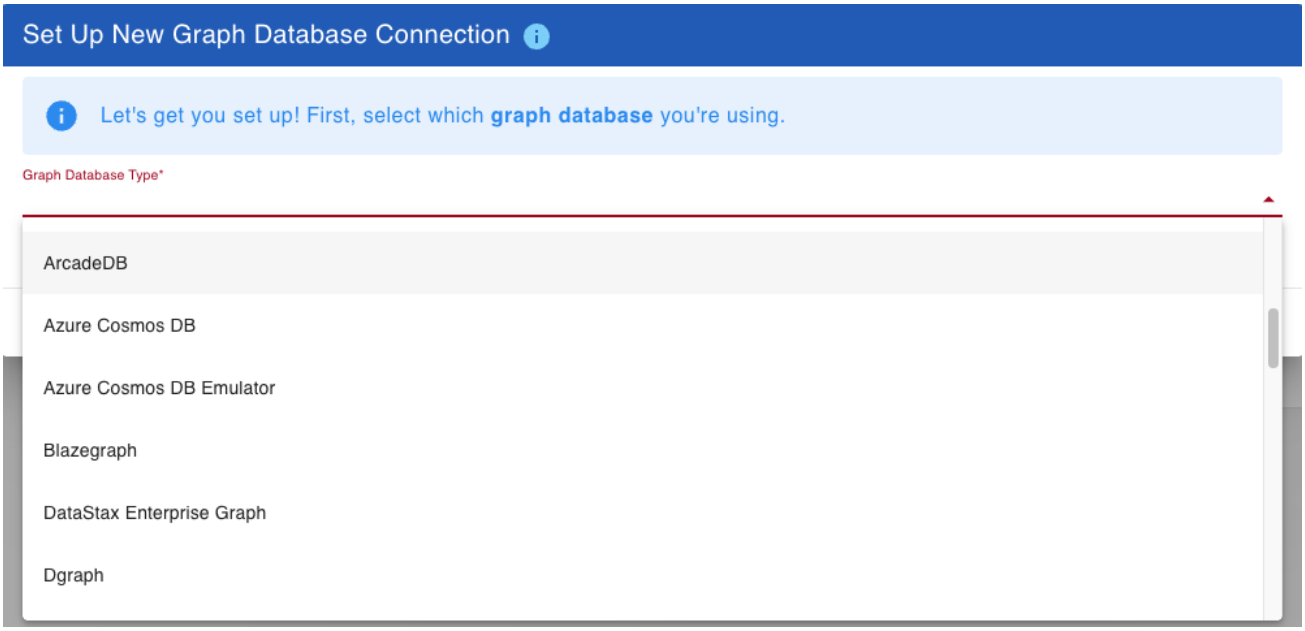
```
"_
Darcadedb.server.plugins=GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugin
"
```

▼ *Details*

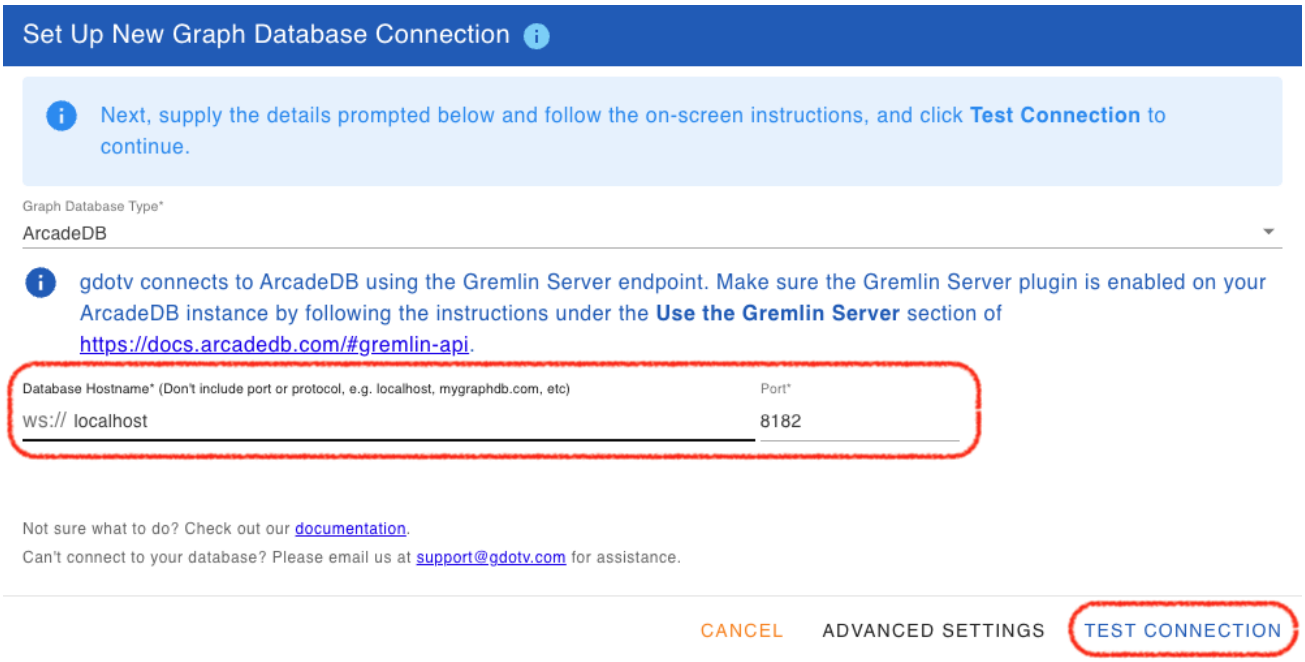
On the title screen, create a new connection:



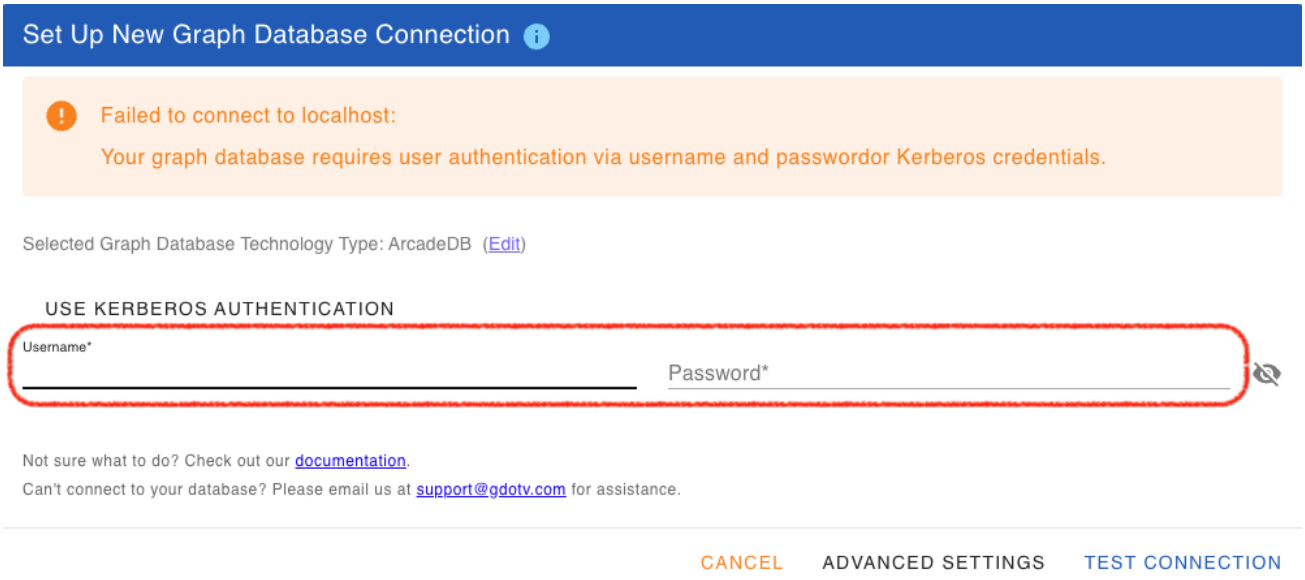
choose ArcadeDB from the list of graph database types:



enter the host name of the ArcadeDB server:



and enter username and password:



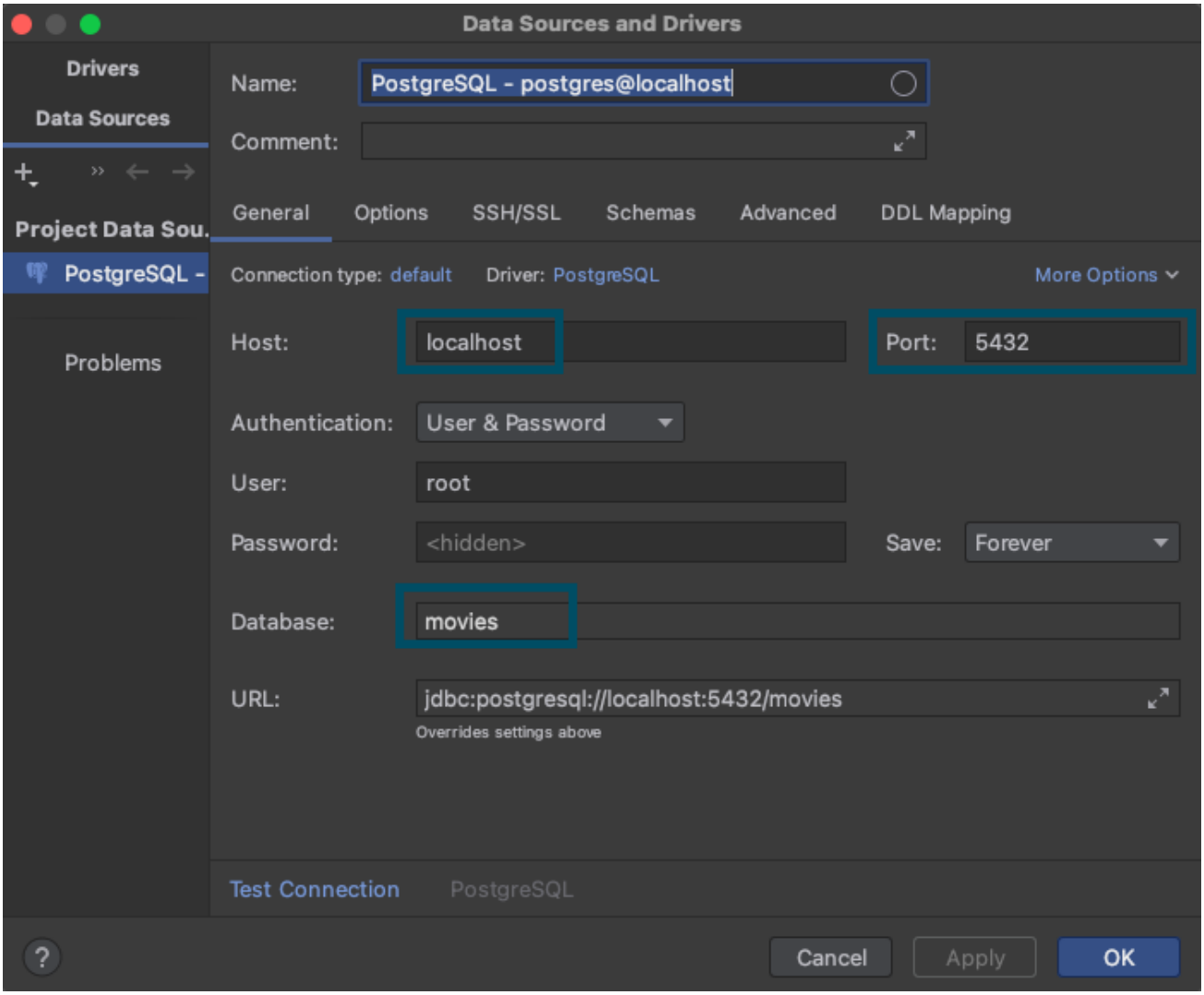
In case of non-standard configurations of the server, under "Advanced Settings" more fine-grained settings can be made.

10.15.2. JetBrains DataGrip/Database Plugin

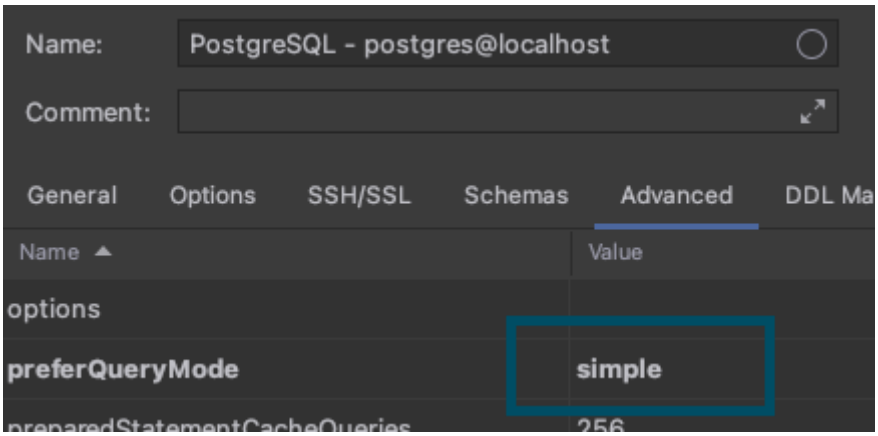
Connecting via JetBrains DataGrip database plugin is relatively straightforward. The introspection features aren't working yet, but the basics seem to work well.

▼ *Details*

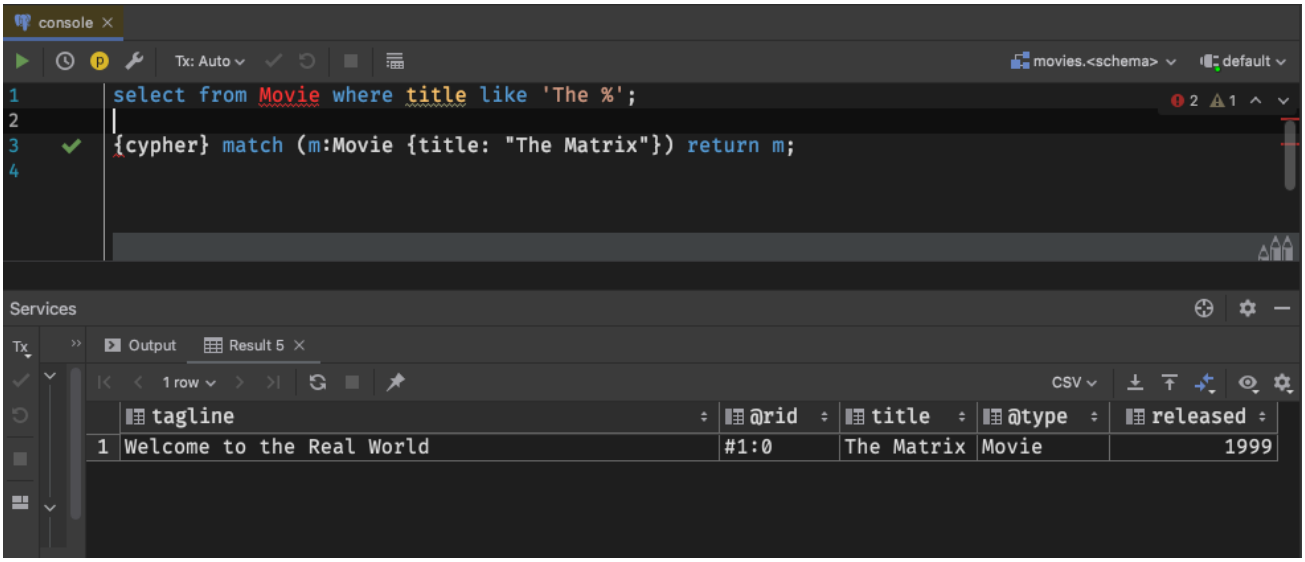
To connect, create a new Postgres datasource and point it to the IP/port of your ArcadeDB server. (0.0.0.0:5432 by default) You will need to fill out the database field, or you'll get an error on connection. At present, changing the current database requires editing the datasource.



Next, you'll need to set `preferQueryMode` to **simple** on the Advanced tab, like this:



You can then run queries via a console. Even non-SQL queries will work, though expect squigglies!

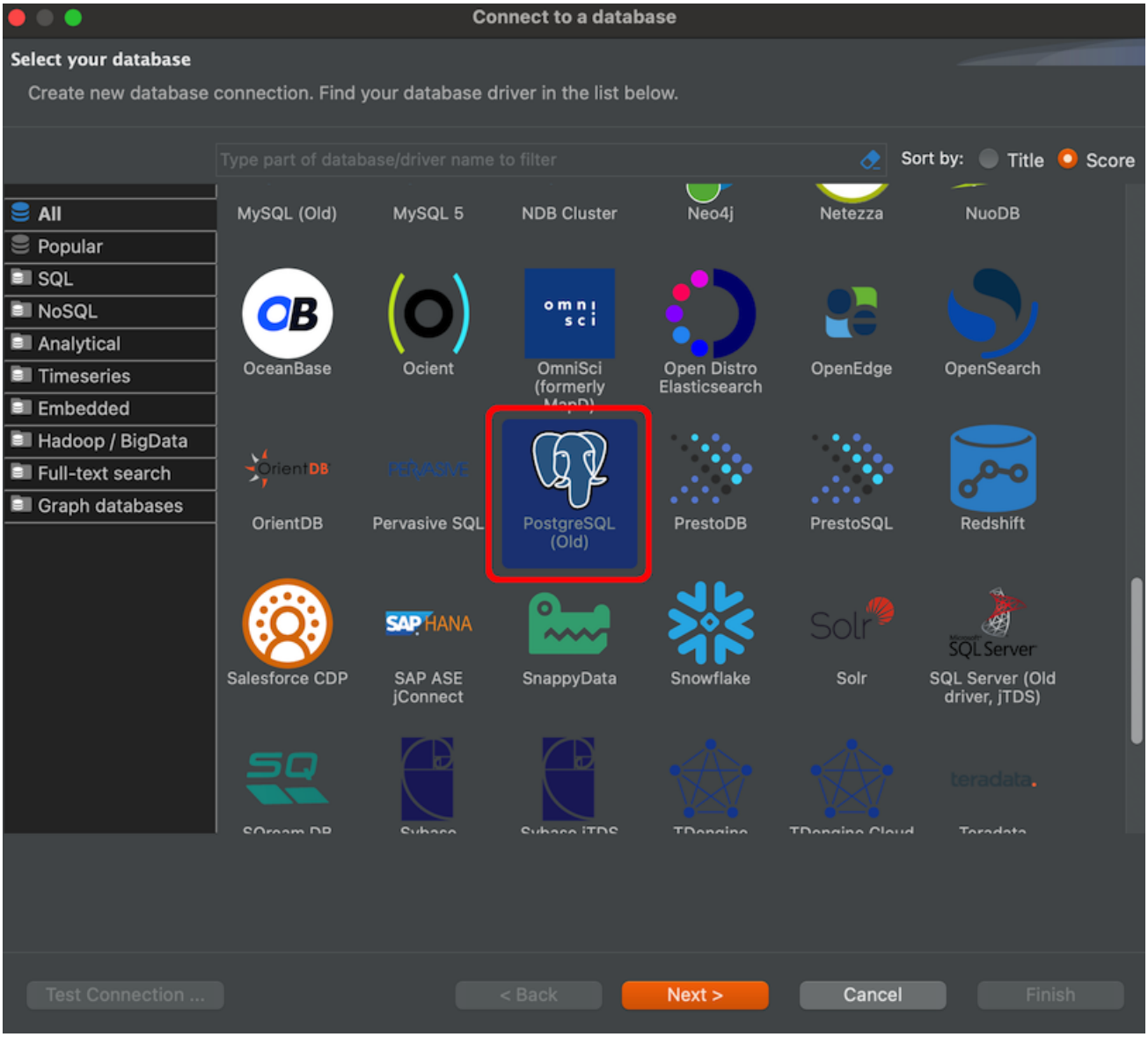


10.15.3. DBeaver

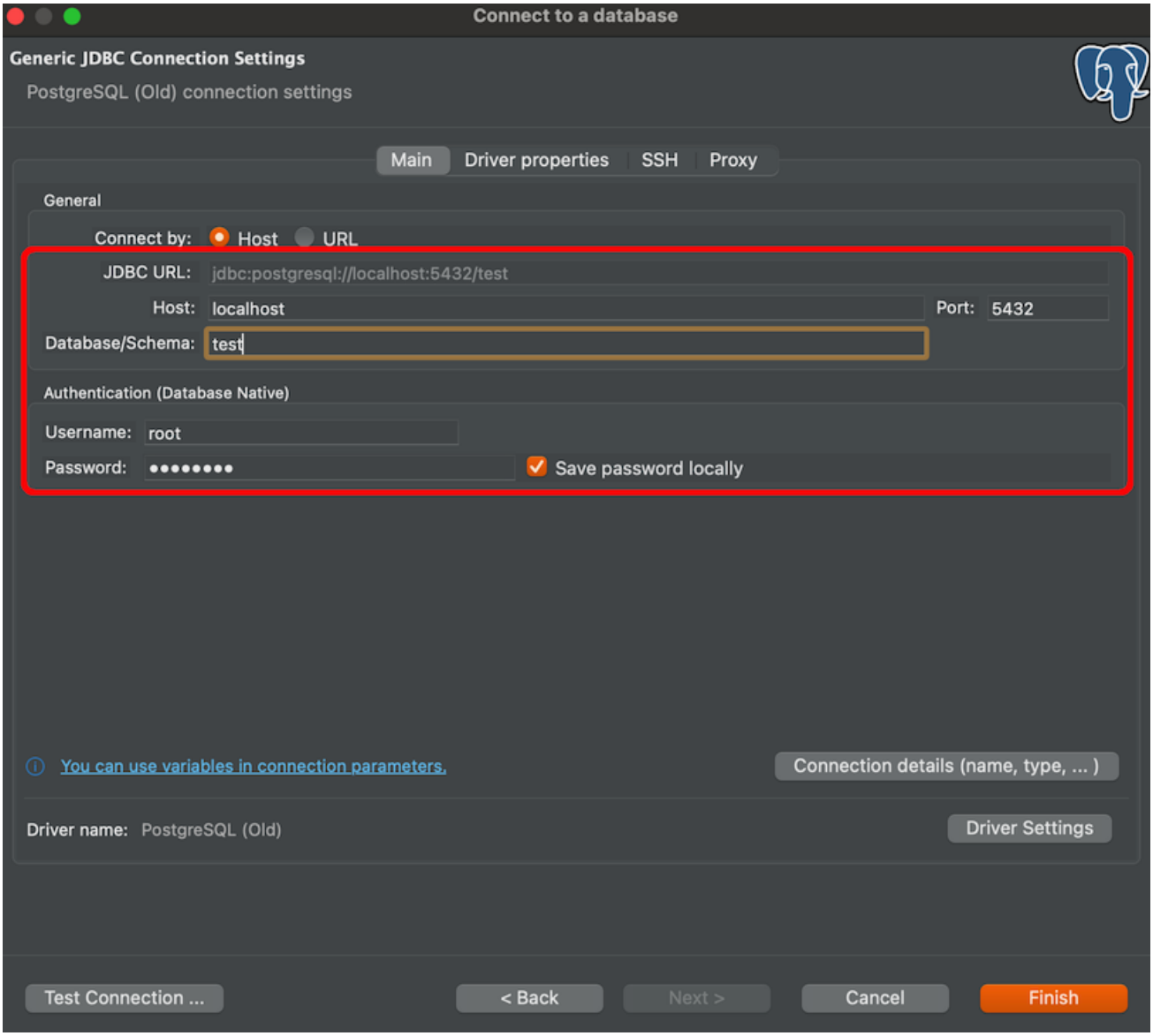
The universal database tool DBeaver has basic compatibility via the legacy Postgres connector.

▼ *Details*

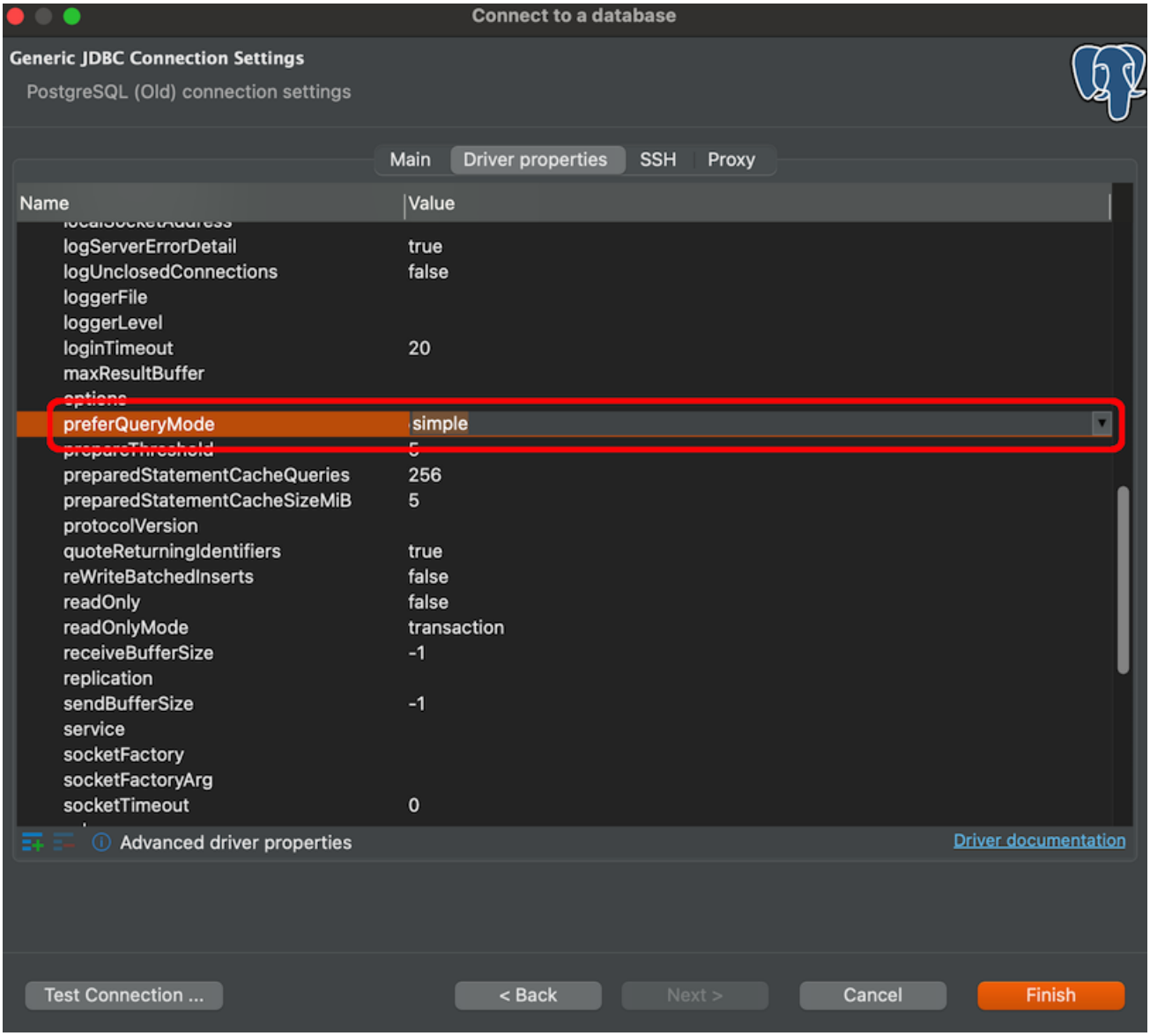
Create a new connection with the "PostgreSQL (Old)" driver:



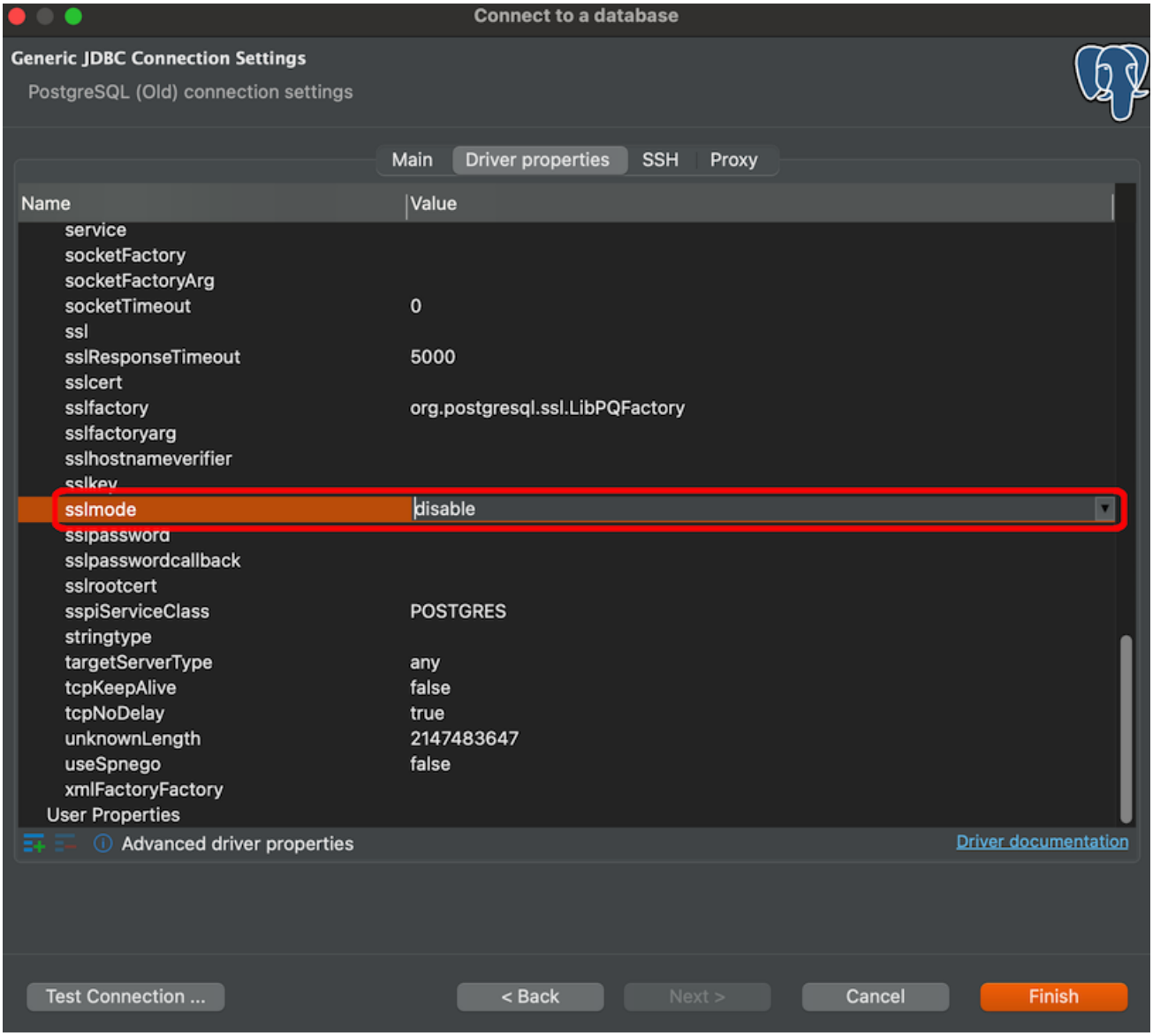
Add your host, port, database, username and password to the general connection settings:



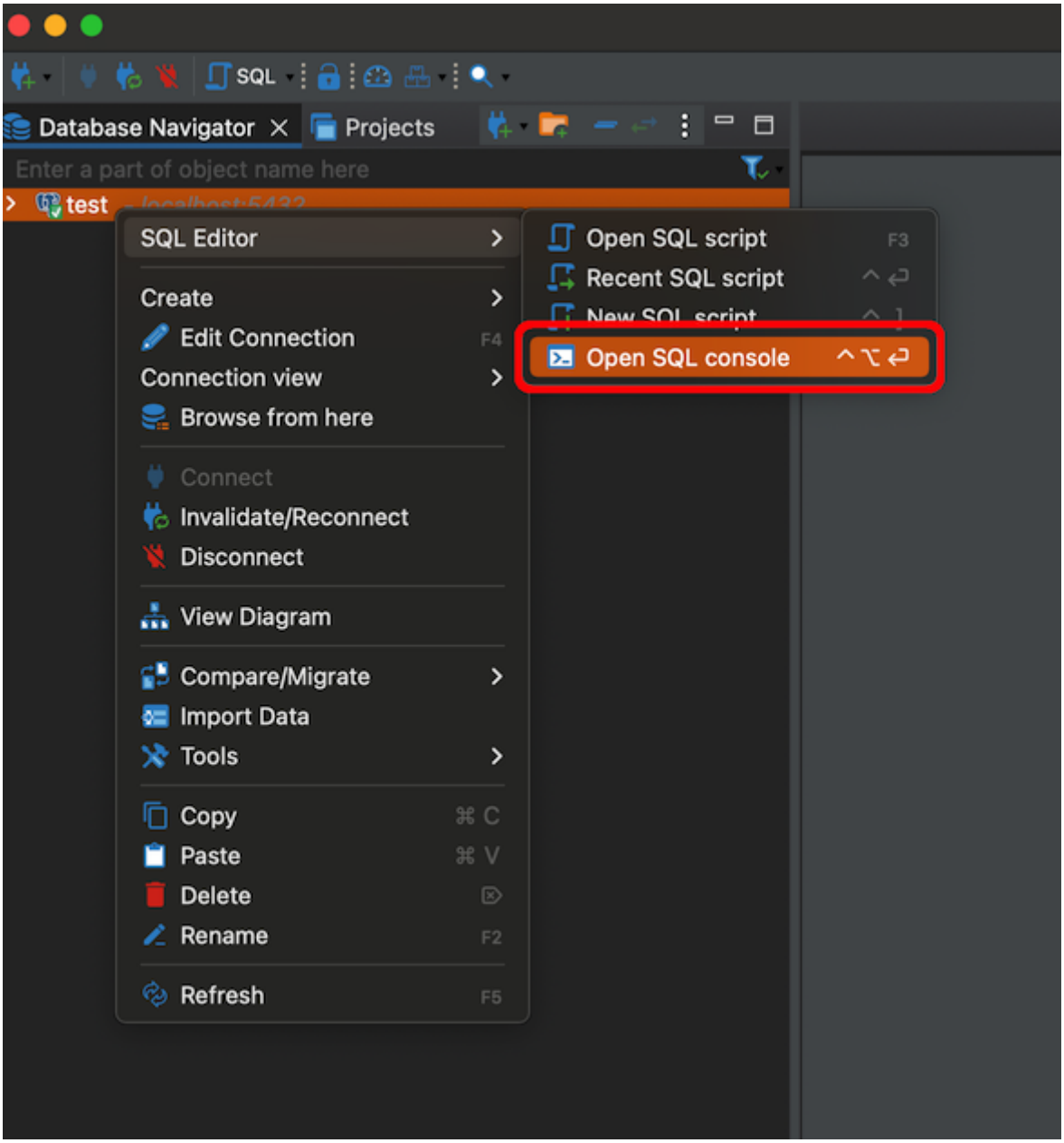
Set the `preferQueryMode` option to `simple` on "Driver Properties" tab:



Set the `sslmode` option to `disable`:



The "Finish" the connection wizard and double click the created connection to connect. Then with a right-click the SQL console can be started:



Now the SQL console can be used to communicate via DBeaver with ArcadeDB.

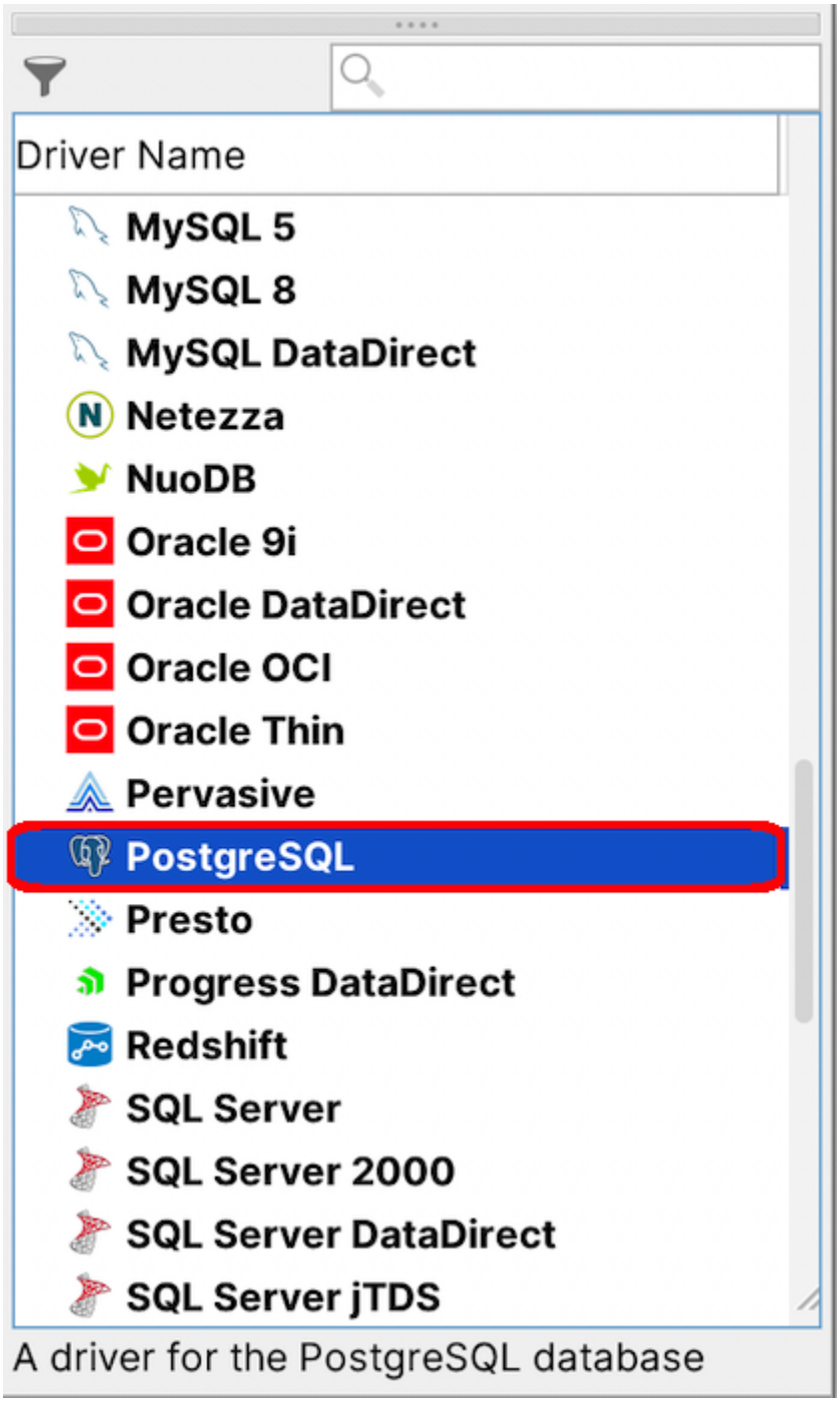
Note that this is only a basic support using a generic relational driver for a NoSQL database, so various functionalities can result in errors.

10.15.4. DbVisualizer

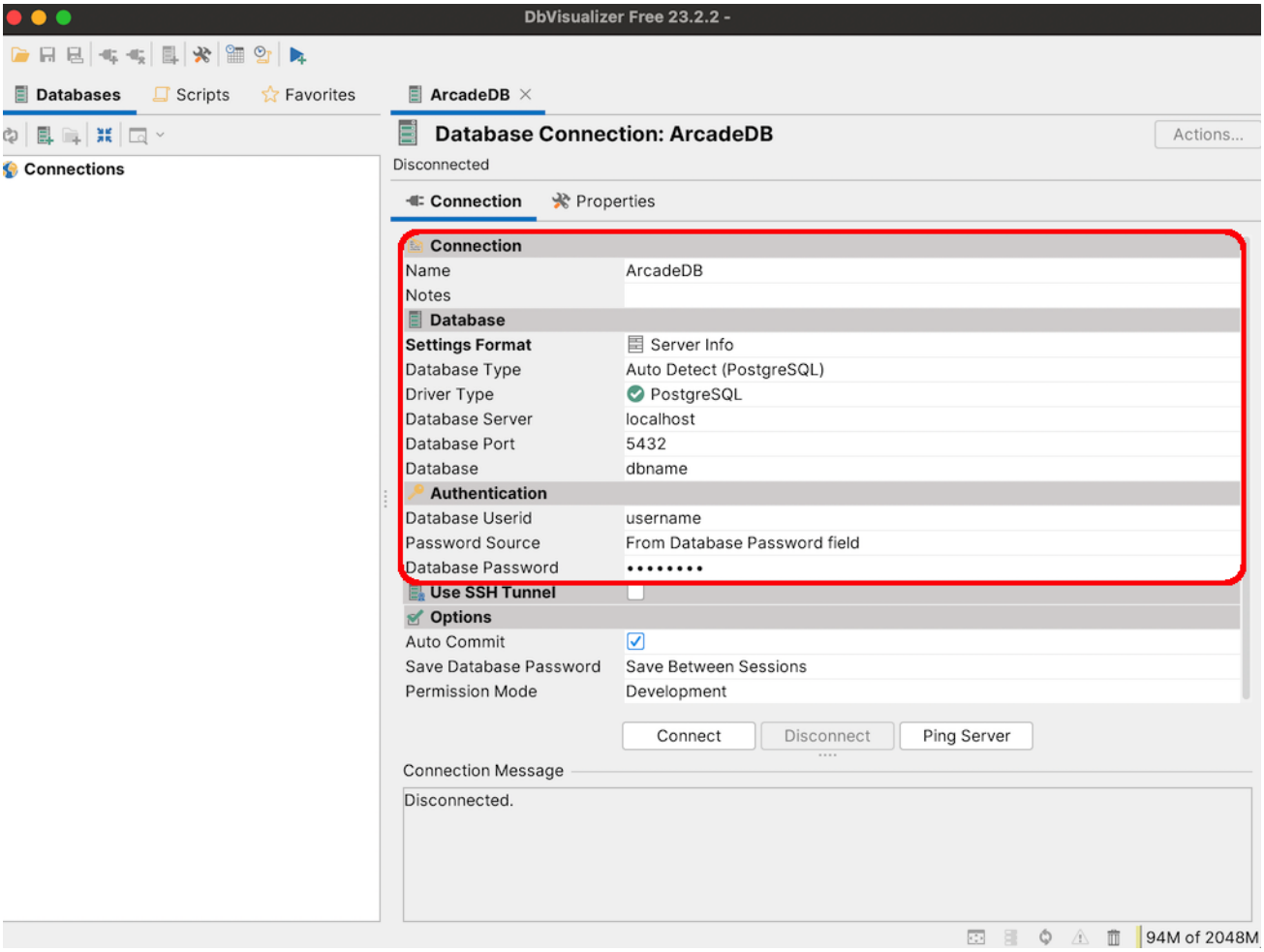
The database client DbVisualizer can also be used via its PostgreSQL driver.

▼ *Details*

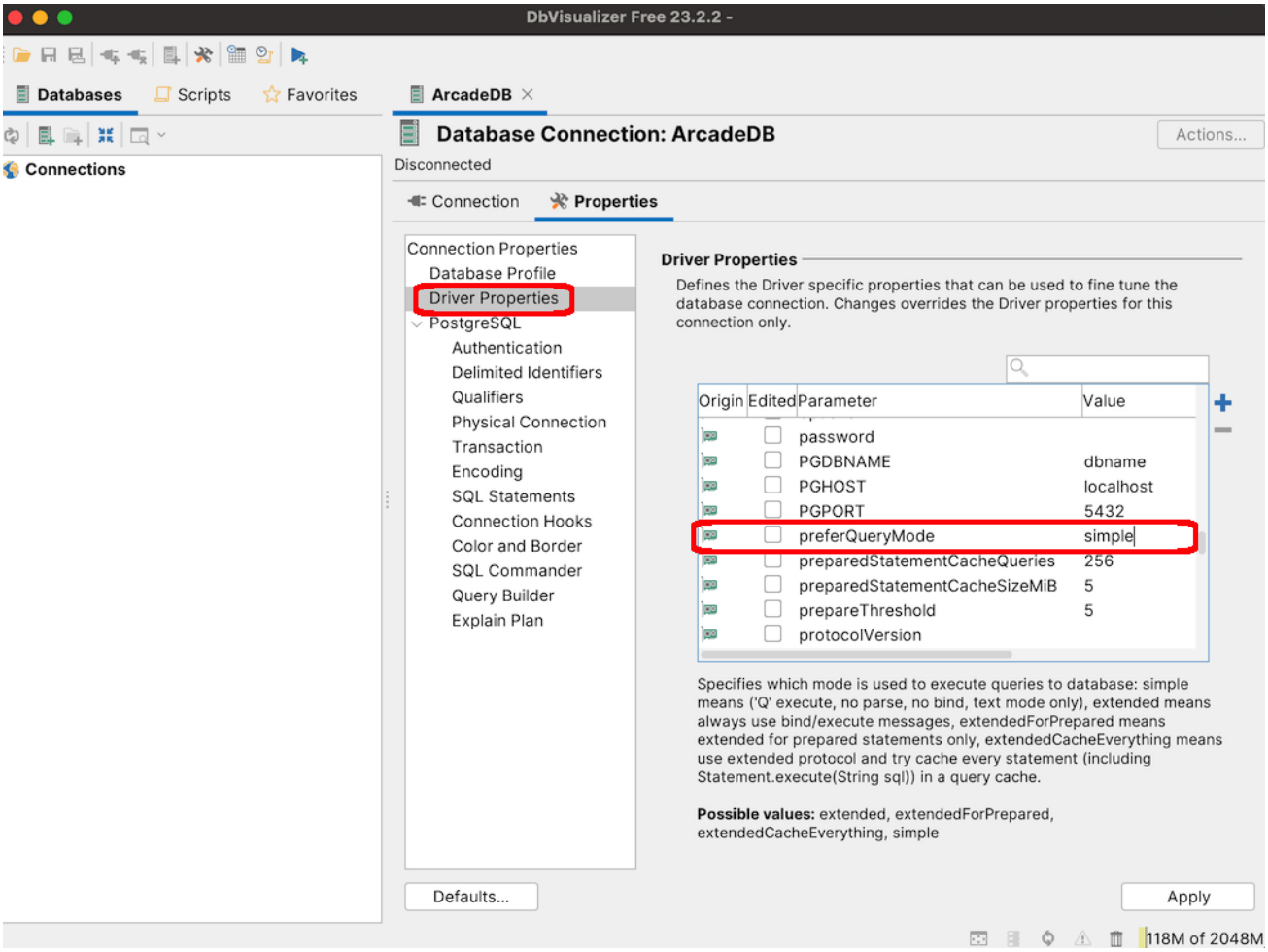
Create a new connection and select "PostgreSQL":



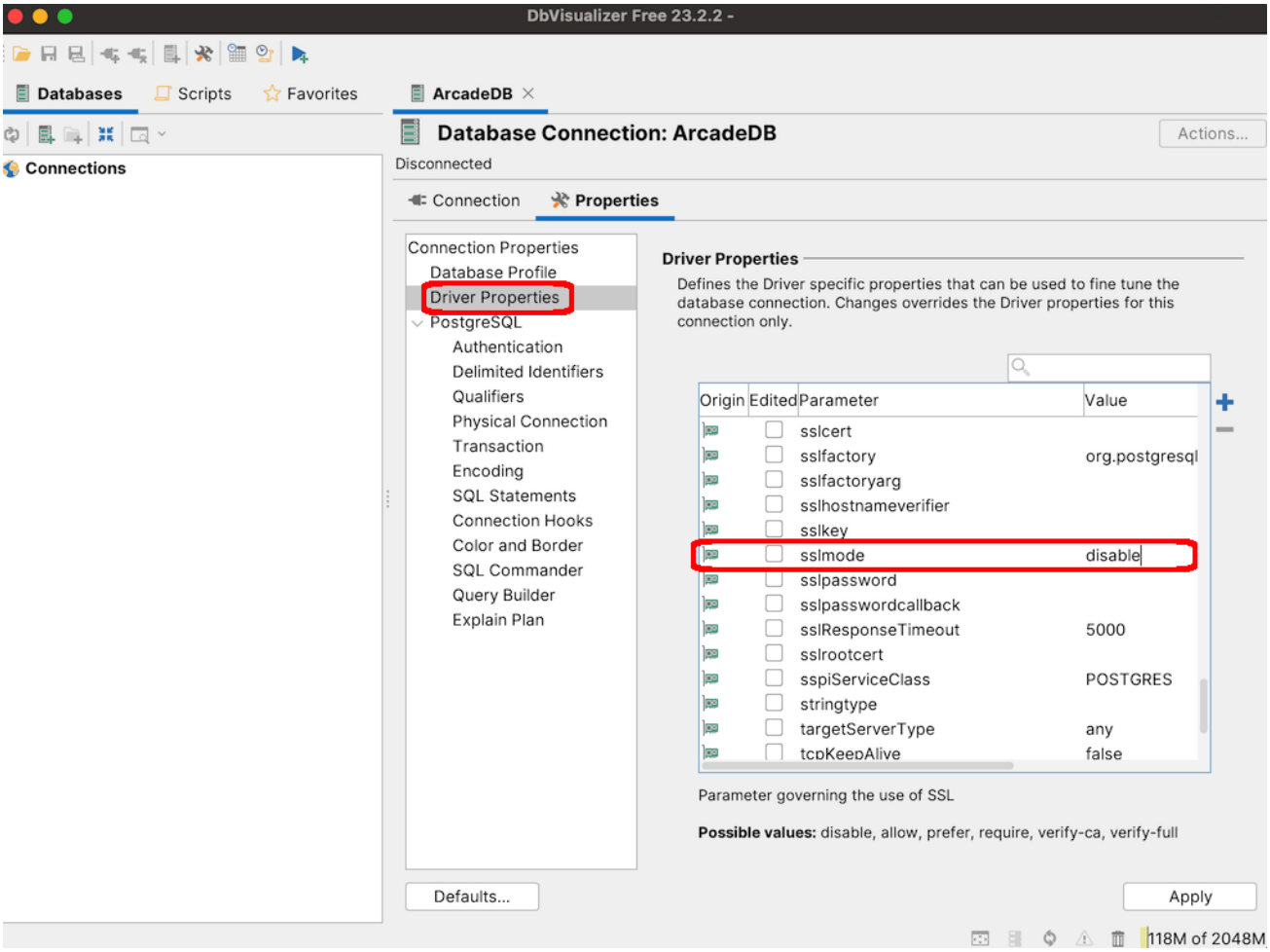
Enter server, port, database, userid, and password:



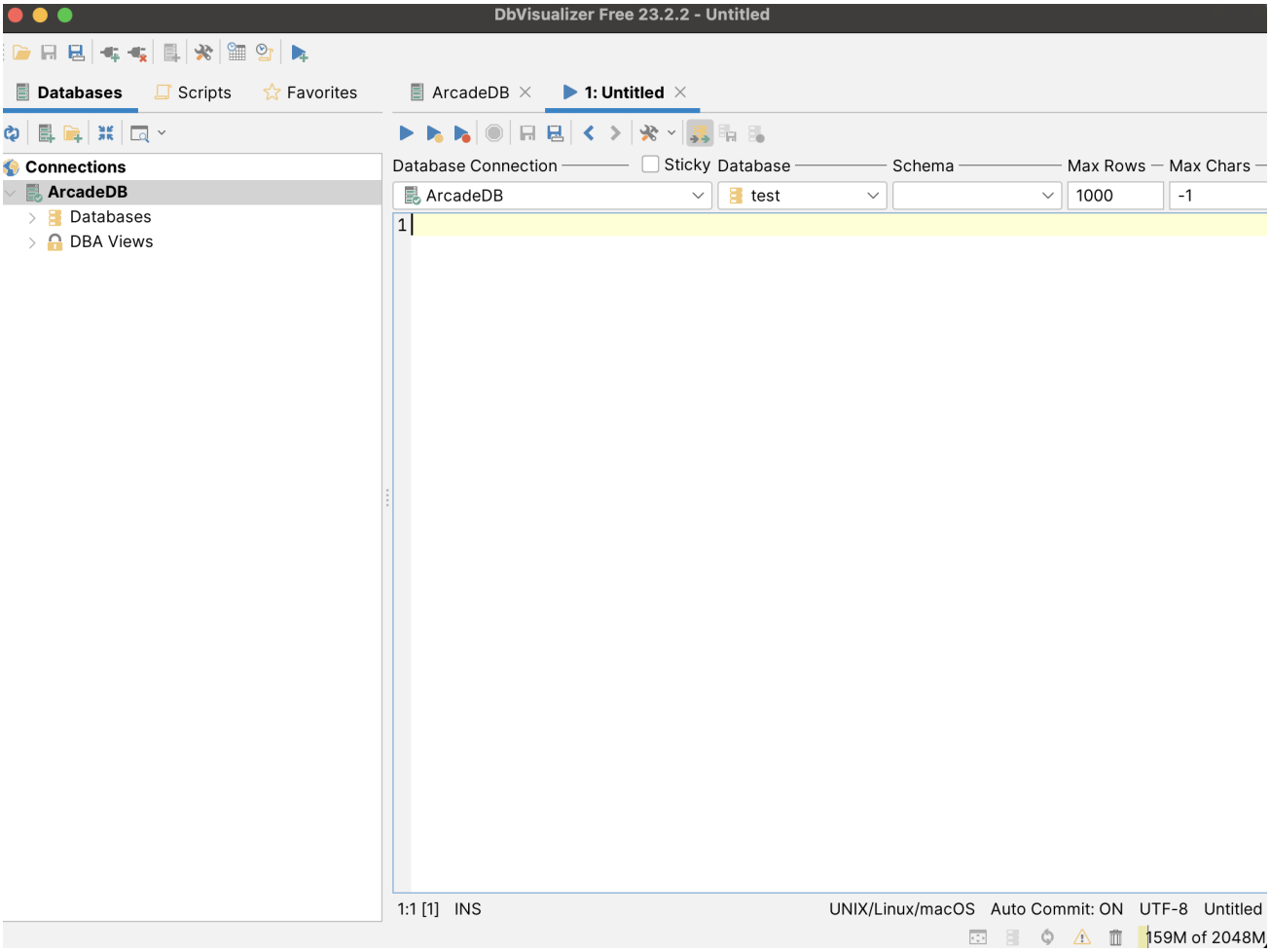
Go to the "Properties" tab and set `preferQueryMode` to `simple`:



Also set `sslmode` to `disable`:



After applying the changes and connecting the SQL commander is available:

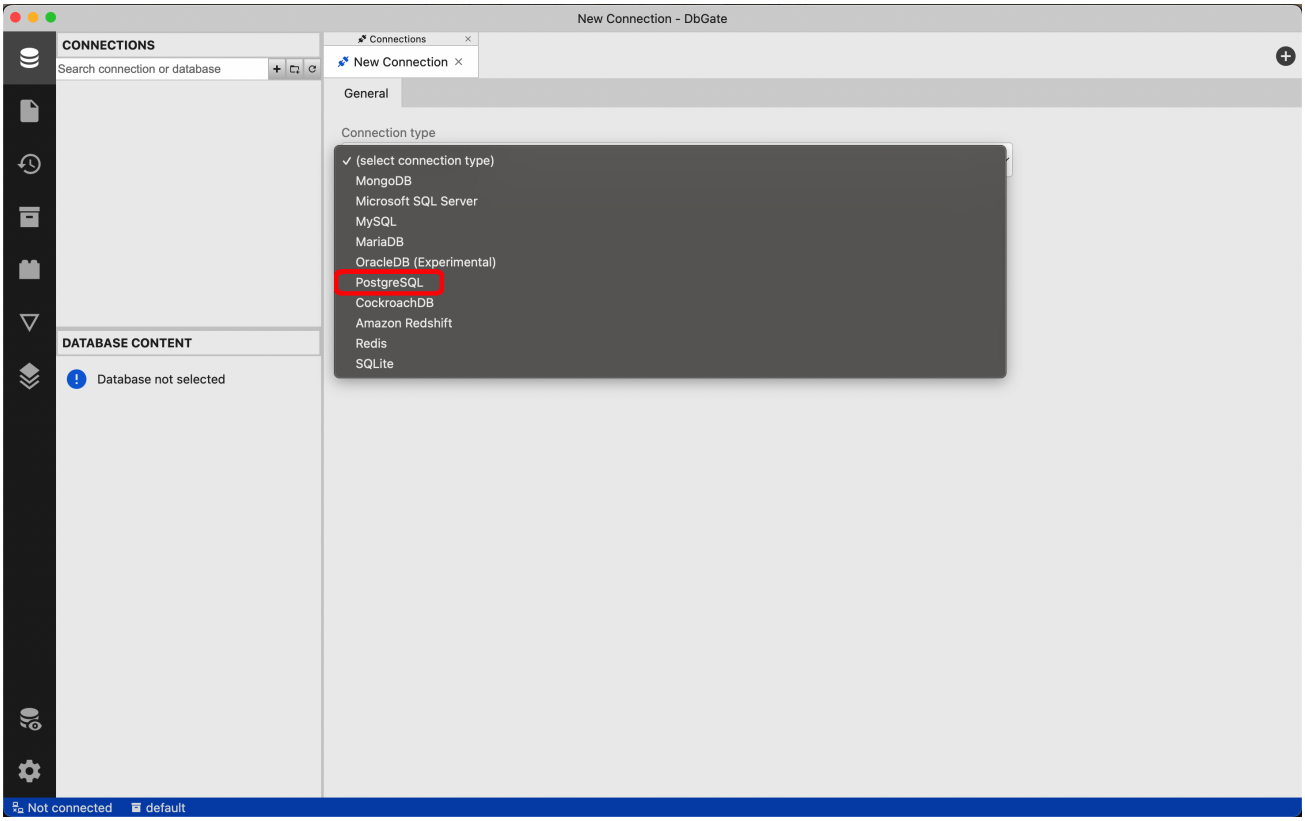


10.15.5. DbGate

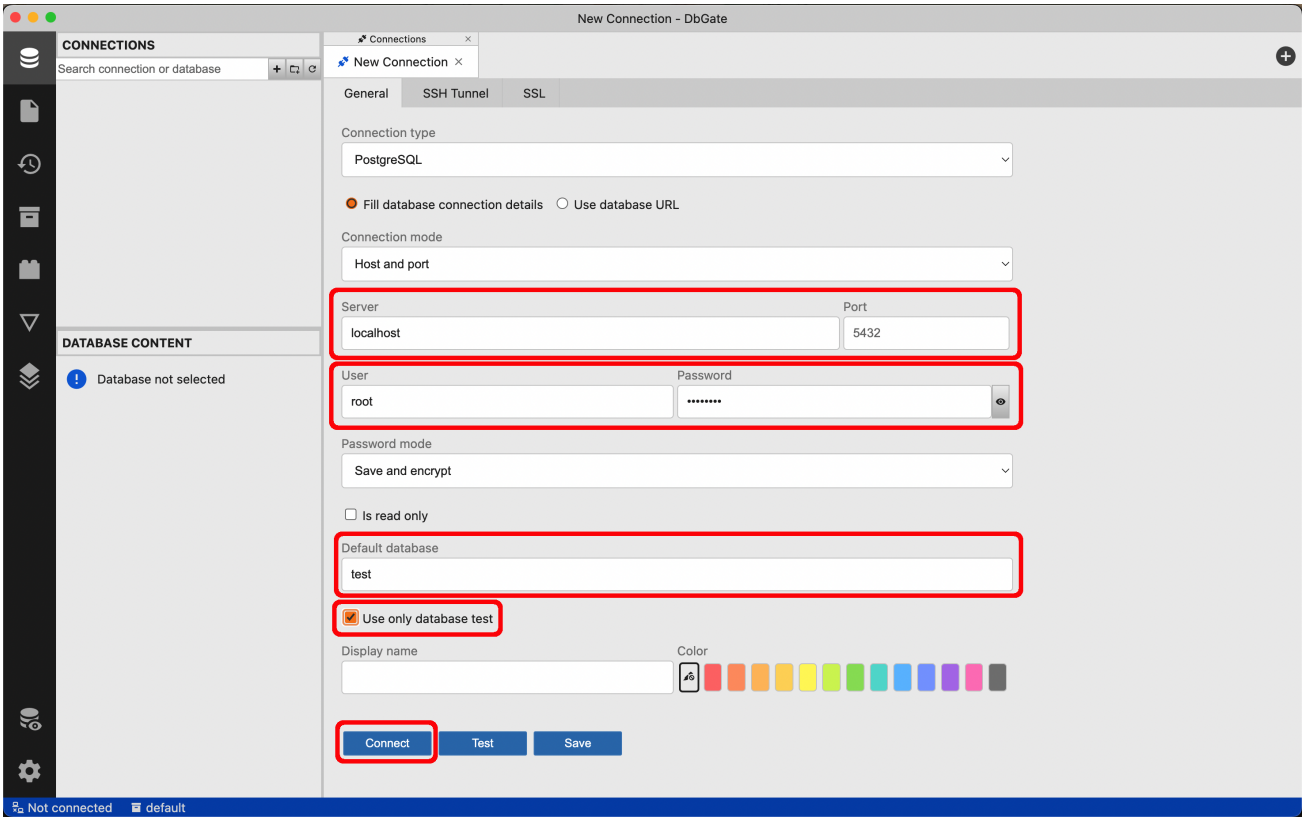
The SQL+noSQL database client DbGate is compatible via the PostgreSQL driver.

▼ *Details*

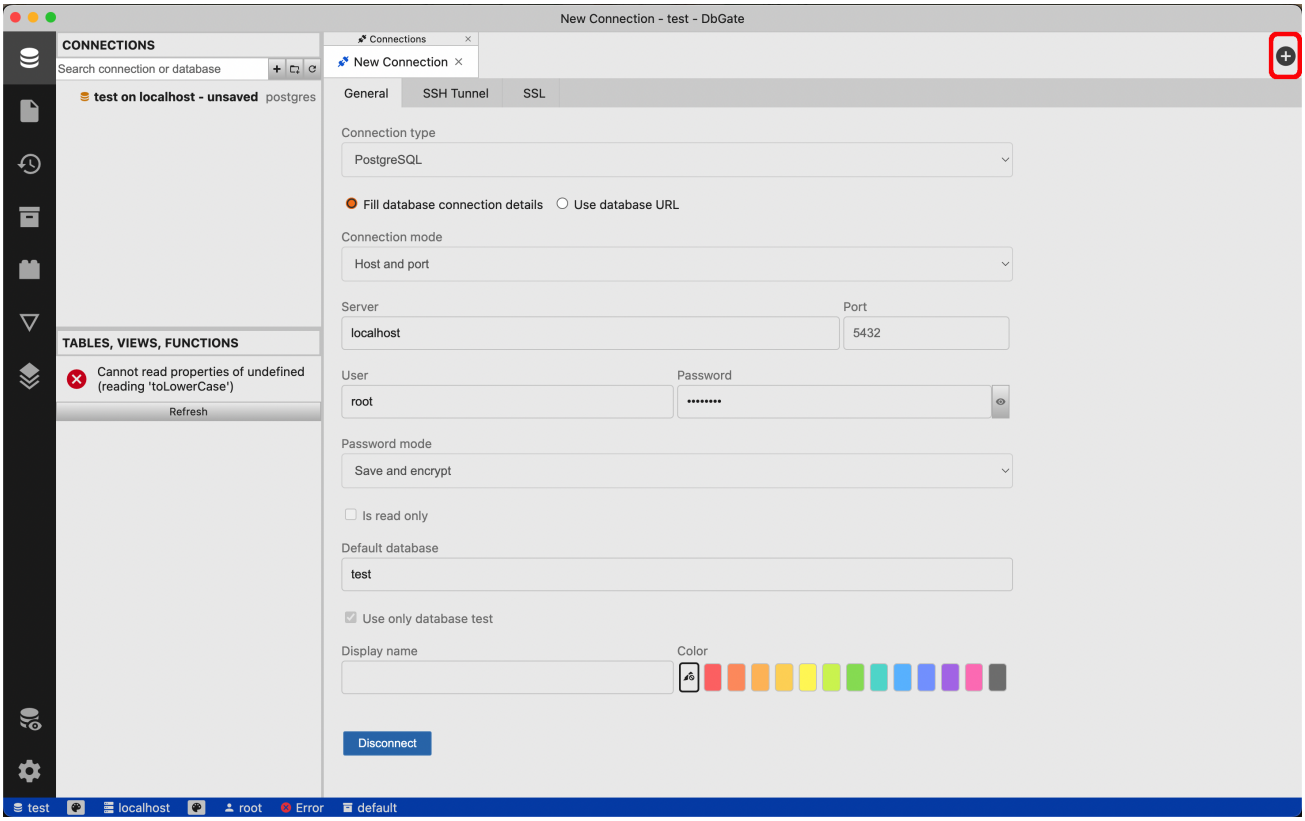
Add a new connection of type PostgreSQL:



Enter host, port, user, password, database, make sure to check "Use only database <db>", and connect:



It seems the reported error can be ignored, so now new queries can be composed by:

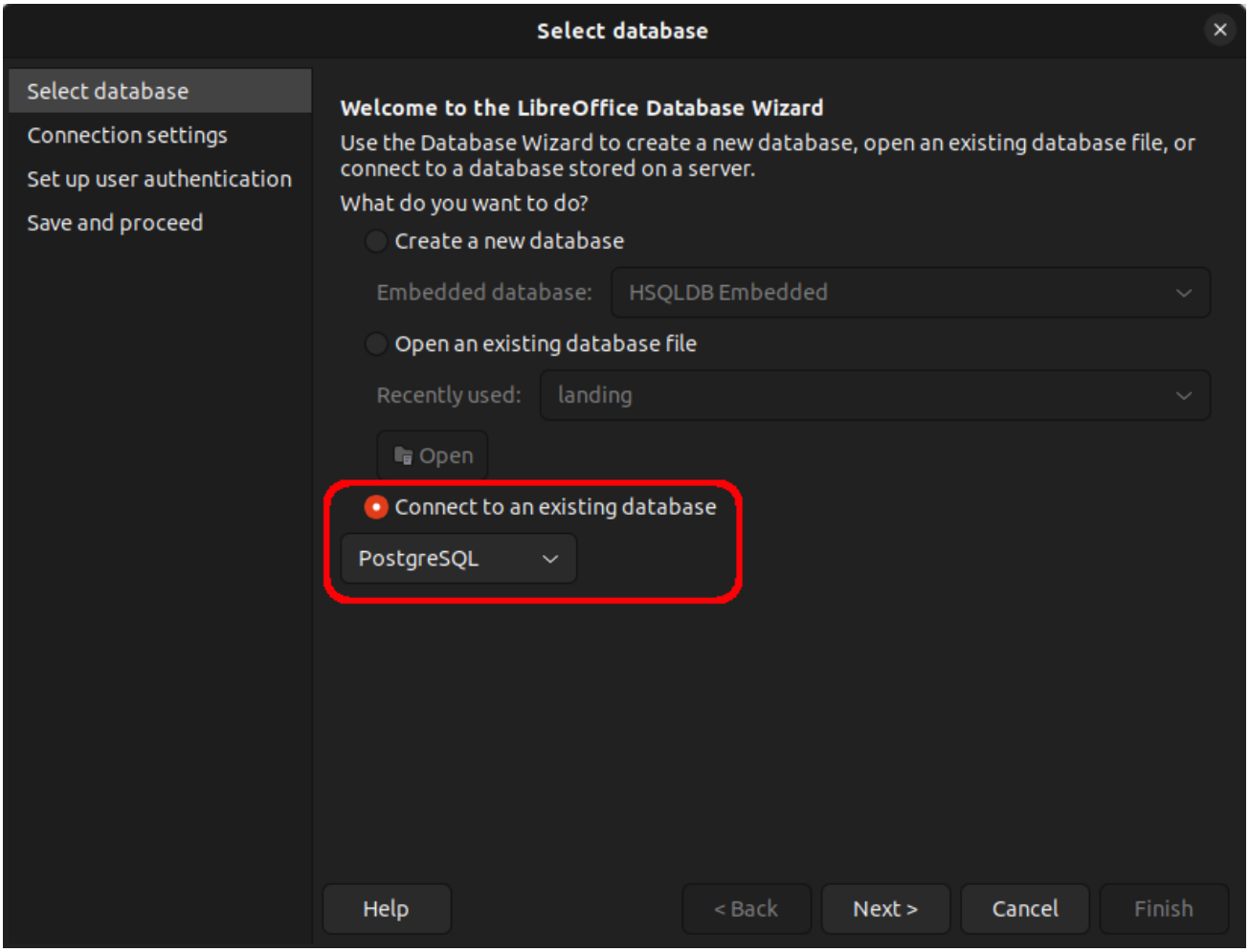


10.15.6. LibreOffice Base

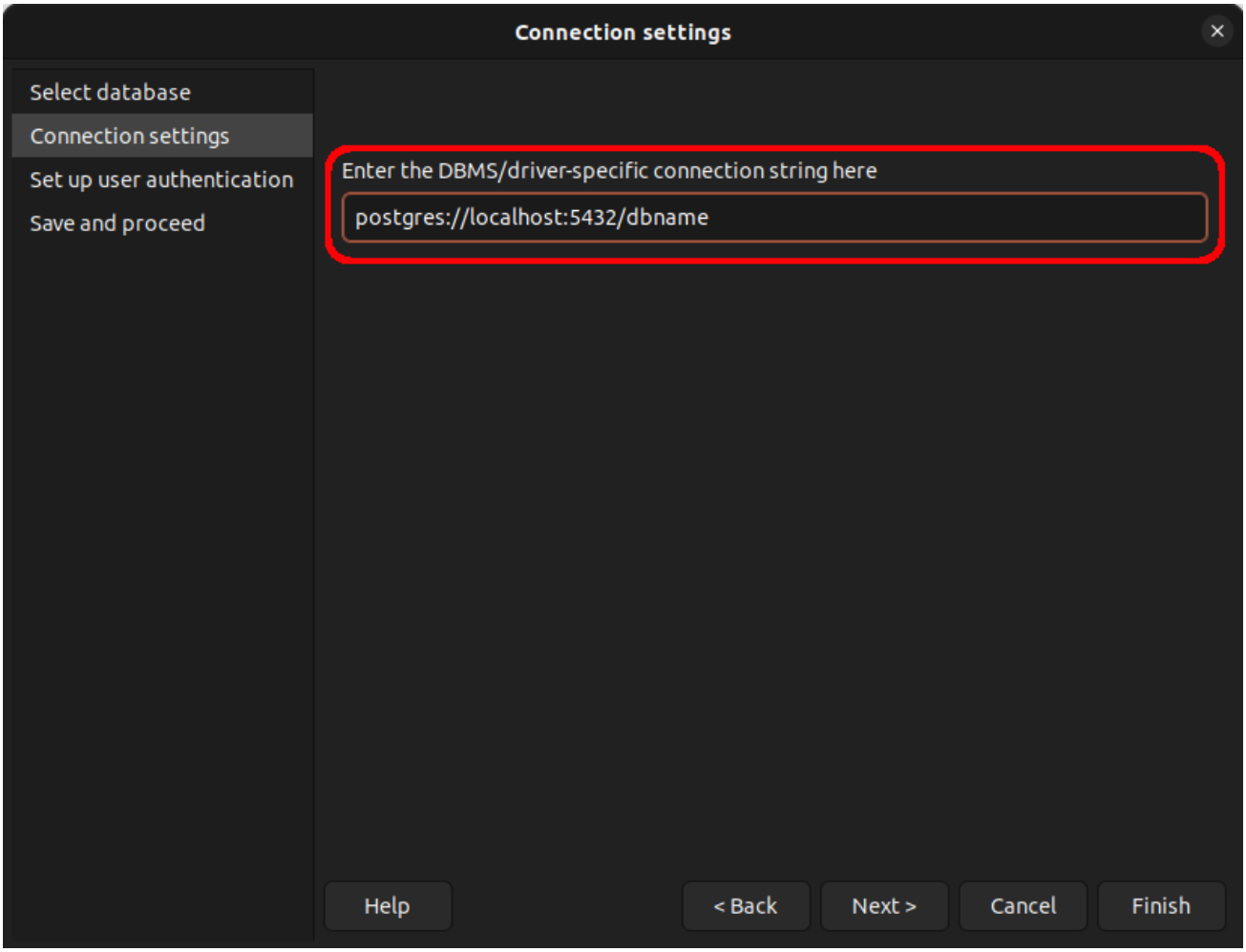
There is minimal support for ArcadeDB in LibreOffice Base via a PostgreSQL connection:

▼ *Details*

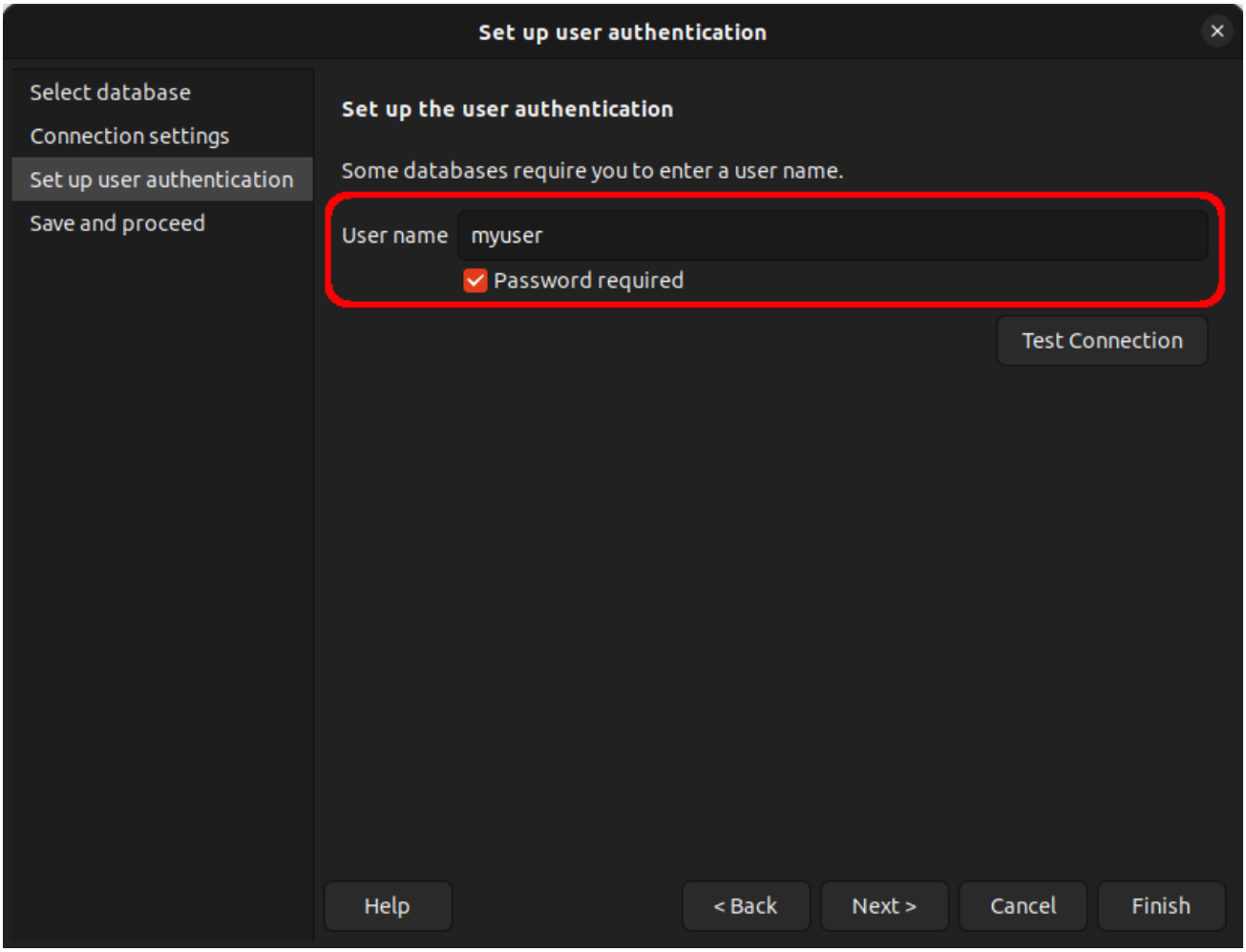
Select "Connect to existing database" and choose "PostgreSQL"



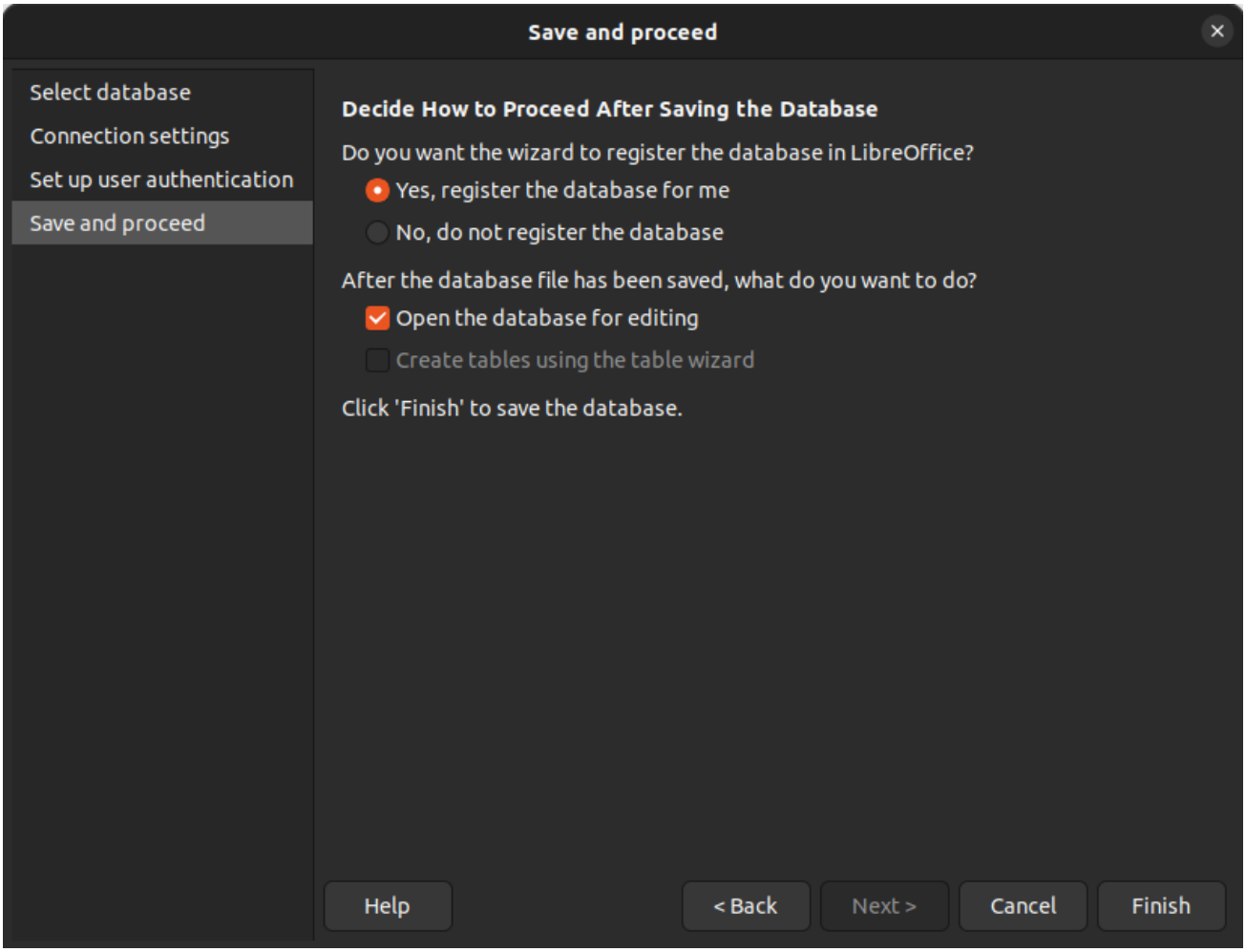
Enter the postgres protocol connection string (without username and password), for example: `postgres://localhost:5432/dbname`



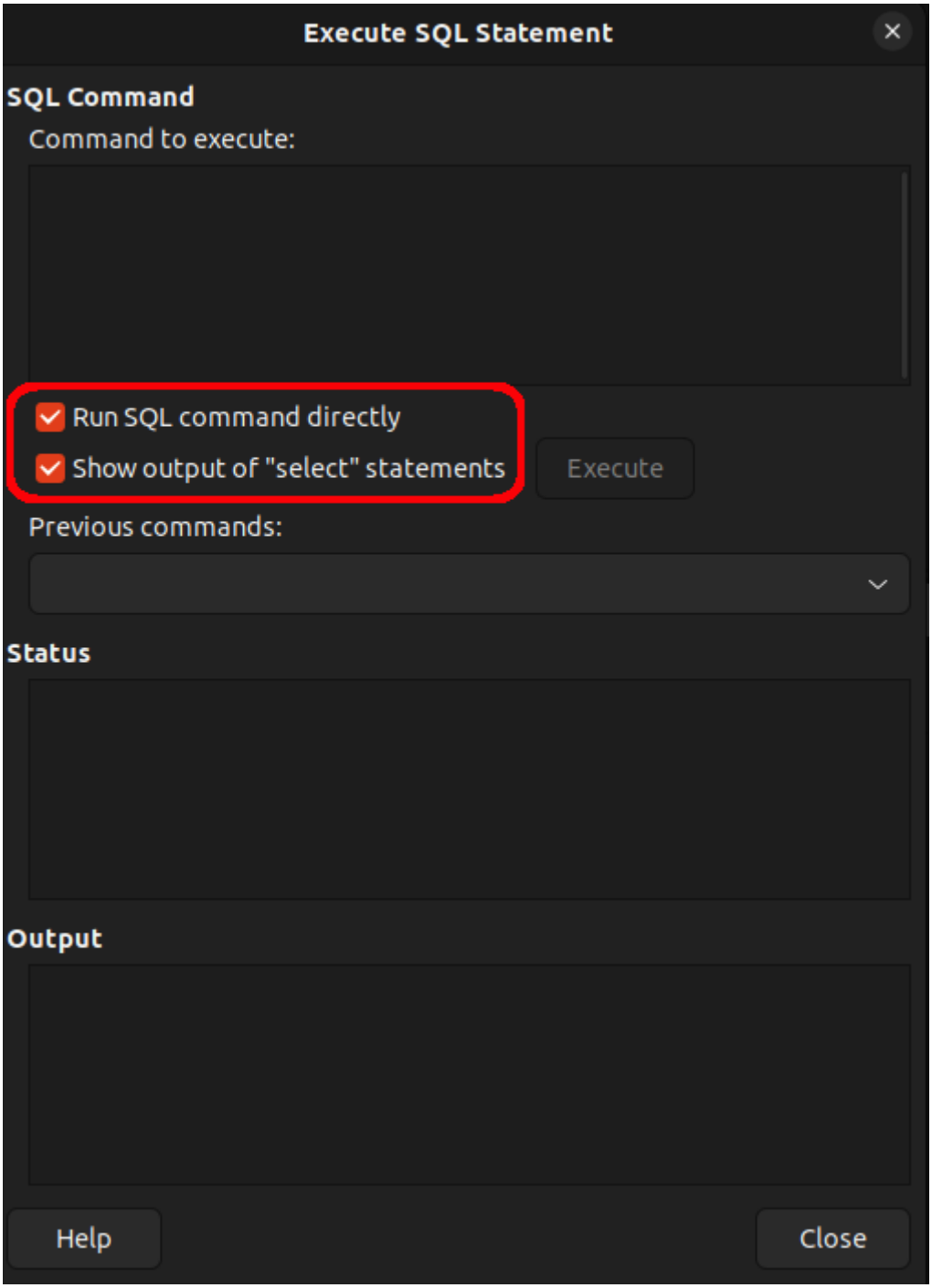
Enter the user name and check that a password is required (try with the "Test Connection" button)



Choose if you want to register the database in LibreOffice, select to open for editing, and "Finish" the wizard.



Now, in the menu under "Tools" → "SQL..." queries and commands can be send to ArcadeDB.



Make sure that "Run SQL command directly" is selected, and to view results check "Show output ..."

10.15.7. adbcBridge

adbcBridge is an open-source driver for ADBC, Apache Arrow's database API, that sits on top of an ODBC driver and returns query results as Arrow tables in Python, Go, Rust, Java and C#. It reaches ArcadeDB through the Postgres protocol plugin and the PostgreSQL ODBC driver (psqlodbc).

▼ *Details*

Install the PostgreSQL ODBC driver (the `odbc-postgresql` package on Debian and Ubuntu), then the Python packages:

```
pip install adbcbridge adbc-driver-postgresql
```

Connect with an ODBC connection string and read a query as an Arrow table:

```
import adbcbridge

conn = adbcbridge.connect(
    uri="Driver=PostgreSQL Unicode;Server=localhost;Port=5432;"
        "Database=mydatabase;Uid=root;Pwd=password;BoolsAsChar=0;"
)
with conn.cursor() as cur:
    cur.execute("SELECT name, price FROM Product WHERE price > ?", (10.0,))
    print(cur.fetch_arrow_table())
conn.close()
```

`BoolsAsChar=0` makes psqlodbc describe a `BOOLEAN` property as a boolean instead of a character.

With a psqlodbc connection string, adbcBridge hands the connection to the Apache Arrow ADBC PostgreSQL driver when that driver is installed, which is why `adbc-driver-postgresql` is in the install line. This is the route to use with ArcadeDB v26.10.1 to v26.11.1. From v26.12.1 the ODBC route through psqlodbc works as well, for example with the environment variable `ADBC_ODBC_DELEGATE=never`.

ArcadeDB has no `CREATE TABLE`, so create the type and its properties before loading data into it.

Chapter 11. BI & Analytics

11.1. BI and Analytics Integration

Most modern BI tools speak SQL over PostgreSQL — and so does ArcadeDB. The PostgreSQL wire protocol on port 5432 lets virtually any Postgres-compatible tool connect to ArcadeDB without a custom driver, an ODBC bridge, or middleware. For tools that need richer integration than ANSI SQL provides, ArcadeDB also ships native plugins (Grafana today, more on the roadmap).

11.1.1. Connection paths

There are two ways a BI tool talks to ArcadeDB. Pick the one your tool supports:

Path	What it is	When to use
Native plugin / data source	A tool-specific extension that ArcadeData publishes (or that the tool ships out of the box) which knows ArcadeDB-specific features — graph queries, multiple query languages, time-series macros, graph visualization.	Use whenever ArcadeDB ships a native plugin (today: Grafana). Always preferred over the generic Postgres path because it surfaces graph-aware visualizations, multi-language queries, and time-series-aware query builders.
PostgreSQL wire protocol	ArcadeDB exposes port 5432 as a Postgres-compatible endpoint. Any tool with a built-in PostgreSQL connector — or any Java tool that loads the standard PostgreSQL JDBC driver — points at that endpoint and runs ArcadeDB SQL exactly as it would against Postgres.	Use for tools that already speak Postgres natively: Apache Superset, Metabase, Tableau, Power BI, DBeaver, dbt, Looker, and so on.

ArcadeDB does not ship its own JDBC driver. Tools that require a JDBC URL connect through the standard PostgreSQL JDBC driver (`org.postgresql:postgresql`); the connection target is ArcadeDB's port 5432 instead of a real PostgreSQL server. Functionally it is the same wire-level path as the built-in Postgres connectors — same protocol, same auth, same query surface — and the same set of language prefixes works (see "Sending non-SQL queries" below).

11.1.2. Supported tools

Tool	Connection path	ArcadeDB-specific features
Grafana	Native plugin (recommended); Postgres data source fallback	Visual time-series query builder, ArcadeDB SQL with `$__timeFilter` macros, OpenCypher with Node-Graph visualization, Gremlin traversals, full alerting support.

Tool	Connection path	ArcadeDB-specific features
Apache Superset	PostgreSQL wire protocol (SQLAlchemy + psycopg2)	ArcadeDB SQL through Superset's SQL Lab and dashboards. Cross-database joins via Superset's virtual datasets.
Metabase	Built-in PostgreSQL JDBC connector	ArcadeDB SQL through Metabase's question editor and dashboards.
Tableau	Built-in PostgreSQL connector	ArcadeDB SQL via Tableau's PostgreSQL data source. Live or extract mode.
Power BI	Built-in PostgreSQL connector	ArcadeDB SQL via Power BI's PostgreSQL connector. Import or DirectQuery mode.

11.1.3. Setting up the Postgres endpoint

The Postgres protocol plugin is bundled with the standard distribution; you only need to enable it. See Postgres Protocol Plugin for the exact server-side configuration. Once enabled, ArcadeDB listens on port 5432 (TLS optional) and the tool sees it as a regular PostgreSQL server — same hostname/port/database/username/password fields you already know.

For Java tools that connect via JDBC, drop the standard PostgreSQL JDBC driver (`org.postgresql:postgresql`) on the classpath and use a JDBC URL of the form:

```
jdbc:postgresql://<host>:5432/<database>
```

with `<database>` being your ArcadeDB database name. There is no ArcadeDB-specific driver to install — the wire protocol is what carries the connection.

11.1.4. Sending non-SQL queries from a Postgres-only tool

The PostgreSQL connection lets you send any ArcadeDB SQL query, including graph traversals (`MATCH`, `TRAVERSE`) and vector functions. To send **Cypher**, **Gremlin**, **GraphQL**, **MongoDB QL**, or a multi-statement **SQL Script** from a tool that only speaks SQL, prefix the query with the language tag:

```
{cypher} MATCH (m:Movie)<-[a:ACTED_IN]-(p:Person) WHERE id(m) = '#1:0' RETURN *
```

The `{cypher}` prefix tells ArcadeDB to route the query to the OpenCypher engine instead of the SQL planner. The same pattern works for `{gremlin}`, `{graphql}`, `{mongo}`, and `{sqlscript}`. The result set is returned over the Postgres protocol like any other query, so the BI tool can chart it without knowing it came from a non-SQL engine.

11.1.5. Suggesting a missing tool

If your favourite BI or analytics tool is not in the list above and it supports PostgreSQL, it almost certainly works against ArcadeDB out of the box — try it and let us know how it goes on GitHub Discussions. If the tool needs a dedicated plugin or has rough edges over the Postgres protocol, open

an issue at https://github.com/ArcadeData/arcadedb/issues — community-driven plugin work is very welcome.

11.2. Apache Superset

Apache Superset is the most popular open-source BI and data exploration platform. It connects to ArcadeDB via the PostgreSQL wire protocol using SQLAlchemy and psycopg2.

11.2.1. Setup

1. In Superset, go to **Settings** > **Database Connections** > + **Database**.
2. Select **PostgreSQL**.
3. Enter the connection details:

Host	`localhost` (or your ArcadeDB server host)
Port	`5432`
Database	Your ArcadeDB database name
Username	Your ArcadeDB username
Password	Your ArcadeDB password

   Or use the SQLAlchemy URI:

```
postgresql+psycopg2://root:arcadedb@localhost:5432/mydb
```

4. In the **Advanced** tab, add the following to **Extra** to disable SSL:

```
{
  "connect_args": {
    "sslmode": "disable"
  }
}
```

5. Click **Test Connection**, then **Connect**.

11.2.2. Usage

Use **SQL Lab** to run queries:

```
SELECT name, age, city FROM Person LIMIT 100
```

Graph traversals work via ArcadeDB SQL:

```
SELECT name, out('FriendOf').size() AS friends
```

```
FROM Person
WHERE out('FriendOf').size() > 3
```

Cypher queries work with the language prefix:

```
{cypher}MATCH (p:Person)-[:FRIEND_OF]->(f:Person)
RETURN p.name AS person, f.name AS friend LIMIT 50
```

Charts and dashboards can be created from SQL Lab results or by selecting an ArcadeDB type as a table in the chart builder.

11.2.3. Known Limitations

- Schema introspection is partial - Superset may not discover all types or columns automatically
- Language prefixes (`{cypher}`, `{gremlin}`) work in SQL Lab but not in the visual chart builder
- Some ArcadeDB SQL extensions (e.g., `out()`, `in()`) may confuse Superset's query parser in the visual builder
- SSL is not supported - the `sslmode=disable` setting is required
- No graph visualization (Superset does not have a network chart type)

11.3. Grafana

Grafana is the leading open-source platform for monitoring and observability. ArcadeDB provides two integration paths: a dedicated Grafana plugin (recommended) and a fallback via the built-in PostgreSQL data source.

11.3.1. ArcadeDB Grafana Plugin (Recommended)

The ArcadeDB Grafana plugin provides a native integration with dedicated query editors, time series visual builder, and graph visualization via Grafana's Node Graph panel.

Features:

- **Time Series mode** - visual query builder with auto-discovered types, fields, tags, and aggregation functions (SUM, AVG, MIN, MAX, COUNT)
- **SQL mode** - ArcadeDB SQL with macro expansion (`$__timeFrom`, `$__timeTo`, `$__timeFilter(col)`, `$__interval`)
- **Cypher mode** - OpenCypher queries with optional Node Graph visualization
- **Gremlin mode** - Apache Gremlin traversals with optional Node Graph visualization
- **Alerting** - full Grafana alerting support via the Go backend
- **Template variables** - dashboard variables populated from ArcadeDB queries

Installation:

```
grafana-cli plugins install arcadedb-arcadedb-datasource
```

Or build from source:

```
git clone https://github.com/ArcadeData/arcadedb-grafana-datasource.git
cd arcadedb-grafana-datasource
mage -v build:linux
```

For self-hosted Grafana with unsigned plugins, add to `grafana.ini`:

```
[plugins]
allow_loading_unsigned_plugins = arcadedb-arcadedb-datasource
```

Configuration:

1. Go to **Connections** > **Data Sources** > **Add data source** and search for **ArcadeDB**.
2. Set the **URL** to your ArcadeDB HTTP API (e.g., `http://localhost:2480`).
3. Enable **Basic Auth** and enter your ArcadeDB credentials.
4. Set the **Database** name in the ArcadeDB section.
5. Click **Save & Test**.

Time Series query example:

1. Select **Time Series** mode.
2. Choose a type (e.g., `cpu_metrics`), fields (e.g., `usage`), and optional tag filters.
3. Add an aggregation: AVG on `usage` with a bucket interval or leave it to auto-calculate.

SQL query example:

```
SELECT ts, temperature FROM weather
WHERE $__timeFilter(ts)
ORDER BY ts
```

Cypher query with Node Graph:

Enable the **Node Graph** toggle, then:

```
MATCH (p:Person)-[r:FRIEND_OF]->(f:Person)
RETURN p, r, f LIMIT 100
```

The result renders as an interactive graph in Grafana's Node Graph panel.

11.3.2. PostgreSQL Data Source (Fallback)

If you cannot install the ArcadeDB plugin, use Grafana's built-in PostgreSQL data source with ArcadeDB's PostgreSQL wire protocol.

1. Go to **Connections** > **Data Sources** > **Add data source** and select **PostgreSQL**.
2. Set **Host** to `localhost:5432`, enter the **Database** name and credentials.
3. Set **TLS/SSL Mode** to `disable`.
4. Click **Save & Test**.

In panels, switch to **Code** mode and write SQL:

```
SELECT name, age FROM Person LIMIT 100
```

Use the `{cypher}` prefix for Cypher queries:

```
{cypher}MATCH (p:Person) RETURN p.name, p.age LIMIT 100
```

The PostgreSQL fallback does not support the time series visual builder, Node Graph visualization, ArcadeDB-specific macros, or alerting optimizations. Use the dedicated plugin for the full experience.

11.3.3. Grafana Time Series Endpoints

ArcadeDB also provides native Grafana DataFrame-compatible HTTP endpoints for time series data. See Time Series for full documentation of the `/grafana/health`, `/grafana/metadata`, and `/grafana/query` endpoints, as well as the PromQL-compatible endpoints.

11.4. Metabase

Metabase is a popular open-source BI tool known for its ease of use. It connects to ArcadeDB via the built-in PostgreSQL JDBC driver.

11.4.1. Setup

1. In Metabase, go to **Settings** > **Admin** > **Databases** > **Add database**.
2. Select **PostgreSQL**.
3. Enter the connection details:

Display name	`ArcadeDB`
Host	`localhost`
Port	`5432`
Database name	Your ArcadeDB database name

Username	Your ArcadeDB username
Password	Your ArcadeDB password

4. Expand **Additional JDBC connection string options** and add:

```
sslmode=disable
```

5. Click **Save**. Metabase will sync the schema.

11.4.2. Usage

Click + **New** > **SQL query**, select your ArcadeDB database, and write queries:

```
SELECT name, age FROM Person ORDER BY name LIMIT 50
```

Graph traversal via ArcadeDB SQL:

```
SELECT name, out('FriendOf').size() AS friendCount
FROM Person
WHERE out('FriendOf').size() > 2
```

Cypher via language prefix in native query mode:

```
{cypher}MATCH (p:Person)-[:FRIEND_OF]->(f:Person)
RETURN p.name AS person, f.name AS friend LIMIT 50
```

Metabase's "Simple question" and "Custom question" builders show ArcadeDB types as tables with their properties as columns.

11.4.3. Known Limitations

- Schema sync may not discover all types or properties
- The visual question builders work for basic queries but may have gaps for complex filter types
- Language prefixes only work in native query mode, not the visual builders
- Auto-generated queries from Metabase may occasionally use PostgreSQL-specific syntax that ArcadeDB does not support
- SSL is not supported

11.5. Power BI

Microsoft Power BI connects to ArcadeDB via its PostgreSQL connector.

11.5.1. Setup

1. In Power BI Desktop, click **Get Data** > **More...** > **Database** > **PostgreSQL database**.
2. Click **Connect**.
3. Enter the connection details:

Server	`localhost:5432` (include the port)
Database	Your ArcadeDB database name
Data Connectivity mode	Import (recommended)

4. When prompted for credentials, select **Database** and enter your ArcadeDB username and password.
5. If prompted about encryption, choose to connect without encryption.

11.5.2. Usage

For custom queries, use the **Advanced options** in the connection dialog:

1. In the **SQL statement** field, enter your query:

```
SELECT name, age, city FROM Person LIMIT 1000
```

2. Click **OK**.

Graph traversals and Cypher via language prefix:

```
SELECT name, out('FriendOf').size() AS friendCount FROM Person WHERE age > 25
```

```
{cypher}MATCH (p:Person)-[:FRIEND_OF]->(f:Person)
RETURN p.name AS person, f.name AS friend, p.city AS city
```

After loading data, build visualizations by dragging fields from the Fields panel. For graph/network visualizations, install a custom visual from AppSource (e.g., "Network Graph by Powerviz").

Use **Import** mode instead of **DirectQuery**. DirectQuery works for basic queries but may have issues with complex types or ArcadeDB-specific SQL extensions.

11.5.3. Known Limitations

- DirectQuery mode may not work reliably for all query types - Import mode is recommended
- The Navigator dialog may show incomplete or no tables - use native SQL queries instead
- Language prefixes work in the SQL statement field but not through the visual query builder

- No built-in graph/network visualization - requires a marketplace custom visual
- SSL is not supported - connections must be unencrypted
- Refreshing data in Power BI Service requires a gateway configured for PostgreSQL connections

11.6. Tableau

Tableau is a leading commercial BI platform. It connects to ArcadeDB via the built-in PostgreSQL connector.

11.6.1. Setup

1. In Tableau Desktop, go to **Connect** > **To a Server** > **PostgreSQL**.
2. Enter the connection details:

Server	`localhost`
Port	`5432`
Database	Your ArcadeDB database name
Username	Your ArcadeDB username
Password	Your ArcadeDB password
Require SSL	Unchecked

3. Click **Sign In**.

11.6.2. Usage

For the best experience with ArcadeDB, use **Custom SQL** rather than dragging tables:

1. In the Data Source tab, click **New Custom SQL**.
2. Enter your query:

```
SELECT name, age, city FROM Person
```

Or with graph traversal:

```
SELECT name, out('FriendOf').size() AS friendCount, city
FROM Person
```

3. Click **OK**. Tableau executes the query and shows the result schema.
4. Drag fields to rows/columns to build visualizations.

Cypher queries work via the language prefix in Custom SQL:

```
{cypher}MATCH (p:Person)-[:FRIEND_OF]->(f:Person)
RETURN p.name AS person, p.city AS city, COUNT(f) AS friendCount
```

Use **Import** mode (extract) rather than **Live Connection** for large datasets or when ArcadeDB-specific SQL extensions cause issues with Tableau's query rewriting.

11.6.3. Known Limitations

- Schema discovery may show incomplete table/column lists - prefer Custom SQL
- Tableau may rewrite queries using PostgreSQL-specific syntax that ArcadeDB does not fully support
- Relationships between tables must be defined manually via Custom SQL
- No built-in graph/network visualization type
- SSL is not supported - ensure "Require SSL" is unchecked

Chapter 12. Advanced

12.1. Performance Tuning

This guide covers common performance optimizations for ArcadeDB in production.

12.1.1. Memory Configuration

ArcadeDB uses both JVM heap and off-heap memory. Key settings:

```
# JVM heap (set via JAVA_OPTS or bin/server.sh)
JAVA_OPTS="-Xms2g -Xmx4g"

# Page cache size in MB (default: 25% of the maximum heap, so 4096 MB under -Xmx16g)
arcadedb.maxPageRAM=8192
```

The page cache lives inside the JVM heap. When the database is larger than the cache, pages are evicted and read again from disk, so a dedicated embedded or server process with a database bigger than a quarter of its heap benefits from a larger share: 40-50% of `Xmx` is a good starting point, leaving the rest for transactions, query working sets and indexes. A value above 80% of the heap is reduced automatically. Grow `Xmx` first if the heap is the limit.

JVM settings inside a fixed memory budget

When a container, a pod or a VM caps the memory of the process, the heap is only a part of what counts against the cap: the JVM also needs metaspace, thread stacks, direct buffers, the code cache and the collector's own structures, and the OS needs file cache for the database files. Which share of the cap goes to the heap depends on who owns the process:

Deployment	Intended setting	Why
Server in the Docker image or in Kubernetes	The image's defaults: `-XX:MaxRAMPercentage=75` and generational ZGC, no `-Xms`. Give the container the memory you want ArcadeDB to have and the heap follows.	The cap covers heap and non-heap memory, so 75% leaves a quarter of it for the rest. ZGC keeps pauses short on a large heap, and an unpinned heap grows with the working set instead of being filled before the first collection.
Embedded, in your own JVM	There is no default: ArcadeDB does not start the JVM. Start with a heap of 50% of the budget when the application or the OS file cache needs room, up to 75% for a process that only runs ArcadeDB, and leave the collector at the JVM default (G1).	The application's own memory, and the OS file cache that serves the database files, share the budget with the heap.

Pinning `-Xms` to `-Xmx` fills the heap with garbage before the first young collection on

> G1, so the process memory climbs towards the heap size whatever the live working set is. A "peak memory" measured under a pinned heap reports the setting, not what ArcadeDB needs. To compare memory across engines, either leave `-Xms` small and fix only `-Xmx`, or report the live heap (the `Heap used` gauge in Studio, or `jcmd <pid> GC.heap_info` after a full collection) beside the peak.

The collector is chosen by `ARCADEDB_OPTS_GC`, not by `JAVA_OPTS` or `JAVA_TOOL_OPTIONS`: the image sets `ARCADEDB_OPTS_GC="-XX:+UseZGC"`, and a second collector flag on the same command line makes the JVM refuse to start ("Multiple garbage collectors selected"). To run G1 in the image, replace the variable (`ARCADEDB_OPTS_GC="-XX:+UseG1GC"`), or set it to an empty value to take the JVM default. The image runs on Java 25, where ZGC is always generational, so `-XX:+ZGenerational` is obsolete: adding it only prints a warning at every start.

12.1.2. Index Selection

ArcadeDB supports multiple index types. Choose based on your access pattern:

Index Type	Best For	Trade-off
LSM-Tree (default)	Range queries, `ORDER BY` on the key, prefix scans	Slower point lookups than Hash
Hash	Keys only read, updated and deleted by equality (`WHERE key = value`), such as ids and tokens	No range queries and no ordering
LSM_VECTOR	Vector similarity search	Specialized for ANN queries
FULL_TEXT	Text search (`SEARCH_INDEX()`)	Specialized for keyword search
GEOSPATIAL	Spatial queries (`geo.*()`)	Specialized for geometry

Create appropriate indexes for your query patterns:

```
-- LSM-Tree for range queries (default)
CREATE INDEX ON User (email) UNIQUE

-- Hash for exact lookups
CREATE INDEX ON Session (token) UNIQUE_HASH

-- Composite index for multi-column queries
CREATE INDEX ON Order (customerId, orderDate) NOTUNIQUE
```

12.1.3. Query Optimization

Use `EXPLAIN` and `PROFILE` to understand query execution:

```
-- Show the query plan without executing
EXPLAIN SELECT FROM User WHERE email = 'alice@example.com'
```

```
-- Execute and show actual timings
PROFILE SELECT FROM User WHERE email = 'alice@example.com'
```

`PROFILE` really executes the statement, so it counts as a read only when the statement it wraps is a read. `PROFILE INSERT …` is refused on the read-only query endpoint (`GET`/`POST /api/v1/query`) and needs the command endpoint and write permission, like the plain `INSERT`. `EXPLAIN` never executes, so it is always allowed there.

Common optimizations:

- **Add indexes** for `WHERE` clause predicates — check `EXPLAIN` output for full scans
- **Limit result sets** — Use `LIMIT` to avoid fetching unnecessary records
- **Project only needed fields** — `SELECT name, email FROM User` is faster than `SELECT * FROM User`
- **Use parameterized queries** — Avoid SQL parsing overhead for repeated queries

Parameters are worth more than they look, especially in Cypher. A parsed query is cached by its exact text, so writing the value inside the query produces a new text on every call and the cache is never reused:

```
-- a different query text every time: re-parsed on every call, and
fills the cache with single-use entries
MATCH (p:Person) WHERE p.id = 1234 RETURN p.name, p.age

-- one text, parsed once, reused from then on
MATCH (p:Person) WHERE p.id = $id RETURN p.name, p.age
```

For a small indexed lookup the parse dominates the cost of the whole query, so this single change is often the difference between the two forms being several times apart in latency.

12.1.4. Bucket Configuration

ArcadeDB distributes records across buckets for parallel access. By default a new type gets one bucket (the `arcadedb.typeDefaultBuckets` setting is `1`); set a higher value, or pass the number of buckets when creating the type, to get more.

For write-heavy types, choose the number of buckets when you create the type, with `BUCKETS`. For asynchronous loads, use a count equal to, or a multiple of, the async parallel level (`arcadedb.asyncWorkerThreads`, cores minus 1 by default):

```
CREATE DOCUMENT TYPE HighWriteType BUCKETS 8
```

You can also add a bucket to an existing type later, but only records created afterwards go into it, and existing records are not moved. `ALTER TYPE` adds and removes buckets by name (`+` adds, `-`

removes), and a bucket that does not exist is created:

```
ALTER TYPE HighWriteType BUCKET +HighWriteType_extra1
ALTER TYPE HighWriteType BUCKET +HighWriteType_extra2
```

For read-heavy types with few writes, fewer buckets reduce overhead; remove a bucket by name:

```
ALTER TYPE LookupTable BUCKET -LookupTable_extra1
```

12.1.5. Page Size Tuning

The default bucket page size (64KB) works well for most workloads. It applies to types created after the change, existing types keep theirs. Adjust for specific patterns:

```
# Larger pages for sequential scan workloads (analytics)
arcadedb.bucketDefaultPageSize=131072

# Smaller pages for random access workloads
arcadedb.bucketDefaultPageSize=32768
```

LSM-Tree indexes have their own page size (default 256KB), set with `arcadedb.indexDefaultPageSize` (new indexes only). A transaction that changes a page works on a private copy of the whole page, so write-heavy workloads of small transactions pay less with smaller index pages (for example `arcadedb.indexDefaultPageSize=32768`), while lookups are slightly faster with larger ones.

12.1.6. Connection Pool Settings

For remote applications, size the HTTP worker threads (there is no setting for a maximum number of connections) and the async queue based on concurrency:

```
# Threads executing HTTP requests
arcadedb.server.httpWorkerThreads=256

# Async queue size
arcadedb.asyncOperationsQueueSize=1024
```

12.1.7. Concurrent Requests (Query Admission Gate)

(since v26.10.2) Every request of a remote client passes through a queue before it runs. When the server is busy, requests wait and start in arrival order (FIFO). Without the queue they would all start at once, and under a burst of heavy queries the last ones would be refused by the query heap budget or exhaust the heap.

How it works

- A request starts at once when fewer than `arcadedb.queryMaxConcurrent` requests are running and the running queries hold less than `arcadedb.queryAdmissionHeapWatermark` percent of the query heap budget (`arcadedb.queryMaxHeapRAM`). Otherwise it waits in the queue.
- The wait happens before anything of the request runs. A query started from inside a running request, such as a function or a script, does not queue again.
- A request that waits longer than `arcadedb.queryQueueTimeout`, or finds `arcadedb.queryQueueMaxSize` requests already waiting, is refused with a retryable error. Nothing of it ran, so the client can send it again as it is.
- When no admitted request is running, the next one starts whatever the heap, so memory held by work outside the queue cannot block it.

What goes through the queue

Protocol	Gated	Refused with
HTTP	Every database request: query, command, batch load, vector and full-text search, time series, PromQL and Grafana. Begin, rollback and the health probe are exempt	`503`
Postgres	Every statement	SQLSTATE `40001`
BOLT	Every query. A result stream keeps its slot until it is read to the end or discarded	transient error
Redis	Every command but `PING`, `SELECT` and, since v26.11.1, the commands on keys held in memory (`GET`, `SET`, `GETDEL`, `EXISTS`, `INCR`, `INCRBY`, `INCRBYFLOAT`, `DECR`, `DECRBY`)	`TRYAGAIN`
gRPC	Queries, commands, record operations, searches and time series. Begin and rollback, the admin service and the streaming loads are exempt	`ABORTED`
Gremlin Server	Scripts and traversals (sessions are not gated)	error response
MCP	The tools that read or write data	tool error
MongoDB	Every command, but it never waits: one that cannot start at once is refused	`ExceededTimeLimit` (262)

The embedded Java API does not go through the queue.

Sizing

- `arcadedb.queryMaxConcurrent` defaults to twice the number of cores, at least 4. Each running request holds one slot for as long as it runs, so long analytical queries next to short ones need

room for both: a handful of long queries holding every slot makes the short ones wait and, past the timeout, be refused.

- Lower it to protect memory when most queries are heavy. Raise it when many requests spend their time waiting on disk or on clients rather than on CPU.
- Keep `arcadedb.queryQueueMaxSize` (default 256) below `arcadedb.server.httpWorkerThreads` (default 500): each waiting HTTP request parks a worker thread.
- Keep `arcadedb.queryQueueTimeout` (default 30 seconds) below your clients' own timeouts, so they receive the retryable refusal rather than a timeout of their own.
- A transaction that took explicit locks (`LOCK TYPE`, `LOCK BUCKET`) keeps them while one of its requests waits in the queue.

```
# At most 16 requests at once, the next ones wait up to 10 seconds
arcadedb.queryMaxConcurrent=16
arcadedb.queryQueueTimeout=10000

# Disable the queue: every request starts at once (the behavior before v26.10.2)
# arcadedb.queryMaxConcurrent=0
```

Monitoring

Studio shows the queue in the Server tab, under Executor Pools, as "Query Admission Gate": the running requests against the limit, the waiting ones, and the refused ones. The same numbers are exported as metrics (`pool=query_admission`) and reported by the profiler (`queryAdmission*`).

- Refusals that keep growing mean the server is undersized for the load, or the limit is too tight for it.
- `queryAdmissionHeapDeferrals` growing while requests wait means the heap budget, not the number of slots, holds them back: give the server more heap, raise `arcadedb.queryMaxHeapRAM`, or reduce the size of the queries.

12.1.8. Write Optimization

- **Batch transactions** — Group related writes in a single transaction to amortize commit overhead
- **Async operations** — Use the async API for non-blocking writes (Java Async API)
- **Bulk loading** — Use `BulkInsert` via gRPC or the Java embedded API for initial data loads
- **WAL flush tuning** — The `arcadedb.txWalFlush` setting controls whether each commit triggers an fsync. In `production` mode it defaults to `1` (fsync without metadata). For bulk imports or when your storage has battery-backed write cache, you can set it to `0` for maximum throughput. See concepts/transactions.pdf

12.1.9. Read Optimization

- **Page cache** — Increase `arcadedb.maxPageRAM` if cache hit rate is low

- **Index coverage** — Ensure queries hit indexes (check with `EXPLAIN`)
- **Projections** — Select only the fields you need
- **Materialized views** — Pre-compute expensive aggregations

12.1.10. High Availability Resync

- **Snapshot compression** — A follower that falls too far behind is resynced with a full snapshot of the database, compressed by the leader on a single thread per transfer. At the Deflater default (level 6) this can cap a resync at a few MB/s on a fast private network. Since v26.10.2 `arcadedb.ha.snapshotCompressionLevel` defaults to `1` (fastest DEFLATE), which keeps most of the size reduction of database pages at a fraction of the CPU. Use `0` to ship the files stored on very fast links, or `-1` to restore the previous level on a WAN or metered link where bytes cost more than CPU
- **Concurrent followers** — Two followers of the same database are served at the same time, up to `arcadedb.ha.snapshotMaxConcurrent` (default 2). Raise it only if more than two followers can resync together and the leader has I/O and CPU to spare

12.1.11. Further Reading

- Server Settings — Complete list of all configuration parameters
- Monitoring — Track performance metrics with Prometheus
- LSM-Tree Architecture — Understanding the storage engine

12.2. Settings

ArcadeDB allows changing settings at JVM (server or embedded) and per database level.

Server/Embedded (JVM) Level	Database Level
Those settings are valid for all the databases open in the same Server or JVM when run embedded. If defined, they override the default value (look at the table below to see the default values). They are used only if a database does not override them. Such settings are not saved, so you need to set them everytime.	Database level settings are stored in the database and override the Server/Embedded (JVM) settings if present. You can change these settings via SQL or API when run embedded.

12.2.1. JVM startup (server/embedded only)

All the settings modified at JVM startup are not persistent and need to be set everytime you're running ArcadeDB server or your embedded application. If you're updating a setting at JVM level, prefix the setting name with `arcadedb.` by using this syntax:

```
$ java ... -Darcadedb.<name>=<value> ...
```

Where `<name>` is the name of the setting and `<value>` the value you want to override. Example to

change the server mode from development (default) to production:

```
$ java ... -Darcadedb.server.mode=production ...
```

Example to increase the default page size for buckets to 1 MB:

```
$ java ... -Darcadedb.bucketDefaultPageSize=1048576 ...
```

Alternatively, these settings can be set via the environment variable `ARCADEDB_SETTINGS`, which the launcher scripts (`bin/server.sh`, `bin/console.sh` and friends) put on the JVM command line:

```
ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata" bin/server.sh
```

`JAVA_OPTS` is passed through as well, but it is meant for JVM flags: keeping settings in `ARCADEDB_SETTINGS` means overriding one variable never discards the other.

Boolean values

A setting of type `Boolean` accepts only `true` or `false` (in any casing, surrounding spaces are ignored). Anything else — `yes`, `1`, `on`, or a misspelling such as `ture` — is refused: the setting keeps its default and a warning naming the setting and the rejected value is logged at startup. This holds however the value is provided: a JVM system property, an environment variable, `ARCADEDB_SETTINGS`, the server configuration file, or an admin command. *(The server configuration file is checked this way.)*

The point is that a typo can never quietly mean `false`. For a setting that guards something — for example `ha.tls.mutualAuth` or `ha.peerAllowlist.enabled`, both of which default to `true` — silently reading an unrecognised value as `false` would have turned the protection off while it looked enabled. If a setting is not taking effect, check the server log for that warning.

Settings with a fixed list of values

Some settings accept only a given set of words — `server.mode` is `development`, `test` or `production`, for instance. A value outside the list is refused wherever it is written, including the server configuration file, and the setting keeps its previous value with a warning naming what was allowed. Casing does not matter: `Production` is accepted and understood the same as `production`.

Setting names

A setting name is case-insensitive, and the `arcadedb.` prefix is optional in the server configuration file: `server.mode`, `Server.Mode` and `arcadedb.server.mode` all name the same setting.

A name in the server configuration file that matches no setting — a typo, or a leftover from an older version — is ignored and reported with a warning naming it, so it no longer disappears in silence.

Server-level settings

A setting documented as server level can be written in the server configuration file, with `SET SERVER`

`SETTING`, or as a `-D` JVM property, and all three take effect.

A setting that accepts only a fixed set of values — `server.mode`, `ha.quorum`, `ha.serverRole` and the others whose description lists their values — refuses anything outside that set, whichever channel writes it: the configuration file, a `-D` property, `SET SERVER SETTING`, `ALTER DATABASE … SETTING`, the gRPC settings call or the `set_server_setting` MCP tool. The setting keeps the value it had and the refusal names the values it accepts.

Before this, the administrative channels stored an unaccepted value without complaint. A typo such as `SET SERVER SETTING arcadedb.server.mode prodction` was answered with `200` and stored as written; every reader compares the value against `production`, found it different, and kept the **development** behaviour — so a server you believed was in production mode was not, and nothing said so.

If you have scripts that write one of these settings, check the values they send before upgrading: a value that was silently ignored before now returns an error.

12.2.2. SQL (Database Level)

All the changes executed via SQL Alter Database command are relative to the current database only and are persistent. Here is an example to increase the default page size for buckets to 1 MB:

```
ALTER DATABASE `arcadedb.bucketDefaultPageSize` 1048576
```

The current settings can be listed from SQL using:

```
SELECT expand(settings) FROM schema:database
```

12.2.3. Programmatically (Server/Embedded and Database levels)

You can access to the database configuration with `database.getConfiguration()` to read and write per database settings. Example to increase the default page size for all the buckets to 1 MB on the current database:

```
database.getConfiguration().setValue(GlobalConfiguration.BUCKET_DEFAULT_PAGE_SIZE,
1048576);
```

To change a setting at Server/Embedded (JVM) level, set the value in the `GlobalConfiguration` enum. Example to increase the default page size for buckets to 1 MB for all the databases open in the current JVM (server/embedded):

```
GlobalConfiguration.BUCKET_DEFAULT_PAGE_SIZE.setValue(1048576);
```

12.2.4. Available settings by scope (in alphabetic order):

The tables that follow contains all the available settings in ArcadeDB separated by scope:

- `JVM`, as the settings applied at the JVM level, so to both clients and servers
- `SERVER`, as the settings that only apply to the ArcadeDB Server. If you're embedding a server in your Java application you can use these settings
- `DATABASE` are all the settings that can be saved in the database configuration and are restored once the database is open

JVM

Name	Description	Type	Default Value
`asyncCommandPoolThreads`	Maximum number of threads in the JVM-wide pool that runs DDL sent through the asynchronous API, for example a `CREATE INDEX` or `REBUILD INDEX` sent to `POST /api/v1/command` with `awaitResponse=false`. Only DDL goes here, because it has to quiesce the per-database async workers; everything else runs on `asyncWorkerThreads`. `0` means the number of available cores (at least 2)	Integer	0
`asyncCommandQueueSize`	Size of the queue in front of the asynchronous DDL pool (see `asyncCommandPoolThreads`). When it is full the statement runs on the submitting thread instead of being refused, so a client that asked not to wait may wait. `0` means 1024	Integer	1024
`dumpConfigAtStartup`	Dumps the configuration at startup	Boolean	false
`dumpMetricsEvery`	Dumps the metrics at startup, shutdown and every configurable amount of time (in seconds)	Long	0

Name	Description	Type	Default Value
`fileSync`	**For testing only.** Set it to `false` to skip every fsync ArcadeDB issues: data and WAL files, the directory entries of created and renamed files, schema and configuration saves, security files, backup restores and HA snapshots. Data then survives a crash of the process, because the operating system still writes it, but not a power failure or a crash of the operating system. Only the exact text `false` turns it off, so a typo keeps the fsync. Its default follows the JVM system property `arcadedb.fileSync`, so a test run that sets it keeps it across `GlobalConfiguration.resetAll()`. ArcadeDB's own unit tests run with it off, which halves the time of the engine test suite, and the tests that verify fsync turn it back on. Never turn it off in production: to trade durability for commit speed use `txWalFlush` instead *(since v26.11.1)*	Boolean	true
`log.impl`	Logger implementation. `default` uses `java.util.logging`; `slf4j` routes the logs through the SLF4J facade, so an application that embeds ArcadeDB receives them in its own logging backend. An unrecognized value is reported and falls back to `default`	String	default
`opencypher.idBucketBits`	Number of bits reserved for the bucketId when packing a RID into the numeric value returned by the Cypher `id()` function (and SQL's `.asCypherRID()` method). Out of the 63 usable bits (the sign bit is kept clear so `id(n)` is never negative), this many go to the bucketId and the rest to the record position. The default of 16 allows up to 65536 buckets and ~1.4e14 positions per bucket. Increase it for databases with many buckets, decrease it for buckets holding a very high number of records. Must be between 1 and 31. Changing it alters the `id()` output, so encode and decode must use the same value	Integer	16
`parallelScanProducerPoolThreads`	Maximum number of threads in the JVM-wide pool that runs the producer tasks of parallel bucket scans. Kept apart from the query parallelism pool because producers block on the result queue of their query. `0` means the number of available cores (at least 2); consider an explicit cap on machines with very many cores	Integer	0

Name	Description	Type	Default Value
`polyglotEngineEnabled`	Enables the GraalVM Polyglot engine that registers scripting languages (js, python, ...) as query engines. When `false` no polyglot language is registered, which saves tens of MB of heap. The `low-ram` profile sets it to `false`. Since v26.11.1 the engine is built the first time a script language is used, or the list of the languages is asked for (as the server does when it starts): a SQL or openCypher statement no longer builds it, which used to cost about a third of the first statement of an embedded JVM (issue #10245)	Boolean	true
`profile`	Specify the preferred profile among: default, high-performance, low-ram, low-cpu	String	default
`queryAdmissionHeapWatermark`	Percentage of `queryMaxHeapRAM` above which the query admission gate (see `queryMaxConcurrent`) stops starting new queries and lets them wait in its queue until the running ones give heap back. A running query keeps growing after it started, so this is headroom, not a guarantee: the heap budget still refuses the query that would exceed it. When no query admitted by the gate is running, the next one starts whatever the heap, so a budget held by other work (for example queries run through the embedded API) cannot stall the queue for good. Ignored when the gate or the heap budget is disabled. 0 or a negative value admits on the number of running queries alone. The profiler counts as `queryAdmissionHeapDeferrals` the times the head of the queue had a free slot but found the heap above the watermark *(since v26.10.2)*	Integer	80

Name	Description	Type	Default Value
queryMaxConcurrent	Maximum number of requests of remote clients that may run at once in the JVM, across every database (see Concurrent Requests for how the queue works and how to size it). It counts every HTTP request that runs work in a database (query, command, batch load, vector and full-text search, time series, PromQL and Grafana), and the requests received over Postgres, BOLT, Redis, gRPC, Gremlin Server, MCP and MongoDB. A request that arrives when this many are running, or when the heap the running queries hold reserved is above queryAdmissionHeapWatermark, waits in a queue and starts in arrival order (FIFO) as soon as both allow it, instead of running at once and risking a refusal from the heap budget (queryMaxHeapRAM). The wait happens before anything of the request runs, never in the middle of it, and a query started from inside a running one (a function, a script) shares its slot. A request that waits longer than queryQueueTimeout, or that finds queryQueueMaxSize requests already waiting, fails with QueryAdmissionException, which every protocol answers as retryable: HTTP 503, Postgres SQLSTATE 40001, a BOLT transient error, Redis TRYAGAIN, gRPC ABORTED, MongoDB ExceededTimeLimit (262). Nothing of it ran, so it can be retried as it is. Waiting requests park the thread that received them, so keep queryQueueMaxSize below server.httpWorkerThreads to leave HTTP threads for the other requests. The MongoDB protocol runs its requests on threads shared by other connections, so it never waits: a request that cannot start at once is refused. Never held behind the queries: beginning and rolling back a transaction (HTTP sessions, gRPC), the Grafana health probe, Redis PING, the MCP tools that only report on the server, the gRPC admin service. Not gated: the embedded API, which the server's own work (security, schema, replication, triggers) also uses, Gremlin Server sessions and the gRPC client-streaming loads. A request holds its slot for as long as it runs, so with long analytical queries next to short ones, size the limit for both: a handful of long queries holding every slot makes the short ones wait and, past the timeout, be refused. A transaction that took explicit locks (LOCK TYPE, LOCK BUCKET) keeps them while one of its requests waits in the queue, which lengthens the wait of the writers that need them; a request waiting for an explicit lock gives its slot back first, so the lock holder can always be admitted. A	Integer	(2 x cores, min 4)

Name	Description	Type	Default Value
queryMaxHeapRAM	Maximum heap (in MB) the in-memory buffers of all the queries running in the JVM may hold at once, across every database: the rows an ORDER BY sorts, the keys a DISTINCT or a UNION remembers, the groups a GROUP BY keeps, the result of a LET subquery, the records a TRAVERSE has visited and the rows it keeps queued to expand (with their $path and $stack) *(since v26.12.1)*, the records a MATCH prefetches, the nodes a MATCH while:/maxDepth: traversal holds along its current path *(since v26.12.1)*, the record addresses an UPDATE reads ahead of its writes, the values an aggregate like list() or percentile() holds *(since v26.10.2)*, and in OpenCypher also the values collect() gathers, the rows a Cartesian product or a hash join buffers and the eager read ahead of a write. Every buffer reserves its estimated size from this budget as it grows and gives it back once its rows are served, when its query closes, or - for a result set the application abandoned without closing it - once the garbage collector reclaims it. Where queryMaxHeapElementsAllowedPerOp stops one runaway query, this budget stops many large queries running together from exhausting the heap although each one stays under that limit. A query whose buffers would take the reservations past the budget fails with QueryHeapBudgetExceededException, which is transient: the same query can succeed once the others release what they hold, and HTTP answers it with 503 (the other protocols classify it as retryable). A query that alone needs more than the whole budget fails with a HeapLimitExceededException instead (a CommandExecutionException, as it was before v26.11.1, now typed like the refusal of queryMaxHeapElementsAllowedPerOp), since no retry can help it. The first 256KB a query buffers are not reserved, so small queries never contend on the budget. A GROUP BY computed by a parallel scan reserves about one copy of its groups plus a small partial per worker, whatever the number of cores: a worker merges its groups into one shared set once it holds its share of the budget, and a GROUP BY of many keys aggregates the keys a worker does not hold in one shared set partitioned by key (see queryParallelScan) *(since v26.10.2)*. The reservations in use, their peak and the refusals are reported by the profiler as queryHeapReserved, queryHeapReservedPeak, queryHeapBudget and queryHeapRefusals. A scan of a type with large records reads ahead less as the	Long	(heap/2)

Name	Description	Type	Default Value
`queryParallelismPoolThreads`	Maximum number of threads in the JVM-wide pool that runs query-time parallel work (graph algorithms, parallel index scans). Used instead of the JDK common `ForkJoinPool`, which is shared with user code and has no back pressure. `0` means the number of available cores (at least 2)	Integer	0
`queryParallelismQueueSize`	Maximum number of tasks that can wait in the queue of the query parallelism pool (see `queryParallelismPoolThreads`). When the queue is full the submitting thread runs the task itself, which lowers parallelism but never fails the query	Integer	1024
`queryQueueMaxSize`	Maximum number of queries that may wait in the queue of the query admission gate (see `queryMaxConcurrent`) at once. A query that finds the queue full fails at once with `QueryAdmissionException` (HTTP `503`) without having run. Each waiting query parks the HTTP worker thread that received it, so this bounds how many of those threads the queue can take away from the other requests. The `low-ram` profile sets it to 8. 0 or a negative value disables the queue: a query that cannot start at once is refused *(since v26.10.2)*	Integer	256
`queryQueueTimeout`	Maximum time in milliseconds a query waits in the queue of the query admission gate (see `queryMaxConcurrent`) before it fails with `QueryAdmissionException` (HTTP `503`) without having run. 0 or a negative value does not wait: a query that cannot start at once is refused *(since v26.10.2)*	Long	30000
`restore.threads`	Number of threads used to restore a full backup, one ZIP entry per thread. `-1` (default) uses the available cores, at most 8; `0` selects the legacy single-threaded restore. An archive read over HTTP(S) or an encrypted archive is always restored sequentially	Integer	-1

Name	Description	Type	Default Value
`sparseVectorScoringMaxPartitions`	Maximum number of RID ranges one `LSM_SPARSE_VECTOR` top-K query is split into for parallel scoring. `1` keeps every query on the calling thread. `0` (default) lets the engine decide per query: it only claims workers no other query needs, and stops splitting once enough queries are in flight to keep the pool busy, so an idle server spends spare cores on latency while a busy one keeps its throughput. A value above 1 forces the split regardless of load, which costs throughput on a busy server (measured at 16 concurrent clients, a forced 8-way split gave about half the throughput of no split); use it only when a single query at a time must be as fast as possible. Re-read on every query	Integer	0
`sparseVectorScoringMinPostingsForPartitioning`	Minimum number of postings (the summed document frequency of the query terms) an `LSM_SPARSE_VECTOR` top-K query must traverse before it is split into parallel ranges. Below it the query stays on the calling thread. With the default `sparseVectorScoringMaxPartitions` of 0, the number of ranges also follows this figure, one range per half of it. Raise it on a small or shared pool where CPU matters more than latency. Re-read on every query. The default was 200000 before v26.10.2	Long	50000
`sparseVectorScoringPoolThreads`	Maximum number of threads in the JVM-wide pool that runs the parallel scoring of `LSM_SPARSE_VECTOR` top-K queries: per-bucket scoring on types with several buckets, and the split of one index traversal into RID ranges (see `sparseVectorScoringMaxPartitions`). `0` means the number of available cores (at least 2). Needs a JVM restart: the pool is created once, on first use	Integer	0
`sparseVectorScoringQueueSize`	Maximum number of tasks that can wait in the queue of the sparse-vector scoring pool. When the queue is full the submitting thread runs the task itself, which lowers parallelism but never fails the query. Needs a JVM restart	Integer	1024

Name	Description	Type	Default Value
`sparseVectorScoringTimeoutSeconds`	Deadline, in seconds, for the parallel fan-out of an `LSM_SPARSE_VECTOR` top-K query, shared by all its per-bucket tasks. On expiry the pending tasks are cancelled and the query fails with a descriptive error. `0` disables the timeout (not recommended in production). Values below 5 seconds can cause spurious failures on a saturated host. Re-read on every query	Integer	30
`sparseVectorScoringWindow`	Size, in RID positions, of the window the `LSM_SPARSE_VECTOR` top-K traversal scores at a time. Measured about 1.8x faster at the median and 2.4x at p99 than the document-at-a-time traversal on learned-sparse corpora. Scores can differ in the last bit, so the order among exact ties may change. `0` or a negative value selects the classic document-at-a-time Block-Max MaxScore traversal. Rounded up to a multiple of 64 and capped at 1048576. Re-read on every query.	Integer	16384
`test`	Tells if it is running in test mode. This enables the calling of callbacks for testing purpose	Boolean	false
`vectorIndex.maxConcurrentRebuilds`	Maximum number of vector index graph rebuilds that can run at the same time across all databases. Rebuilds use a lot of memory, so running many in parallel can exhaust it. `1` serializes all rebuilds	Integer	1
`vectorIndex.rebuildPermitTimeoutMs`	Maximum time in milliseconds an asynchronous vector index rebuild waits for a rebuild permit (see `vectorIndex.maxConcurrentRebuilds`) before it skips that cycle. A skipped cycle is retried at the next trigger	Integer	600000

Available Plugins

The following plugins are available for ArcadeDB Server:

- `MongoDB` (`com.arcadedb.mongo.MongoDBProtocolPlugin`) - Implements the MongoDB wire protocol
- `Postgres` (`com.arcadedb.postgres.PostgresProtocolPlugin`) - Implements the Postgres wire protocol
- `Redis` (`com.arcadedb.redis.RedisProtocolPlugin`) - Implements the Redis wire protocol
- `Http` (`com.arcadedb.server.http.HttpServerPlugin`) - Implements the HTTP/REST API
- `gRPC` (`com.arcadedb.server.grpc.GrpcServerPlugin`) - Implements the gRPC API. Bundled in the `full` distribution but not started until it is added to `server.plugins` (see gRPC settings)

SERVER

Name	Description	Type	Default Value
`backup.enabled`	State of backup lock. Disable for increased performance of massive insertions.	Boolean	True
`bolt.debug`	Prints the BOLT protocol messages to the console	Boolean	false
`bolt.defaultDatabase`	Database used by BOLT connections that name none. When not set, the first available database is used	String	
`bolt.host`	Host or interface the BOLT protocol plugin listens on	String	0.0.0.0
`bolt.maxConnections`	Maximum number of concurrent BOLT connections. `0` means unlimited	Integer	0
`bolt.maxMessageSize`	Maximum total size in bytes of one BOLT message after its chunks are reassembled, so a client that never sends the end-of-message chunk cannot grow the buffer without bound	Integer	16777216
`bolt.maxOpenStreams`	Maximum number of result streams one BOLT connection may keep open at the same time inside an explicit transaction	Integer	1024
`bolt.packstream.maxDepth`	Maximum nesting depth of a PackStream value (list, map, structure) decoded on the BOLT protocol	Integer	1000
`bolt.packstream.maxElements`	Maximum number of elements a PackStream list, map or structure may declare on the BOLT protocol	Integer	1048576
`bolt.packstream.maxValueLength`	Maximum length in bytes of one PackStream `BYTES` or `STRING` value on the BOLT protocol. A larger declared length is refused before any allocation	Integer	16777216
`bolt.port`	TCP port of the BOLT protocol plugin	Integer	7687
`bolt.routing.ttl`	Time to live, in seconds, of the BOLT routing table entries	Long	300
`bolt.ssl`	TLS mode of BOLT connections, using the shared SSL key/trust store settings (`ssl.*`). There is no in-band upgrade: the server detects TLS from the first bytes of each connection. `DISABLED` (no TLS), `OPTIONAL` (TLS or plaintext on the same port, detected per connection: a client that falls back to plaintext sends its credentials in the clear) or `REQUIRED` (TLS only, a plaintext connection is closed; recommended once certificates are in place)	String	DISABLED
`bolt.websocket.maxFrameSize`	Maximum payload size in bytes of one BOLT WebSocket frame. A frame declaring a larger size is refused before any allocation	Integer	16777216

Name	Description	Type	Default Value
`ha.addPeerProbeTimeout`	Milliseconds to wait for a TCP connection to a peer's Raft address before an add-peer request (`POST /api/v1/cluster/peer`, `connect cluster`, gRPC `ConnectCluster`) is refused as unreachable with HTTP 400 / gRPC `INVALID_ARGUMENT`. A successful probe is not proof of a healthy peer and changes nothing. `0` skips the probe.	Long	2000
`ha.appendBufferSize`	AppendEntries batch byte limit for replication (e.g. '32MB').	String	32MB
`ha.appendElementLimit`	Maximum number of Raft log entries per AppendEntries batch. Bounds the per-batch in-memory footprint on the follower during catch-up resync, where many batches may queue before the state machine can apply them. Lowering this value reduces peak heap pressure on followers catching up from a far-behind state. The byte limit (`ha.appendBufferSize`) remains the dominant per-batch heap bound; this element count is the secondary cap that governs when entries are small enough that many fit under the byte limit. Must be a positive integer (>= 1).	Integer	64
`ha.applyBacklogMax`	Byte budget for the Raft entries a follower may hold received but not yet applied, for example `256MB`. Those entries stay on the follower's heap until applied, so while the leader estimates that a follower is over the budget it sends that follower heartbeats only, and resumes when the follower has caught up. Every node resolves the budget from its own heap and setting and reports it to the leader, and the bound applied to a follower is the smaller of the leader's budget and that follower's own, so a follower with a smaller heap than its leader is held to its own budget. A follower that reports none (an older version, or one where the setting is 0) is held to the leader's. It is a soft limit: a follower can overshoot it during a write burst between two 5-second capability polls. Empty (default) means a quarter of the maximum heap, never less than 32MB. `0` disables the back-pressure this node applies as a leader, whatever its followers report *(since v26.12.1)*	String	

Name	Description	Type	Default Value
`ha.autoAcquireDatabases`	When `true`, a node that joins the cluster pulls (full snapshot install) every database of the leader it has never seen locally, so a new empty node becomes a full replica without manual steps. When `false` it only refreshes the databases it already has. A per-node policy, read on every reconcile	Boolean	true
`ha.clientElectionRetryCount`	Number of retries performed by RemoteDatabase after receiving HTTP 503 NeedRetryException during an election.	Integer	3
`ha.clientElectionRetryDelayMs`	Delay in ms between RemoteDatabase election retries. A longer `Retry-After` sent by the server takes precedence, up to `network.retryAfterMaxWait`.	Long	2000
`ha.bootstrapFromLocalDatabase`	When true (default), at first cluster formation peers exchange `(fingerprint, lastTxId)` per database; the highest-`lastTxId` peer is elected as bootstrap source via leadership transfer. Matching peers bootstrap locally with zero bytes transferred; mismatched peers reinstall the leader-shipped full snapshot.	Boolean	true
`ha.bootstrapTimeoutMs`	Maximum time in ms the bootstrap leader waits for every configured peer to report its bootstrap state before proceeding.	Long	120000
`ha.clusterName`	Cluster name. Useful in case of multiple clusters in the same network	String	arcadedb
`ha.clusterToken`	Shared secret for inter-node authentication. If empty, the token is derived at every start from `ha.clusterName` and the root password, see HA security.	String	
`ha.clusterTokenAllowRotation`	Lets a node whose cluster token is derived from the root password (file or inline) start with a token that differs from the one it last ran with. By default such a node refuses to start. Set it on every node for one deliberate rotation, then remove it. *Since v26.11.1*	Boolean	false
`ha.clusterTokenPath`	Path to a file containing the shared secret for inter-node authentication. Read only when `ha.clusterToken` is not set; the file content is trimmed of surrounding whitespace. Keeps the secret off the command line (e.g. a Kubernetes Secret mounted on `tmpfs`).	String	null

Name	Description	Type	Default Value
`ha.divergedFollowerMaxReformats`	Maximum number of automatic Raft storage reformats (see `ha.divergedFollowerRecovery`) in one divergence episode before the follower stops and logs a SEVERE message for the operator. The budget is restored once the follower has looked healthy for five times the recovery window. `0` means no limit	Integer	5
`ha.divergedFollowerRecovery`	When `true`, a follower stuck at a stale term (it sees a leader at a newer term and has applied everything it can, yet its last applied entry is from an older term) reformats its Raft storage and rejoins as a fresh peer, and the leader re-syncs it with a snapshot. Destructive for the local Raft storage only: the database files are kept and re-synced. The condition must last `ha.divergedFollowerRecoveryDurationMs`, and `ha.divergedFollowerMaxReformats` bounds the retries. Since v26.10.2 a follower whose Raft layer was restarted in place, and that has taken no entry and seen no new leader since, is not reformatted: it raises the `follower-stuck-at-stale-term` alert and must be restarted by hand. `false` leaves a manual restart as the only remedy	Boolean	true
`ha.divergedFollowerRecoveryDurationMs`	How long in ms a follower must stay stuck at a stale term, without its applied index advancing, before it reformats its Raft storage and rejoins. Any advance of the applied index restarts the window. Floored at 2x `ha.electionTimeoutMax`. This window used to be `ha.staleFollowerRecoveryDurationMs` (default 60000); a deployment that raised the old setting to delay the reformat must raise this one instead.	Long	20000
`ha.electionTimeoutMin`	Minimum Raft election timeout in ms. Increase for high-latency WAN clusters.	Integer	5000
`ha.electionTimeoutMax`	Maximum Raft election timeout in ms. Increase for high-latency WAN clusters. Must be above `ha.electionTimeoutMin`: a value at or below it is widened to twice the minimum (with a warning), because without a spread between the two the nodes of a cluster that lost its leader keep splitting the vote. To shorten the timeout, lower both.	Integer	10000
`ha.enabled`	True if HA is enabled for the current server	Boolean	false
`ha.errorRetries`	Number of automatic retries in case of IO errors with a specific server. 0 (default) is to retry against all the configured servers	Integer	0

Name	Description	Type	Default Value
`ha.forwardLeaderWaitTimeoutMs`	Maximum time in milliseconds a follower waits for a leader to be elected before it fails a write it has to forward to the leader, instead of failing at once during an election. It also bounds how long a follower holds back a refusal that names no leader. `0` restores the fail-fast behavior	Long	20000
`ha.groupCommitMaxQueuedBytes`	Maximum total bytes of transactions waiting in the Raft group-commit queue, next to the entry count bound `ha.groupCommitQueueSize`. When it would be exceeded the server waits up to `ha.groupCommitOfferTimeout` and then refuses the transaction with a retryable `ReplicationQueueFullException`. Must be at least `ha.grpcMessageSizeMax`	Long	268435456
`ha.grpcAllowlistRefreshMs`	Rate-limiting interval in ms for DNS re-resolution in the gRPC peer address allowlist filter.	Long	30000
`ha.grpcFlowControlWindow`	gRPC flow control window size in bytes for Raft AppendEntries traffic. Larger values help catch-up replication after partitions.	Long	4194304
`ha.groupCommitBatchSize`	Maximum number of Raft log entries to batch in a single group commit flush. Higher values improve throughput under concurrent load.	Integer	500
`ha.groupCommitOfferTimeout`	Timeout in ms waiting for space in the group-commit queue before throwing ReplicationQueueFullException.	Integer	100
`ha.groupCommitQueueSize`	Maximum pending transactions allowed in the Raft group-commit queue. When full, the server sheds load by throwing ReplicationQueueFullException (NeedRetryException).	Integer	10000
`ha.grpcMaxConnectionAgeGraceMs`	How long in ms the RPCs still running on a Raft gRPC connection that reached `ha.grpcMaxConnectionAgeMs` have to finish before the connection is closed. Read only when that setting is non-zero. `0` cancels them at the age boundary.	Long	5000

Name	Description	Type	Default Value
`ha.grpcMaxConnectionAgeMs`	How long in ms an inbound Raft gRPC connection may live before the server closes it with a graceful GOAWAY, whether busy or idle. Bounds a connection that is never idle, such as a removed peer that keeps campaigning, at the price of recycling healthy replication streams once per period. gRPC applies a +/-10% jitter and a 1 second floor. Set it well above the Raft election timeout. `0` leaves connections unbounded in age.	Long	0
`ha.grpcMaxConnectionIdleMs`	How long in ms an inbound Raft gRPC connection may carry no RPC before the server closes it with a graceful GOAWAY. Closes quiet connections, such as the socket of a peer removed from the allowlist; it never reaps an active replication stream. Values below 1 second are raised to 1 second. `0` leaves connections unbounded.	Long	300000
`ha.grpcMessageSizeMax`	Maximum size in bytes of a single Raft gRPC message, that is a replicated transaction or schema entry. Higher than the Ratis default of 64MB so bulk-load batches are not refused. With `ha.appendBufferSize` it sets the largest replicated entry. Since v26.10.2 the engine also honors a value set only in `server-configuration.json` when it sizes vector bulk-load chunks and time-series compaction slices; before, it saw only a value set as a system property	Long	134217728
`ha.healthCheckInterval`	Interval in ms for the Raft health monitor to check for CLOSED/EXCEPTION state and auto-recover. The same tick also restarts in place a node whose Raft log writer failed (for example `No space left on device`) once the Raft storage volume has room again. 0 disables.	Long	3000
`ha.idempotencyCacheMaxBodyBytes`	Maximum size in bytes of a single response body eligible for caching in the HTTP idempotency cache. Larger responses are not cached, so a retry of that request executes again. See reference/http-api/http.pdf	Long	1048576
`ha.idempotencyCacheMaxBytes`	Maximum total size in bytes of the cached response bodies in the HTTP idempotency cache. Oldest entries are evicted when exceeded.	Long	67108864
`ha.idempotencyCacheMaxEntries`	Maximum number of entries in the HTTP idempotency cache.	Integer	10000

Name	Description	Type	Default Value
`ha.idempotencyCacheTtlMs`	Time-to-live in ms for entries in the HTTP idempotency cache: how long a retry with the same `X-Request-Id` is replayed rather than executed again. See reference/http-api/http.pdf	Long	60000
`ha.jvmPauseCloseThresholdMs`	JVM-pause length in ms above which Ratis closes this node's Raft division (Ratis default: 60000). `0` or a negative value disables the close: a long pause still steps a leader down, and the ArcadeDB health monitor decides whether the node needs recovery. Set a positive value to restore the Ratis behavior.	Long	0
`ha.k8s`	The server is running inside Kubernetes (enables auto-join on scale-up)	Boolean	false
`ha.k8sSuffix`	When running inside Kubernetes use this suffix to reach the other servers. Example: arcadedb.default.svc.cluster.local	String	
`ha.logCacheSize`	Heap budget for the Raft log entries each server keeps decoded in memory, for example `128MB`. Empty (default) means one eighth of the maximum heap, at most 200MB. It also decides how many closed log segments are cached, and so what a restarted server loads into the heap before catching up: on a small heap keep `ha.logSegmentSize` at no more than half of this value. Entries outside the cache are read back from disk *(since v26.10.2)*	String	
`ha.logPurgeGap`	Number of Raft log entries retained after a snapshot as a buffer for slightly lagging followers. Lower values free disk faster but raise the chance a slow follower needs a full snapshot resync.	Integer	1024
`ha.logPurgeUptoSnapshot`	When true, deletes old Raft log segments after each snapshot to bound disk growth. Set to false to retain full log history for debugging/auditing.	Boolean	true
`ha.logSegmentSize`	Maximum Raft log segment size (e.g. '64MB', '128MB').	String	64MB
`ha.logVerbose`	Verbose HA logging: 0=off, 1=basic (elections), 2=detailed (replication), 3=trace (every state machine apply).	Integer	0

Name	Description	Type	Default Value
`ha.membershipChangeTimeout`	Time in ms an add-peer or remove-peer keeps re-issuing the Raft configuration change before reporting it as not committed. Bounds the whole request, not one attempt: an attempt that cannot finish in what is left of the budget is not started. A peer that answers a connection but never catches up (a legitimately large snapshot install, or a listener that is not a peer of this cluster) is what reaches the full budget; a peer that belongs to a different cluster is refused at once instead. Lower it below a load balancer's idle timeout so you get the error rather than a dropped connection, remembering that an attempt whose network call hangs can take up to about 31s on its own. Read at server start.	Long	90000
`ha.peerAllowlist.enabled`	Reject inbound Raft gRPC connections whose remote address does not resolve to a cluster peer. The accepted hosts are those in `ha.serverList` (expanded with `ha.k8sSuffix` when `ha.k8s` is set), plus the current members of the cluster, which are picked up automatically so a node that joins at runtime is accepted; on Kubernetes the pods of the headless service named by `ha.k8sSuffix` are accepted too, so a StatefulSet scale-up completes without changing any setting. Loopback is always allowed. Not a substitute for mTLS on untrusted networks.	Boolean	true
`ha.peerAllowlistStartupGraceMs`	Startup grace window in ms during which the gRPC peer allowlist fails OPEN (accepts and logs a warning) for an unmatched address, as long as it has never resolved every host in `ha.serverList` at least once. Prevents a self-inflicted partition on Kubernetes, where a peer's headless-service DNS record is only published once its pod is Ready, so a restarting peer connects before its own name resolves. Measured from filter creation; enforces normally once all peers resolve once or the window elapses. 0 disables fail-open.	Long	60000
`ha.peerAllowlistStickyTtlMs`	How long in ms the gRPC peer allowlist keeps the last successfully-resolved IPs of a peer host when a later DNS re-resolution of that host fails. Bridges transient DNS outages and pod-IP churn so a peer that resolved moments ago is not evicted by a momentary lookup failure. 0 disables stickiness.	Long	300000

Name	Description	Type	Default Value
`ha.peerChannelResetDuration`	Time in ms a follower must stay continuously unreachable before the leader resets that one follower's replication gRPC channel, so the next send re-resolves DNS and reconnects. Recovers a leader appender stuck on a stale DNS result after a follower restarts with a new address (e.g. a Kubernetes pod-IP change). Retried once per interval, up to 5 attempts, then the leader gives up or escalates (see `ha.peerChannelResetEscalation`); the counter re-arms when the follower reconnects. Only the unreachable peer's channel is touched. Requires `ha.peerUnreachableThreshold` > 0. Set to 0 to disable. See Replication-Channel Self-Healing.	Long	60000
`ha.peerChannelResetEscalation`	When the `ha.peerChannelResetDuration` retry budget is exhausted and the follower's replication channel is still dead, transfer leadership to a healthy peer so the new leader builds a fresh appender to that follower. Without it the leader stays wedged until an operator restarts the process, because the reset streak only re-arms when the follower becomes reachable again. The target is chosen with the same rules as a manual step-down and is never the wedged follower; when no healthy target exists the leader logs for operator intervention instead. Each healthy peer escalates a given follower at most once per 30-minute cooldown, so leadership churn is bounded rather than perpetual. Set to `false` to only log. See Replication-Channel Self-Healing.	Boolean	true
`ha.peerUnreachableThreshold`	Time in ms since the last successful RPC to a follower before the leader reports it as unreachable in the resync narrative. Also the "unreachable" signal `ha.peerChannelResetDuration` depends on. Does not change Raft membership or quorum. Set to 0 to disable, which also disables the channel-reset recovery built on it.	Long	10000
`ha.proxyBatchReadTimeout`	Timeout in ms a follower waits for the leader to answer a batch load it relayed via `POST /api/v1/batch`, before answering the client HTTP 504. Separate from `ha.proxyReadTimeout` because a bulk load's duration depends on the payload size, not a fixed control-plane budget.	Long	600000

Name	Description	Type	Default Value
`ha.proxyCommandTimeout`	Fallback timeout in ms a follower waits for the leader to answer a forwarded SQL/DDL write, used only when the database has no `arcadedb.command.timeout` of its own. The budget is read from the database's configuration, not from the configuration passed to the call.	Long	3600000
`ha.proxyConnectTimeout`	Connect timeout in ms for the leader proxy (replica-to-leader forwarding). It also applies when the cluster uses SSL, where the connection previously had a fixed 5s timeout the setting could not change. Read when the connection is created, so a change needs a restart.	Long	5000
`ha.proxyLongCommandTimeout`	Milliseconds a follower waits for the leader to answer a forwarded `restore backup`, `restore database` or `import database` before answering the client HTTP 504. These commands legitimately run for minutes, so `ha.proxyReadTimeout` does not apply to them. `0` or a negative value does not disable it.	Long	3600000
`ha.proxyReadTimeout`	Read timeout in ms for the leader proxy. Covers long-running queries forwarded from a replica.	Long	30000
`ha.quorum`	Write quorum: `majority` or `all`. *The legacy values* `none`, `one`, `two`, `three` *are not supported*	String	majority
`ha.quorumTimeout`	Timeout in ms waiting for the quorum acknowledgment	Long	10000
`ha.raftPort`	TCP/IP port for Raft gRPC communication. Used as the default when HA_SERVER_LIST entries do not specify an explicit port.	Integer	2434
`ha.raftPersistStorage`	If true, the Raft storage directory is preserved across restarts, enabling node rejoin by replaying its persisted log instead of a full snapshot resync. **Defaults to `true`**: wiping the Raft log on restart could permanently diverge a lagging follower after a full-cluster cold restart (`WALVersionGapException`) or silently re-form a fresh single-node cluster. Ensure `ha.raftStorageDirectory` lives on durable storage. A throwaway/test cluster can opt out with `false`.	Boolean	true

Name	Description	Type	Default Value
ha.raftStorageDirectory	Parent directory under which the per-node Raft storage sub-folders (raft-storage-<nodeName>) are created. When empty (the default), the server root path (arcadedb.server.rootPath) is used. Because ha.raftPersistStorage now defaults to true, this directory must live on **durable** storage (a persistent volume, not an ephemeral/tmpfs mount) so a restarted node can replay its log and rejoin without a full snapshot resync. Set this to a writable, statically-named path (e.g. a PVC mount) on Kubernetes StatefulSets with readOnlyRootFilesystem: true, where the dynamically-named raft-storage-* directory under the read-only server root cannot be created or mounted.	String	
ha.raftStorageMinFreeSpacePerc	Percentage of free space on the volume of the Raft storage directory below which the periodic snapshot tick forces a snapshot and log purge and logs a WARNING. Guards against a full volume, after which the node rejects every append until restarted. 0 disables the check	Integer	20
ha.ratisCloseTimeoutMs	Maximum time in milliseconds the server waits for the Raft server to close, on shutdown and before an in-place Raft restart. A close that takes longer is logged at SEVERE with its stack and left running; a shutdown then goes on, and an in-place restart fails (counting toward ha.ratisRestartMaxRetries) while that close still runs. 0 or a negative value waits without a bound *(since v26.10.2)*	Long	30000
ha.ratisRestartMaxRetries	Maximum consecutive Ratis restart attempts by the health monitor before the server shuts down for cluster-level recovery. Also bounds, per incident, the in-place restarts that recover a failed Raft log writer.	Integer	10
ha.readConsistency	Default read consistency for replica reads: eventual, read_your_writes, linearizable.	String	read_your_writes
ha.replicationChunkMaxSize	Maximum channel chunk size for replicating messages between servers	Integer	16777216
ha.replicationLagWarning	Raft log index gap threshold for replication lag warnings. 0 to disable.	Long	1000
ha.resyncCatchupLagThreshold	Minimum apply backlog (entries a follower received but has not applied yet) before the catch-up narrative of ha.resyncProgressLogging is logged, so the small backlog of normal write load is not reported	Long	1000

Name	Description	Type	Default Value
`ha.resyncProgressInterval`	Minimum interval in milliseconds between two resync progress log lines of a follower	Long	5000
`ha.resyncProgressLogging`	Logs a concise narrative of followers becoming unreachable and reconnecting on the leader, and of the resync progress (log catch-up, snapshot download) on a restarting follower. It does not control the raw Apache Ratis retry messages, which are set by the `GrpcLogAppender` level in `arcadedb-log.properties`	Boolean	true
`ha.schemaDelta`	Ships a schema change to the followers as a delta against the schema they already hold, instead of the whole schema document on every DDL. On a large schema that is kilobytes instead of megabytes per Raft entry. The leader sends the whole document anyway after it becomes leader, for a compaction entry, an entry shipped in instalments, a change too large for a delta to pay off, and whenever any peer has not confirmed it can decode a delta, so a rolling **upgrade** needs no ordering. A rolling **downgrade** to an earlier version that cannot decode a delta does: see Schema delta replication. While on, the leader keeps one parsed copy of the schema per replicated database in heap. Turning it off is safe at any time.	Boolean	true
`ha.schemaIncrementalApply`	A follower applying a DDL entry creates only the components for the files that entry added, instead of rebuilding every component in the database. The full rebuild costs time proportional to all the files per entry, which made building a large schema quadratic (a 1209-type schema took about 2h53m to replicate). Entries the incremental path cannot express (a file retired, a compacted index, a bloom filter, a new dictionary) fall back to the full rebuild on their own, and since v26.10.2 each fallback is logged at INFO with its reason and duration and counted in the `arcadedb.ha.schema.full_load` metrics (see how-to/operations/ha.pdf). Read per entry, so a change takes effect without a restart. Set to `false` only to rule out a suspected incremental-apply bug.	Boolean	true

Name	Description	Type	Default Value
`ha.snapshotInterval`	Interval in milliseconds between periodic Raft snapshot checkpoints on every node, so the retained Raft log is bounded by age on a cluster with few writes and not only by `ha.snapshotThreshold`. A tick does nothing when fewer than `ha.snapshotMinEntries` entries were applied since the last snapshot. It also bounds how fast the free-space check of `ha.raftStorageMinFreeSpacePerc` reacts. `0` disables it	Long	300000
`ha.snapshotMinEntries`	Minimum number of Raft log entries applied since the last snapshot before a periodic `ha.snapshotInterval` tick takes a new one. Values below 1 are raised to 1	Long	64
`ha.staleFollowerLagThreshold`	Number of Raft log entries a follower may lag behind the commit index, while not catching up, before the health monitor starts a snapshot download from the leader. Must stay below `ha.snapshotThreshold`. `0` disables this recovery	Long	10000
`ha.staleFollowerRecoveryDurationMs`	How long, in milliseconds, the lag of `ha.staleFollowerLagThreshold` must last before recovery starts, so transient catch-up lag is ignored. The stuck-at-stale-term reformat has its own window, `ha.divergedFollowerRecoveryDurationMs`	Long	60000
`ha.tsMaxSealedInlineSize`	Maximum size in bytes of time-series sealed-store content one Raft entry carries during a replicated compaction. A larger sealed store is shipped as a sequence of slices of at most this size, which the follower stages and installs at once. Always clamped to the replicated entry limit, the smaller of `ha.grpcMessageSizeMax` and `ha.appendBufferSize`. Since v26.10.2 a value set only in `server-configuration.json` is honored; before, only a system property reached the engine	Long	50331648

Name	Description	Type	Default Value
`ha.txSchemaCheck`	Have every transaction a node replicates carry the Raft log index that node had applied when the transaction began, so the leader can refuse a transaction prepared before a schema change it has already applied. Without it a replica that has not yet applied a committed `CREATE INDEX` can ship a transaction whose record ends up missing from the new index on every node, with no error. The refusal is a retryable `ConcurrentModificationException`, so expect retries on replicas during DDL; callers that do not retry (the default for plain HTTP commands) see the error. The index is written only once every peer advertises the capability, so a mixed-version cluster is unprotected until the last node is upgraded. This setting controls only whether THIS node states an index.	Boolean	true
`ha.securityConvergenceReadinessTimeout`	How long in ms `/api/v1/ready` (and the gRPC `Ready` RPC) keeps answering NOT READY on a node that may be enforcing stale users, groups or API tokens: one added to the cluster while running (`POST /api/v1/cluster/peer`, `connect cluster`, Kubernetes auto-join) that has not yet installed the cluster's security documents, or one caught up by a snapshot install whose copies the leader has not yet confirmed. Requires `server.readinessRequiresHA`. When the window expires the node reports READY and logs once at SEVERE which documents never converged. `0` disables the wait. See Health probes.	Long	30000
`ha.securityEntryCapabilityGate`	Refuse a group or API-token change (HTTP 409 / gRPC `FAILED_PRECONDITION`, naming the peer) while any peer of the Raft configuration has not proved, via `POST /api/v1/cluster/capabilities`, that it can decode the log entry the change is replicated as. Protects a rolling upgrade: an older node would halt on the unknown entry. An unreachable peer counts as a no, so admin security changes are refused while any node is down. Turn it off only when you know every node can decode that entry. Only the group and API-token entries honour it: a database configuration change (`ALTER DATABASE`) is always probed and refused.	Boolean	true

Name	Description	Type	Default Value
`ha.securitySeedRetryTimeout`	Time budget in ms a peer-admission call (`POST /api/v1/cluster/peer`, `connect cluster`) spends retrying, with exponential backoff, the security documents (`server-users.jsonl`, `server-groups.json`, `server-api-tokens.json`) it seeds to the newly joined peer. A snapshot install does not carry them. `0` leaves a single best-effort attempt.	Long	3000
`ha.serverList`	Servers in the cluster, comma-separated. Each entry uses the readable object form `[name@]host:{raft:2434,http:2480,https:2490,priority:10}` (recommended) or the positional form `[name@]host:raftPort:httpPort[:priority[:httpsPort]]`. The httpPort is required for replica-to-leader HTTP forwarding. The optional `name@` prefix gives the peer a human-readable name used in logs and Studio. The optional priority (default 0) prefers higher-valued nodes during elections. The optional httpsPort encrypts peer-to-peer transfers (e.g. snapshot download) when `ssl.enabled` is true; on a homogeneous cluster it is derived from this node's local HTTPS port when omitted. Example: `frankfurt@db1:{raft:2434,http:2480,https:2490,priority:10},london@db2:{raft:2434,http:2480,https:2490}`	String	
`ha.serverRole`	Node role in the cluster: `any` (default, eligible for leader election) or `replica` (never elected leader; Raft priority set to 0). Useful for witness/read-scale nodes.	String	any
`ha.snapshotCompressionLevel`	DEFLATE level the leader applies to the ZIP stream of a database snapshot shipped to a follower: 0 (stored) to 9 (smallest, slowest), or -1 for the Deflater default (level 6). A value outside -1..9 is ignored and the default is used. Only the serving side reads it. *Since v26.10.2: default changed from -1 to 1*	Integer	1
`ha.snapshotDownloadTimeout`	Read timeout in ms for downloading a database snapshot from the leader during follower resync.	Integer	300000
`ha.snapshotGapTolerance`	Maximum acceptable gap between the snapshot index and persisted applied index before triggering a snapshot download.	Long	10
`ha.snapshotInstallBackupWaitMs`	Milliseconds a snapshot install, or the apply of a replicated drop database, waits for a backup, export or import of the same database already running on this node before it replaces or drops the database files anyway (logging a warning). `0` does not wait.	Long	60000

Name	Description	Type	Default Value
`ha.snapshotInstallRetries`	Maximum retry attempts for snapshot download from the leader during snapshot installation.	Integer	3
`ha.snapshotInstallRetryBaseMs`	Base delay in ms for exponential backoff between snapshot download retries. Actual delay is baseMs * 2^attempt.	Long	5000
`ha.snapshotMaxConcurrent`	Maximum number of concurrent snapshot downloads served by the leader. Requests over this limit receive HTTP 503.	Integer	2
`ha.snapshotMaxEntrySize`	Maximum uncompressed size in bytes for a single entry in a snapshot ZIP. Decompression-bomb guard. Set 0 or a negative value to keep the built-in default: the guard cannot be switched off.	Long	10737418240
`ha.snapshotThreshold`	Number of Raft log entries after which the leader automatically takes a snapshot.	Long	100000
`ha.snapshotWatchdogTimeout`	Delay in ms before the snapshot-gap watchdog triggers a download. Floored at 4x `ha.electionTimeoutMax`.	Long	30000
`ha.snapshotWriteTimeout`	Timeout in ms for writing a snapshot to a follower. If the transfer stalls beyond this duration, the connection is force-closed.	Long	300000
`ha.stalledReplicaResyncDurationMs`	How long in ms a replica must stay continuously STALLED — its `matchIndex` not advancing while the leader keeps committing — before the **leader** forces it to resync. This is the leader-driven counterpart to a follower detecting its own lag: it covers the case where the follower cannot self-detect the stall because its own commit index never advances. A follower still at the never-appended sentinel (`matchIndex=-1` while the leader holds committed entries) is treated as STALLED regardless of the numeric lag, so it is recovered even when the leader is only a few entries ahead and the lag stays below `ha.replicationLagWarning`. The same duration doubles as the grace period before a replica's status flips from HEALTHY to STALLED, so a brief join or snapshot-install window is not misreported. Set to 0 to disable leader-driven recovery; the STALLED condition is still detected and logged.	Long	60000

Name	Description	Type	Default Value
`ha.stopServerOnReplicationFailure`	After a phase-2 replication failure, step-down is attempted first. If every step-down fails and this flag is true, the JVM exits so an orchestrator can restart. Default is false: the server keeps running and logs CRITICAL.	Boolean	false
`ha.tls.enabled`	Negotiate TLS on the Raft gRPC transport, which carries AppendEntries, RequestVote and the Ratis admin/client calls between the nodes of a cluster. When true, `ha.tls.certChainFile`, `ha.tls.privateKeyFile` and `ha.tls.trustCertCollectionFile` must all name a readable PEM file or the node refuses to start with a ConfigurationException naming the offending setting. Default is false so a development or test cluster still starts with no configuration at all; the peer-address allowlist (`ha.peerAllowlist.enabled`) is IP-based and is defeated by a spoofed source address or a compromised peer, so it is a best-effort default rather than a substitute for TLS on an untrusted network. TLS on the gRPC port does not replace the `X-ArcadeDB-Cluster-Token` check on the HTTP side channels (snapshot download, database verify), nor is it affected by `ssl.enabled`, which covers those side channels only. See mTLS for the Raft gRPC transport.	Boolean	false
`ha.tls.certChainFile`	PEM file holding this node's Raft gRPC certificate followed by any intermediate CA certificates. The certificate must carry a subject alternative name matching the address the other nodes use to dial this node in `ha.serverList` (on Kubernetes, the pod's stable headless-service DNS name, not the pod IP). Every node both accepts and initiates Raft connections, so with `ha.tls.mutualAuth=true` this one certificate is presented as the TLS server certificate AND as the client certificate: it needs BOTH the `serverAuth` and `clientAuth` extended key usages. A CA that issues single-purpose certificates otherwise produces a handshake failure the startup file checks cannot catch, because the file is perfectly readable and merely the wrong kind of certificate. Required when `ha.tls.enabled` is true.	String	

Name	Description	Type	Default Value
`ha.tls.privateKeyFile`	PEM file holding the PKCS#8 private key that matches `ha.tls.certChainFile` — the PEM armor labels it simply `PRIVATE KEY`, whereas a PKCS#1 key is labeled `RSA PRIVATE KEY` and is not accepted. A WARNING is logged when the file is readable by the group or by others, since any local account that can read it can present this node's identity to the cluster; it is a warning and not a refusal, because a mounted secret may arrive with permissions the node cannot change. Required when `ha.tls.enabled` is true.	String	
`ha.tls.trustCertCollectionFile`	PEM file holding the cluster CA certificate(s) that sign every node certificate. A peer presenting a certificate this collection does not chain to is rejected during the TLS handshake, before any Raft message is read. Required when `ha.tls.enabled` is true — including with `ha.tls.mutualAuth=false`, since the dialing node still needs the cluster CA to validate the certificate presented by the node it dials.	String	
`ha.tls.mutualAuth`	Require the dialing peer to present a client certificate signed by the CA collection in `ha.tls.trustCertCollectionFile`, binding peer identity to the certificate rather than to a source IP address. Set to false for server-only TLS, which encrypts the traffic but leaves the Raft port open to any client that trusts the cluster CA — the back door this setting exists to close, so the server logs a WARNING at startup when it is off. Turning it off does NOT make `ha.tls.trustCertCollectionFile` optional. Only meaningful when `ha.tls.enabled` is true.	Boolean	true
`ha.writeBufferSize`	Raft log write buffer size (e.g. '40MB'). Must be at least `ha.appendBufferSize` + 8 bytes; otherwise the server fails to start with ConfigurationException.	String	40MB
`network.sameServerErrorRetry`	Number of automatic retries in case of IO errors with a specific server. If replica servers are configured, look also at `HA_ERROR_RETRY` setting. 0 (default) = no retry	Integer	0

Name	Description	Type	Default Value
network.retryAfterMaxWait	Upper bound, in milliseconds, on how long the remote client honors the Retry-After a server sends with a request it refused before running it (a 503 from a node installing a snapshot): the transaction retry loop of RemoteDatabase.transaction() and the election retry loop wait at least that long before retrying, and the hint is capped at this value, so a misbehaving server cannot park the client. A random spread of up to a tenth of the hint is added on top, so the clients a node refused together do not all come back at once. The most a refused request can wait is therefore 1.1 times this value per retry: txRetries - 1 pauses for a transaction, ha.clientElectionRetryCount for a command. 0 ignores Retry-After, leaving only the retry backoff	Long	30000
network.socketTimeout	TCP/IP Socket timeout (in ms)	Integer	30000
network.remoteFetchConnectTimeout	Connect timeout, in milliseconds, for an outbound fetch of a URL you supply - IMPORT DATABASE, RESTORE DATABASE and the openCypher LOAD CSV - applied to every hop of a redirect chain. 0 means no timeout at all, so a host that neither accepts nor refuses the connection blocks the calling thread indefinitely.	Integer	30000
network.remoteFetchReadTimeout	Read timeout, in milliseconds, for an outbound fetch of a URL you supply - IMPORT DATABASE, RESTORE DATABASE and the openCypher LOAD CSV. It bounds EACH read, not the whole transfer: a large remote source may legitimately take much longer than this to arrive, as long as it keeps sending. 0 means no timeout at all, so a source that stops sending and never closes the socket - a hung HTTP connection, or a proxy holding the socket open after the origin died - blocks the calling thread for as long as that socket lives. Raise it for a genuinely slow origin rather than disabling it; when it fires, the error names the URL, the wait and this setting.	Integer	30000

Name	Description	Type	Default Value
`network.maxPreAuthConnections`	Maximum number of connections a binary wire-protocol listener (Postgres, Redis, BOLT) may hold in the phase before authentication. Each accepted socket costs one thread and one file descriptor before the client has proved who it is, and `network.socketTimeout` only bounds how long each one may stay there, not how many there can be. Past this cap the listener closes further connections immediately rather than accepting and then timing them out. The cap is per listener, so a flood against one protocol cannot use up the budget that lets clients of another log in. 0 means unlimited.	Integer	500
`network.preAuthTimeout`	Maximum time, in milliseconds, a binary wire-protocol connection (Postgres, Redis, BOLT) may stay in the phase before authentication, counted from the moment it is accepted: past it the connection is closed whatever it is doing. `network.socketTimeout` only closes a connection that sends nothing for that long, so a client that keeps sending a few bytes, or a command it is refused for, would otherwise hold its slot of `network.maxPreAuthConnections` for ever. Connections closed this way are counted in the `<protocol>.connection.preauth.expired` metric and reported in the log at most once every 10 seconds. 0 disables the limit. *Since v26.11.1*	Long	30000
`network.socketKeepAlive`	Enable TCP keepalive (SO_KEEPALIVE) on every wire-protocol socket. The Postgres and Redis executors drop the socket read timeout to infinite once a connection is authenticated, because an authenticated client legitimately holds an idle connection open, and those protocols carry no application-level heartbeat. With keepalive off, a peer that dies without a FIN/RST (host crash, silent partition) leaves the server thread blocked in a read forever, leaking a thread and a file descriptor per event. Keepalive lets the OS discover the dead peer and fail the read.	Boolean	true
`network.socketKeepAliveIdle`	Seconds an authenticated connection may sit idle before the OS sends the first TCP keepalive probe. Only applied where the JDK and the platform expose TCP_KEEPIDLE (Linux and macOS do); elsewhere the system-wide default applies, typically 2 hours. 0 leaves the system default in place.	Integer	120

Name	Description	Type	Default Value
`network.socketKeepAliveInterval`	Seconds between TCP keepalive probes once the first one has gone unanswered. 0 leaves the system default in place.	Integer	15
`network.socketKeepAliveCount`	Number of unanswered TCP keepalive probes after which the connection is declared dead. With the defaults a dead peer is detected about 3 minutes after the connection goes idle. 0 leaves the system default in place.	Integer	4
`postgres.languages`	Comma-separated query languages a statement received through the Postgres wire protocol may select with a leading `{language}` marker (for example `{cypher} MATCH (n) RETURN n`). The marker is only read as the first token of the statement. SQL is always accepted; a statement naming any other language is refused. Leave empty to accept SQL only. *Since v26.12.1*	String	sqlscript,cypher,opencypher,gremlin,graphql,mongo
`postgres.maxParamSize`	Maximum size in bytes of one bind parameter value on the Postgres wire protocol. A value declaring a larger size is refused before any allocation	Integer	16777216
`postgres.quotedIdentifiers`	Reads double-quoted tokens of SQL statements received over the Postgres wire protocol as identifiers, as PostgreSQL and the SQL standard do. `false` restores the legacy behavior, where a double-quoted token is a string literal	Boolean	true
`redis.maxBulkLength`	Maximum length in bytes of one RESP bulk string accepted by the Redis protocol listener (512MB), the same as Redis `proto-max-bulk-len`	Integer	536870912
`redis.maxMultiBulkDepth`	Maximum nesting depth of a RESP array accepted by the Redis protocol listener	Integer	32
`redis.maxMultiBulkLength`	Maximum number of elements of one RESP array accepted by the Redis protocol listener, the same hard limit Redis applies	Integer	1048576
`redis.tls`	When `true`, the Redis protocol listener accepts only TLS connections, using the shared `ssl.*` key and trust store settings, so `AUTH` credentials are encrypted in transit	Boolean	false
`server.httpAuthSessionAbsoluteTimeout`	Absolute lifetime, in seconds, of an HTTP authentication session from its creation, whatever its activity. `0` disables it	Long	0

Name	Description	Type	Default Value
`server.httpQueryDefaultLimit`	Maximum number of rows the HTTP query and command endpoints return when the request sets no `limit` and the query has no `LIMIT`. A `LIMIT` in a SQL or openCypher query (the `LIMIT` of the final `RETURN`, or of every branch of a `UNION`) raises it; other languages need `limit` in the request. A truncated response carries `"truncated": true` (in the `stats` trailer of a streamed response) and the server logs a warning. `-1` or `0` means no default limit. A value above `server.httpQueryMaxResultRows` is lowered to it	Integer	20000
`server.httpQueryMaxResultRows`	Hard ceiling on the number of rows the HTTP query and command endpoints put in one response, whatever the request `limit`. A result that would exceed it fails the request with HTTP `413` naming this setting, instead of being silently truncated; a streamed (`Accept: application/x-ndjson`) result ends with an `error` line with `status 413` instead (see Stream query results). A request with no limit at all is truncated to `server.httpQueryDefaultLimit` instead. `-1` or `0` means unlimited (not recommended)	Integer	1000000
`server.httpStreamingReadTimeout`	Time, in milliseconds, an endpoint that reads the request body while it works (today only the bulk load `/api/v1/batch`) may spend between reads, for example while it commits, before the connection is closed. A client that stops sending is still cut off after `network.socketTimeout`. `0` disables the relaxation	Integer	600000
`server.httpWorkerThreads`	Maximum number of worker threads of the HTTP server that process blocking requests. Each thread reserves a stack, so a lower value reduces memory on small deployments. The `low-ram` profile sets it to 16	Integer	500
`server.restoreImportAllowLocalUrls`	Allows the `restore database` and `import database` server commands to read `file://` URLs and to fetch from private, loopback or link-local hosts. Disabled by default to prevent SSRF and local file reads through a client-supplied URL	Boolean	false
`server.security.importAllowedLocalPaths`	Comma-separated list of directories SQL `IMPORT DATABASE` may read local files from (`file://` and plain paths). Empty (default) applies no restriction. `classpath://` resources are always allowed	String	

Name	Description	Type	Default Value
`server.security.importBlockLocalNetworks`	When `true`, SQL `IMPORT DATABASE` refuses HTTP(S) URLs that resolve to loopback, link-local, private, wildcard or multicast addresses, to prevent SSRF against cloud metadata endpoints and internal services. Disable only in trusted environments	Boolean	true
`server.shutdownTimeout`	Milliseconds the JVM shutdown hook waits for the server lifecycle lock before letting the JVM exit without a graceful stop. It only matters when another thread holds the lock when the signal arrives (a stop in progress, or a start that already opened its databases). Giving up leaves the databases as a kill would, and the next open replays the WAL. Raise it if the shutdown of very large databases needs longer	Long	60000
`ssl.keyStore`	Path where the SSL certificates are stored	String	null
`ssl.keyStorePassword`	Password to open the SSL key store	String	null
`ssl.trustStore`	Path to the SSL trust store	String	null
`ssl.trustStorePassword`	Password to open the SSL trust store	String	null
`ssl.enabled`	Use SSL for client connections	Boolean	false
`postgres.debug`	Enables the printing of Postgres protocol to the console. Default is false	Boolean	false
`postgres.host`	TCP/IP host name used for incoming connections for Postgres plugin. Default is '0.0.0.0'	String	0.0.0.0
`postgres.port`	TCP/IP port number used for incoming connections for Postgres plugin. Default is 5432	Integer	5432
`postgres.ssl`	TLS mode for Postgres wire protocol connections, using the shared SSL key/trust store settings (`ssl.*`): DISABLED (no TLS, default), OPTIONAL (TLS or plain text, the client chooses: a client that falls back to plain text sends its password in the clear), REQUIRED (TLS only, plain text startup is refused; recommended once certificates are in place).	String	DISABLED

Name	Description	Type	Default Value
`postgres.queryMaxRows`	Maximum number of rows the result of one statement may occupy server-side before the first row is sent to a Postgres client, on both the simple-query and the extended-query (Parse/Bind/Describe/Execute) protocol. The server has to see the whole result before it can announce the column set, and the portal fetch size only bounds what each Execute sends, not what the server holds. A statement exceeding the limit is refused with an error; narrow it with a WHERE or LIMIT clause or raise the limit. An extended-protocol SELECT whose columns are known before its first row (a projection of declared properties, or a statement already described to the client) is fetched from a live cursor and is not subject to this limit *(since v26.11.1)*. 0 means unlimited.	Integer	1000000
`redis.defaultDatabase`	Default database name for Redis protocol connections. If set, RAM commands (SET, GET, etc.) will use this database's globalVariables. Empty means no default (requires SELECT command or key prefix)	String	
`redis.host`	TCP/IP host name used for incoming connections for Redis plugin. Default is '0.0.0.0'	String	0.0.0.0
`redis.port`	TCP/IP port number used for incoming connections for Redis plugin. Default is 6379	Integer	6379
`mongo.host`	TCP/IP host name used for incoming connections for Mongo plugin. Default is '0.0.0.0'	String	0.0.0.0
`mongo.port`	TCP/IP port number used for incoming connections for Mongo plugin. Default is 27017	Integer	27017
`server.databaseLoadAtStartup`	Open all the available databases at server startup. When `false`, a database is opened by the first request that names it, over HTTP too since v26.11.1 (see Databases opened on demand), so a server with many databases starts at once and holds open only the databases in use. Since v26.11.1 the database listings and existence checks of every protocol include a database that is on disk but not open yet	Boolean	true

Name	Description	Type	Default Value
`server.databaseIdleTimeout`	Closes a database no client has used for this long, in milliseconds. Used means any query, command, transaction, record operation, scan or count, from any protocol. A database with a transaction open, a statement running, asynchronous work queued or a backup, restore, import or export in progress is never closed, however long it has been quiet; an open connection without a transaction does not keep it open. The files stay where they are and the next request of any protocol reopens the database. A replicated (HA) database is never closed, nor one listed in `server.databaseIdleExclude`. `0` means never. See Closing idle databases. *(Since v26.11.1)*	Long	0
`server.databaseIdleCheckInterval`	How often, in milliseconds, the server looks for databases to close for idleness (`server.databaseIdleTimeout`) or beyond the cap on open databases (`server.maxOpenDatabases`). Idleness is measured at this resolution. See Closing idle databases. *(Since v26.11.1)*	Long	60000
`server.databaseIdleMaxClosesPerCheck`	The most databases one check closes, for idleness and for the cap on open databases together, so that a burst of databases going idle at once is closed over several checks. `0` means no bound. *(Since v26.11.1)*	Integer	10
`server.databaseIdleExclude`	Comma-separated names of the databases never closed for idleness or for the cap on open databases. *(Since v26.11.1)*	String	
`server.maxOpenDatabases`	The most databases kept open at once. Beyond it, each check closes the least recently used databases that are not in use, idle timeout or not, until the count is back under the cap. A database in use is never closed, so the cap can be exceeded while more databases than that are busy. The exclusions of `server.databaseIdleTimeout` apply. `0` means no cap. See Closing idle databases. *(Since v26.11.1)*	Integer	0
`server.databaseDirectory`	Directory containing the database	String	${arcadedb.server.rootPath}/databases
`server.backupDirectory`	Directory containing the backups	String	${arcadedb.server.rootPath}/backups

Name	Description	Type	Default Value
`server.configDirectory`	Directory the server reads and writes its configuration files in: `server-configuration.json`, the security files (`server-users.jsonl`, `server-groups.json`, `server-api-tokens.json`), `backup.json`, `ai.json`, `mcp-config.json` and `gremlin-server.yaml`. Set it as a `-D` system property or in `ARCADEDB_SETTINGS`, never in `server-configuration.json`, which lives inside it. Read once at startup; the directory is created if missing. `arcadedb-log.properties` is not read from here: logging is still configured with `-Djava.util.logging.config.file`. See Relocating the configuration directory.	String	${arcadedb.server.rootPath}/config
`instance.id`	Optional instance id (`adb-` followed by a lowercase UUID) to use instead of the one ArcadeDB generates and saves in the file `.instance.id` of the databases directory (older versions: `instance.id` of the configuration directory). Set it when no directory is writable or persistent. It identifies the instance to ArcadeData support and is never a credential. Empty means generated. See The instance id.	String	
`instance.derived`	Computes the instance id from `ha.clusterName` and `server.name` instead of saving it, so a server with no persistent directory (a container without a volume, a Kubernetes StatefulSet pod) keeps the same id on every restart. The names must be unique per node and stable. Ignored when `instance.id` is set. See The instance id.	Boolean	false
`support.url`	Base URL of the ArcadeData customer portal used by the Support tab of Studio and by `connect portal`. HTTPS only (plain HTTP is accepted for `localhost`). Overrides the value registered in `support.json`. See Support.	String	https://portal.arcadedb.com
`support.clientId`	Client ID (workspace id) of the customer portal. Together with `support.clientKey` it registers the server without Studio (containers, Kubernetes secrets) and overrides `support.json`. Empty means not set.	String	
`support.clientKey`	Client key (`wsk_…`) of the customer portal. A credential: masked when settings are listed or dumped, never returned by any API and never logged. Empty means not set.	String	

Name	Description	Type	Default Value
`support.autoRegister`	Registers the server as an installation of its workspace in the customer portal without anybody opening Studio: once shortly after the start of a server that holds a support key (`support.clientKey` or `support.json`), and then once a day. It sends the same redacted diagnostics Studio sends (never logs, never database contents) and only fills fields the portal has blank. Set it to `false` to register only from Studio or the console. See Support.	Boolean	true
`server.logsDirectory`	Directory where the server writes its log files. Useful on read-only root filesystems (e.g. Kubernetes `readOnlyRootFilesystem` pods) to relocate logs to a writable mount. The value is resolved very early at startup from, in order, the system property `arcadedb.server.logsDirectory`, the environment variable, then this setting, and supports `${…}` placeholders. The `server.sh`/`server.bat` scripts forward the `ARCADEDB_LOG_DIR` environment variable to this setting.	String	./log
`server.defaultDatabases`	The default databases created when the server starts. The format is `(<database-name>[(<user-name>:<user-passwd>[:<user-group>])][{import|restore:<URL>}][;]'. Pay attention on using `;` to separate databases and `,` to separate credentials. The supported actions are `import` and `restore`. A failing `import` or `restore` aborts server startup with an error. Only one `WITH` setting can be passed to `import`; a comma inside `{…}` starts a new command, not a second setting. Example: `Universe[albert:einstein:admin];Amiga[Jay:Miner,Jack:Tramiel]{import:/tmp/movies.tgz}`. A `:`, `,`, `;`, `[`, `]` or `%` inside a user name, password or group is written percent-encoded (`%3A`, `%2C`, `%3B`, `%5B`, `%5D`, `%25`): `mydb[myuser:my%3Apass:admin]` is the password `my:pass`. A credential with more than three `:` -separated fields is refused rather than truncated (since v26.12.1)	String	
`server.startupRestoreSlotWaitMs`	Milliseconds the `restore:` command of `server.defaultDatabases` waits for a backup, restore or import of the same database, already running on this server, to finish before it refuses. A refusal stops the server boot. `0` or a negative value refuses immediately. See Create default database(s).	Long	60000

Name	Description	Type	Default Value
`server.defaultDatabaseMode`	The default mode to load pre-existing databases. The value must match a com.arcadedb.engine.PaginatedFile.MODE enum value: {READ_ONLY, READ_WRITE}Databases which are newly created will always be opened READ_WRITE.	String	READ_WRITE
`server.httpIncomingHost`	TCP/IP host name used for incoming HTTP connections	String	0.0.0.0
`server.httpIncomingPort`	TCP/IP port number used for incoming HTTP connections. Specify a single port or a range `<from>-<to>`. Default is 2480-2489 to accept a range of ports in case they are occupied.	String	2480-2489
`server.httpsIncomingPort`	TCP/IP port number used for incoming HTTPS connections. Specify a single port or a range `<from>-<to>`. Default is 2490-2499 to accept a range of ports in case they are occupied.	String	2490-2499
`server.httpIoThreads`	Number of threads to use in the HTTP server. Called `server.httpsIoThreads` before 26.11.1, a name that is still read (and reported once) when the current one is not set.	Integer	2 per core
`server.httpSessionExpireTimeout`	Timeout in seconds for a HTTP session (managing a transaction) to expire	Long	5
`server.httpBodyContentMaxSize`	Maximum size in bytes for HTTP request body content, measured on the wire. Set to -1 for unlimited size. A request that declares a `Content-Encoding` is additionally limited by `server.httpBodyContentDecompressedMaxSize` once decoded. A body that declares no length (`Transfer-Encoding: chunked`, or an HTTP/2 request without `content-length`) is refused with `413` as soon as it crosses the limit. The server then reads and discards at most 4 MB more of it, so a body that ends there finishes cleanly and the connection stays usable; past that it stops reading: over HTTP/1.1 the connection is closed, over HTTP/2 only that stream is reset, with `NO_ERROR` *(since v26.11.1; before, the rest of such a body was read and discarded for as long as the client kept sending)*. Default is 100MB	Long	104857600

Name	Description	Type	Default Value
`server.httpBodyContentDecompressedMaxSize`	Maximum size in bytes an HTTP request body may expand to once its `Content-Encoding` has been decoded. `server.httpBodyContentMaxSize` limits only the bytes that arrive, so on the endpoints that decode a body - the InfluxDB line protocol ingest (gzip) and the Prometheus `remote_write` and `remote_read` endpoints (Snappy) - it would otherwise act as a compression-ratio multiplier rather than a limit. A body that decodes past this value is refused with `413` before the decoded bytes are allocated. Set to -1 (the default) to use the same value as `server.httpBodyContentMaxSize`, including its own -1 meaning unlimited; set to 0 for unlimited.	Long	-1
`server.httpTerminateOnClientDisconnect`	Terminates the statement of an HTTP request whose client closes the connection while it runs, so a client that gives up does not leave the server working on an answer nobody will read. Applies to plain HTTP/1.x connections once the request body has been read. A client that half-closes its side of the connection after sending the request and still waits for the response is taken for one that left: set to `false` for such clients. In a transaction session opened with `/begin`, a terminated command (`/command`, `/batch`) also rolls the session's transaction back and ends the session, as its failure would; a terminated read (`GET`/`POST /query`) or time-series, vector, PromQL or Grafana request leaves the session and its transaction as they were. While a watched statement runs, the connection's read timeout (`network.socketTimeout`) is suspended, so a statement longer than it still answers its client; the timeout is restored afterwards and still closes idle keep-alive connections. A client that talks HTTP/2 to the server (for example a JDK `HttpClient` left at its default version) is not covered: its connection is still closed one `network.socketTimeout` into a longer statement (issue #12606), so send long statements over HTTP/1.1. The statements a cluster replica forwards to the leader use HTTP/1.1 since v26.12.1. Since v26.10.2.	Boolean	true

Name	Description	Type	Default Value
server.httpContentSecurityPolicy	Content-Security-Policy sent with Studio's HTML pages and the Swagger UI page at /api/v1/docs. The default refuses inline script, so an HTML-injection bug in Studio cannot run script with the logged-in user's token; inline styles stay allowed. It also allows Google Fonts (fonts.googleapis.com, fonts.gstatic.com) for Studio's typeface and raw.githubusercontent.com for the list of sample datasets: an air-gapped or stricter installation can drop them, and Studio then falls back to a system font and reports that the sample datasets cannot be loaded. The Swagger UI page adds the hash of its bootstrap script to the directive that governs it, the first one the policy declares of script-src-elem, script-src and default-src (when only default-src is declared, a script-src with the default-src sources plus the hash is added). The policy is left as configured when that directive refuses every script ('none' or no source at all; a warning is logged once, and the page's script does not run) and when it already allows every inline script through 'unsafe-inline' with no hash or nonce beside it. Set to an empty string to send no policy. Since v26.10.2.	String	default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline' https://fonts.googleapis.com; font-src 'self' data: https://fonts.gstatic.com; img-src 'self' data: blob:; connect-src 'self' https://raw.githubusercontent.com; object-src 'none'; base-uri 'self'; frame-ancestors 'self'; form-action 'self'
server.httpContentSecurityPolicyReportOnly	Sends server.httpContentSecurityPolicy as Content-Security-Policy-Report-Only instead: the browser reports violations in its console but blocks nothing. Use it to try a custom policy before enforcing it. Since v26.10.2.	Boolean	false
server.httpStreamingWriteTimeout	Budget in ms a single blocking write of a streamed HTTP response (/batch results, Accept: application/x-ndjson on /query and /command, Server-Sent Events of AI chat and long-running server commands) may make no progress. When a client stops reading, the connection is closed and the worker thread released. The timer is re-armed for each chunk of at most 64 KB, so a slow but reading client never trips it. 0 or a negative value leaves streamed writes unbounded.	Integer	60000

Name	Description	Type	Default Value
`server.httpStreamingKeepAliveInterval`	Interval in ms after which a streamed answer (`Accept: application/x-ndjson` on `/query`, `/command` and `/batch`) that has had nothing to send writes a bare newline, which consumers skip. Keeps a client or proxy that bounds silence (the Java remote client does) from failing a healthy slow query. Keep it well below the smallest silence budget of clients and proxies. `0` or a negative value sends no keep-alive.	Integer	5000
`server.httpAuthSessionExpireTimeout`	Timeout in seconds for a HTTP authentication session to expire. This timeout is computed from the latest request using the auth token. See Token-Based Authentication.	Long	1800
`server.httpAuthSessionMax`	Maximum number of concurrent HTTP authentication sessions the server keeps in memory. Once reached, a further `POST /api/v1/login` is answered `503` instead of growing the session map without bound; expired sessions are reclaimed before refusing. Set to `0` for unlimited (not recommended).	Integer	10000
`server.httpAuthSessionMaxPerUser`	Maximum number of concurrent HTTP authentication sessions a single user may hold. Beyond it, that user's oldest session is invalidated to make room for the new one, so a login loop recycles only its own sessions and never affects other users. Set to `0` for unlimited (not recommended).	Integer	100
`server.mode`	Server mode between 'development', 'test' and 'production'	String	development
`server.name`	Server name	String	ArcadeDB_0
`server.plugins`	Server plugins to load, see available plugins. Format as comma separated list of: `<pluginName>:<pluginFullClass>`.	String	

Name	Description	Type	Default Value
`server.queryWarmUp`	Warms the query stack up in the background while the server starts, so the first request of a client does not pay for it: on a low-priority daemon thread started in parallel with the load of the databases, it hashes a throwaway password the way the first login does (nothing is cached), parses a SQL statement, then, once the server is online, runs `SELECT 1` in SQL and `RETURN 1` in OpenCypher on a database the server holds open and serializes the rows to JSON. The first SQL request over HTTP sent right after a fresh start went from about 160 ms to about 60 ms (measured with curl; what is left is mostly the hash of the real user's password). With no database open it skips the statements: it never opens a database. Set to `false` to skip it *(since v26.11.1, issue #10218)*	Boolean	true
`server.rootPassword`	Password for root user to use at first startup of the server. Set this to avoid asking the password to the user	String	null
`server.rootPasswordPath`	Path to file with password for root user to use at first startup of the server. Set this to avoid asking the password to the user	String	null
`server.rootPath`	Root path in the file system where the server is looking for files. By default is the current directory	String	null
`server.securityAlgorithm`	Default encryption algorithm used for passwords hashing	String	PBKDF2WithHmacSHA256
`server.reloadEvery`	Time in milliseconds of checking if the server security files have been modified to be reloaded	Integer	5000
`server.securitySaltCacheSize`	Cache size of hashed salt passwords. The cache works as LRU. Use 0 to disable the cache	Integer	64
`server.apiTokenRequireSecureTransport`	When true, `POST /api/v1/server/api-tokens` refuses to mint an API token unless the request arrived over HTTPS, from a loopback peer, or through a reverse proxy listed in `server.apiTokenTrustedProxies` that reports `https`. A refused mint is answered `412`. When false, an unprotected mint goes ahead and is logged at WARNING. **The default is scheduled to change to `true` in 27.1.1**; see Create an API token.	Boolean	false

Name	Description	Type	Default Value
`server.apiTokenTrustedProxies`	Comma-separated literal IP addresses or CIDR ranges (IPv4 and IPv6) of the reverse proxies allowed to vouch, through `X-Forwarded-Proto` or the RFC 7239 `Forwarded` header, that an API token mint reached them over HTTPS. Applies to the HTTP route and to the gRPC `CreateApiToken` RPC. Headers from any other peer are ignored. Hostnames are rejected, not resolved, and a list that does not parse is treated as empty and logged at SEVERE. List only an L7 proxy that sets or overwrites the header, never an L4 balancer that passes the client's headers through. Empty trusts no proxy.	String	
`server.saltIterations`	Number of iterations to generate the salt or user password. Changing this setting does not affect stored passwords	Integer	65536
`server.eventBusQueueSize`	Size of the queue used as a buffer for unserviced database change events.	Integer	1000
`server.eventBusMaxPendingBytes`	Maximum number of bytes of change-stream frames that may be outstanding towards a single WebSocket subscriber before it is evicted. Frames are sent asynchronously, so a subscriber that never reads accumulates them in the server's send buffer: the producer-side queue is bounded but a slow consumer is charged to the server's heap, not to its own. Past this cap the subscription is dropped and the channel closed, which is what the client would experience anyway. 0 disables the cap.	Long	16777216
`server.wsMaxPendingControlBytes`	Maximum number of bytes of WebSocket request-answer frames - a subscription acknowledgement/error, or an insert-session `started`/`batchAck`/`committed`/error - that may be outstanding towards a single `/ws` connection before it is closed. This bounds the answer side of the connection; `server.eventBusMaxPendingBytes` bounds only the change-stream push frames a client subscribed to. 0 disables the cap.	Long	16777216
`server.wsInsertSessionExpireTimeout`	Timeout in seconds for a `/ws` duplex insert session to expire, counted from the latest frame received on it. An expired session is rolled back and the client is told with an unsolicited error frame.	Long	60

Name	Description	Type	Default Value
`server.wsMaxControlFrameSize`	Maximum size in bytes of a single text frame accepted on `/ws` before an insert session is started on that connection, and of every binary frame. A larger frame closes the connection with `1009 TOO_BIG` as soon as it crosses the cap. `0` or a negative value restores the unbounded behavior.	Long	65536
`server.wsMaxInsertFrameSize`	Maximum size in bytes of a single text frame accepted on a `/ws` connection while it has a duplex insert session open. Up to 64 frames may wait per connection, so the per-connection buffering to budget for is this value times 64. `0` or a negative value restores the unbounded behavior.	Long	16777216
`server.wsMaxInsertChunkRows`	Maximum number of records a single `/ws chunk` frame may carry. A chunk over the cap is refused with an error frame naming the limit and the session stays open, so the client can split its batch and carry on. `0` or a negative value disables the cap.	Integer	100000
`server.healthCheck.enabled`	Run the server health monitor: one background thread that samples free disk space on the filesystem holding the databases, available heap, and JVM pause times, and writes a WARNING to the server event log when one of them degrades. Each warning is rate limited - 24h for low disk, 30 minutes for heap and JVM pauses - so a server that stays degraded reports it periodically rather than on every 10-second sample. Read at server start.	Boolean	true
`server.readinessRequiresHA`	When true and HA is active, `/api/v1/ready` also requires the node to have joined the Raft group (a leader has been elected). Default false preserves current readiness behavior. See Health probes.	Boolean	false
`server.readinessHAMaxLag`	When `server.readinessRequiresHA` is true, the maximum number of Raft log entries a follower may lag behind the commit index (`commitIndex - lastAppliedIndex`) and still report Ready. Keeps `/api/v1/ready` returning 503 until a (re)joined follower has replayed the committed log, so a rolling restart does not drop the write quorum. See Health probes.	Long	100

Name	Description	Type	Default Value
server.logFormat	Console log format: text (default, human-readable) or json (one JSON object per line with correlation fields). Can be set from the server configuration file, SET SERVER SETTING or as a -D JVM property; an explicit -D wins over the configuration file. See Structured logging. *(Honoured outside -D)*	String	text
server.logIncludeTrace	In text log mode, append [traceId=…] to each line while a trace is active. Default false preserves current text output.	Boolean	false
server.grpcQueryMaxResultRows	Hard ceiling on the number of rows the gRPC unary ExecuteQuery materializes. A request limit at or below this cap is honored; a result that would exceed it fails the call with RESOURCE_EXHAUSTED (consistent with the StreamQuery MATERIALIZE_ALL path) rather than silently truncating, and a client cannot bypass it with a larger limit. It bounds the unary ExecuteCommand the same way (the remote schema reload and every command() call of the gRPC client go through it): this ceiling is the default when the request carries no max_rows, and a larger max_rows cannot bypass it. Bounds heap usage and protects against limitless-query denial-of-service. The default is lower than server.grpcStreamMaxMaterializedRows because the unary response is built and returned as a single gRPC message (also bounded by the max message size), whereas StreamQuery emits incrementally. Set to -1 or 0 for unlimited (removes the DoS protection).	Integer	100000
server.grpcStreamMaxMaterializedRows	Maximum number of rows the gRPC StreamQuery MATERIALIZE_ALL retrieval mode buffers in memory before emitting. Exceeding the cap fails the call with RESOURCE_EXHAUSTED so clients fall back to CURSOR/PAGED streaming instead of running the server out of memory. Set to -1 or 0 for unlimited (removes the DoS protection).	Integer	1000000

Name	Description	Type	Default Value
`server.grpcTimeSeriesMaxResultRows`	Hard ceiling on the number of rows - or aggregation buckets - one gRPC `TimeSeriesQuery` answers with. A request limit at or below this cap is honored; a request limit above it is refused **before the first message** with `RESOURCE_EXHAUSTED`, so a client is never handed a partial series it cannot tell apart from a complete one. Separate from `server.grpcQueryMaxResultRows` because `TimeSeriesQuery` is a server-streaming RPC, so the single-message rationale that keeps the unary ceiling low does not apply to it. The default matches the HTTP twin `POST /api/v1/ts/{database}/query` (`server.httpQueryMaxResultRows`), so the same range is served over both protocols. Set to `-1` or `0` for unlimited (the aggregated path materializes every bucket before emitting any, so this removes its DoS protection; the raw path stays lazy either way).	Integer	1000000
`server.grpcStreamWriteTimeoutMs`	Maximum time in milliseconds a gRPC `StreamQuery` worker waits for the client transport to become ready to accept the next batch before aborting the stream. Prevents a slow or abandoned client from pinning the worker thread (and the open `ResultSet`/transaction) indefinitely. Set to `-1` to wait forever (removes the DoS protection).	Long	60000
`serverMetrics`	True to enable metrics	Boolean	true
`serverMetrics.logging`	True to enable metrics logging	Boolean	false
`serverMetrics.prometheus.requireAuthentication`	Require authentication on the `/prometheus` scrape endpoint exposed by the Prometheus metrics plugin (plugin-level key, read by the optional `metrics` module). The endpoint reads this on every scrape, so a change through `SET SERVER SETTING` takes effect at once; before that it kept the value it was started with until the server was restarted.	Boolean	true
`serverMetrics.otlp.enabled`	Register an OTLP metrics registry alongside the `/prometheus` scrape, pushing metrics to `serverMetrics.otlp.endpoint`. Requires `serverMetrics=true`. Plugin-level key read by the optional `metrics` module. See Metrics depth.	Boolean	false
`serverMetrics.otlp.endpoint`	OTLP metrics export endpoint, used when `serverMetrics.otlp.enabled=true`. Metrics use OTLP/HTTP (port 4318, path `/v1/metrics`), not gRPC; a URL without a path gets `/v1/metrics` appended.	String	http://localhost:4318/v1/metrics

Name	Description	Type	Default Value
`serverMetrics.otlp.step`	How often metrics are pushed to `serverMetrics.otlp.endpoint`, in milliseconds. Also the window rates and maxima are measured over. Not positive keeps the default of one minute; a positive value below 1000 is raised to 1000. Read when the metrics plugin starts.	Long	60000
`serverMetrics.serviceName`	The OpenTelemetry `service.name` resource attribute reported by the tracing and OTLP metrics plugins. `OTEL_SERVICE_NAME`, then a `service.name` entry in `OTEL_RESOURCE_ATTRIBUTES`, take precedence. Read when the plugins start.	String	arcadedb
`serverMetrics.tracing.enabled`	Enable OpenTelemetry distributed tracing (requires the optional `tracing` plugin on the classpath, shipped in the `full` distribution). Query/command spans include the statement text as the `db.statement` span attribute, which may contain sensitive data, so secure the OTLP collector endpoint. See Distributed tracing.	Boolean	false
`serverMetrics.tracing.endpoint`	OTLP trace export endpoint (gRPC).	String	http://localhost:4317
`serverMetrics.tracing.samplingRate`	Parent-based trace sampling ratio in [0.0,1.0]. `1.0` samples everything, `0.0` disables sampling.	Float	0.0
`serverMetrics.tracing.excludedPaths`	Comma-separated HTTP request paths that never produce a trace span, matched exactly (query string excluded, one trailing slash ignored, no wildcards). The default leaves out the readiness and health probes. The `arcadedb.http.requests` timer still counts them. An empty string traces every request. Read when the tracing plugin starts.	String	/api/v1/ready, /api/v1/health
`studio.enabled`	Force-enable the Studio web tool even when the server runs in `production` mode. In `development` and `test` mode Studio is always served; in `production` mode it is disabled by default and this setting can re-enable it. See how-to/operations/server.pdf	Boolean	false

DATABASE

Name	Description	Type	Default Value
`asyncCloseTimeout`	Milliseconds to wait for running asynchronous tasks to finish when a database is closed or dropped, before forcing the async workers down with a WARNING. Tasks that never ran are not applied. `0` waits forever	Long	60000
`asyncOperationsQueueImpl`	Queue implementation to use between 'standard' and 'fast'. 'standard' consumes less CPU than the 'fast' implementation, but it could be slower with high loads	String	standard
`asyncOperationsQueueSize`	Size of the total asynchronous operation queues (it is divided by the number of parallel threads in the pool)	Integer	1024
`asyncBackPressure`	When the asynchronous queue is full at a certain percentage, back pressure is applied	Integer	0
`asyncTxBatchSize`	Maximum number of operations to commit in batch by async thread	Integer	10240
`asyncWorkerThreads`	Number of asynchronous worker threads. By default it is cores minus 1, but at least 1	Integer	(machine dependent)
`backup.compressionLevel`	Deflate level of a full backup, from `0` (stored) to `9` (smallest). The backup is CPU bound, and writers are throttled while it runs, so a faster level means a shorter stall. Measured on a 1.25 GB database, level 1 is 3.1x faster than level 9 for a 7.5% larger archive	Integer	1
`backup.compressionThreads`	Number of threads that compress a full backup in parallel, also within one file. `-1` (default) uses half the available cores, at most 8; `0` selects the legacy single-threaded writer. The archive is a standard ZIP either way	Integer	-1
`backup.maxMBPerSecond`	Cap, in MB/s, on the rate a full backup reads the database files, so it cannot saturate the disk. A slower backup keeps writers throttled for longer. `0` means no limit	Integer	0
`checkDatabaseAdjacencyCacheEntries`	Maximum number of adjacency-list entries `CHECK DATABASE` keeps in memory while it checks back references, so the edge list of a super node is walked once instead of once per edge (about 20MB at the default). The least recently probed list is evicted first. `0` disables the cache	Integer	1000000

Name	Description	Type	Default Value
`checkDatabaseRepairBatchPages`	Number of modified pages `CHECK DATABASE … FIX` accumulates before it commits its repair and opens the next transaction, so an HA Raft entry stays below the replicated entry limit. Also bounds `CHECK DATABASE … COMPRESS`. A soft limit, checked between units of work. `0` disables batching: the whole repair is one transaction	Integer	256
`command.regexTimeout`	Maximum time in milliseconds a regular expression evaluation may run before it is aborted, against catastrophic backtracking. Covers SQL `MATCHES`, `LIKE` and `ILIKE`, openCypher `=~`, `text.regexReplace()`, `text.replace()`, `text.split()`, `.normalize()`, full-text regexp and wildcard queries, PromQL `=~`/`!~` and `REGEXP` property validation. One deadline is shared by a whole command (or by a whole scan outside a command), not one per row. Applies even when `command.timeout` is `0`. A pattern that could backtrack without reading any character of its input, so that no deadline could stop it (for example many empty alternatives or optional anchors in a row, a counted loop of an empty group nested a few levels deep, or a look-behind with a very long range), is refused before it runs with the same timeout error, when it can succeed in more than 1,024 different ways without a read, when one attempt makes more than about 65,000 steps without one, or, for a search or a replace-all, when the steps one start position makes without a read times the input length exceed about 130 million. Rewrite such a pattern without the empty alternatives or optional anchors. `0` disables it, together with this check (not recommended)	Long	1000
`externalPropertyBucketDefaultPageSize`	Default page size in bytes of the paired buckets that hold `EXTERNAL` property values (embeddings, large strings, JSON). Larger than the primary bucket default to reduce multi-page records. Minimum is 1048 bytes *(since v26.11.1)*. See External property storage	Integer	262144

Name	Description	Type	Default Value
`externalPropertyBucketPath`	Directory where new paired external-property buckets are created. Empty (default) keeps them next to the primary buckets. Set it to cheaper or slower storage to tier the heavy values away from the rest of the database. The directory must exist and be writable; existing buckets are not moved. Since v26.10.2 an `ALTER DATABASE` of it applies on every node of an HA cluster, so the path must exist on all of them. Since v26.12.1 a value set for one database must lie inside the database directory or inside the directory the server-level value names, and `ALTER DATABASE` of it needs `updateSecurity` (see Tiered storage)	String	
`flushAllPagesTimeout`	Milliseconds without flush progress after which waiting for all the pages of a database to reach the disk (on close, rename, or when a backup suspends flushing) gives up with a SEVERE log instead of hanging on a wedged flush. The window restarts whenever the number of pending pages goes down, so a slow but healthy backlog never trips it. A close that gives up keeps the WAL files and the lock file, so the next open runs recovery and replays the unflushed pages. `0` waits forever	Long	60000
`flushSuspendMaxDeferredRAM`	Maximum RAM (in MB) of dirty pages kept in memory while page flushing is suspended (HA snapshot ship or full backup). Past it, the committing threads of the suspended databases are throttled; other databases are not affected. `0` disables the cap	Long	512
`gavCommitOrderMaxWait`	Milliseconds a commit on a `SYNCHRONOUS` or `ASYNCHRONOUS` Graph Analytical View waits, once it is durable, for the commits ahead of it to reach the view, so that its own change is in the view (or counted as pending) when `commit()` returns. The changes always reach the view in the order the transactions committed; this only bounds the wait. A transaction that never reports (its thread died between its commit point and its callback) costs the commits behind it one such wait, logged as a WARNING, and is then skipped. `0` does not wait. *Since v26.11.1*	Long	30000
`gavUseWhenStale`	When `true`, the query planner uses a stale Graph Analytical View for traversals instead of falling back to OLTP. Faster, but it may not reflect the latest committed changes	Boolean	false

Name	Description	Type	Default Value
`graph.supernodeInterleaveRounds`	Number of round-robin rounds a read of a super node's edges takes one entry per stripe before taking larger batches from each stripe, which keeps the approximate newest-first order while reducing the cost of a full walk. `0` concatenates the stripes without interleaving	Integer	64
`graphql.maxNestingDepth`	Maximum nesting depth of `{}` and `[]` in one GraphQL document. A deeper document is refused as a parse error instead of exhausting the thread stack	Integer	200
`index.buildChunkSizeMB`	Size in MB of the transaction chunks of a bulk index build with the WAL disabled. Larger chunks reduce commit overhead and use more memory	Long	50
`index.buildSortedNonUniqueMinRecords`	Minimum number of records (subtypes included) a type must hold for a `NOTUNIQUE` LSM index created with the default build mode to use the sorted build instead of adding records one at a time, which pays the transaction bookkeeping of every key. When the keys fit the memory budget of the sorted build it takes about two thirds of the time. A build on a replicated database or inside a transaction with pending changes always adds records one at a time. `0` turns the switch off. *Since v26.11.1*	Long	1000000
`index.buildSortedUniqueMinRecords`	Minimum number of records (subtypes included) a type must hold for a `UNIQUE` LSM index created with the default build mode to use the sorted build instead of adding records one at a time, whose duplicate check made the build time grow with the square of the record count. A build on a replicated database or inside a transaction with pending writes to the buckets it indexes always adds records one at a time, but from this many records on it no longer checks every key against the index it is filling: it checks the keys once, over the finished index, and refuses the first duplicate then, so its time grows with the record count, and a duplicate costs the whole build before it is reported. `0` turns the switch off, and that deferred check with it. *Since v26.11.1*	Long	10000

Name	Description	Type	Default Value
`indexBloomFilterRate`	Target false-positive rate of the bloom filter an index compaction writes for each compacted series of an LSM index, so a point lookup skips a series that cannot hold the key. Helps most when the series overlap in key range (UUIDs, emails, a bulk load into a `UNIQUE` index); range scans never use it. Costs about 1.2 bytes per key on disk at 1%, in an `<index>_bf.bfidx` file. Existing indexes get filters at their next compaction. `0` disables it	Float	0.01
`indexCompactionFullSeriesThreshold`	Number of compacted series at which an index compaction becomes a full compaction, merging every series and the mutable pages into one and dropping deleted entries. A full compaction can also start earlier, see `indexCompactionFullDeadEntriesRatio`. `0` disables it	Integer	10
`indexCompactionFullDeadEntriesRatio`	Starts a full compaction of an LSM index as soon as the deleted keys its range scans have skipped since the last full compaction reach this multiple of the entries of its compacted part, whatever the series count. Incremental compactions keep the deleted entries, so under a sliding window (insert the newest ids, `DELETE ⋯ WHERE id < ?` the oldest) every scan from the low end used to walk every key deleted since the last full compaction until `indexCompactionFullSeriesThreshold` was reached. An index whose deleted keys no scan crosses is left to the series threshold. In a cluster the leader compacts, so the scans counted are the ones the leader served. Watch `deadEntriesSkippedSinceFullCompaction` in the index statistics. `0` disables it. *Since v26.11.1*	Float	1
`indexDefaultPageSize`	Default page size in bytes for new plain LSM-tree indexes (not full-text, geo or vector ones) created without an explicit page size (SQL `CREATE INDEX` has no page-size clause). A transaction copies every page it changes, so smaller pages make small write transactions cheaper while larger ones make lookups slightly faster. Existing indexes keep their page size. Minimum is 8192. In a cluster, keep it identical on all the servers.	Integer	262144

Name	Description	Type	Default Value
`bucketDefaultPageSize`	Default page size in bytes for buckets. Existing buckets keep their page size. Minimum is 8216 bytes, the room a page needs for its header, its slot table and one record: `ALTER DATABASE` refuses a smaller value, and so does the creation of a bucket (`CREATE TYPE … PAGESIZE` included) with one, naming the setting *(since v26.11.1; before, the value was accepted and every insert into a type created afterwards failed)*. Default is 65536	Integer	65536
`bucketSelfDescribingChunkLinks`	Store the next-chunk link of a multi-page record as `-(position + 1)` instead of the absolute position, which lets a continuation chunk live at position 0 of a bucket. With `false` (default) such a chunk is refused with a `DatabaseOperationException`. A database that has stored the new form cannot be read by a version older than 26.11.1: turn it on only when no node or restore target runs an older version. In HA set it on every node, it is not gated. Links already written keep their form until their chunk is rewritten. *Since v26.11.1*	Boolean	false
`bucketReuseSpaceMode`	Mode used to reuse space in pages. Use 'low' to have faster updates consuming more space on disk, `medium` for balance. Default is 'high'. Applies per database, so `ALTER DATABASE` on one database does not affect the others. With 'high', a write that frees space lists the page for reuse at once; a background scan looks for the room nothing listed (at most 100 pages are listed at a time) and reads at most 1,024 pages every 5 seconds, so on a very large bucket that room can take hours to be found, while the bucket keeps appending *(since v26.11.1; before, a scan read the whole bucket in one go, stalling the writer for minutes on a file of millions of pages)*. Room a write freed while the list was full is looked for as soon as the listed pages run out, without waiting for the 5 seconds, so an insert right after a bulk delete fills the pages the delete emptied instead of growing the file *(since v26.11.1)*. The edges created through `database.batch()` reuse that room too, and the room left on the last page of the bucket, before they open a new page *(since v26.11.1; before, every batch flush started a new page, so a workload of small batches and deletes grew the edge bucket by one page per batch)*	String	high

Name	Description	Type	Default Value
`bucketWipeOutOnDelete`	Wipe out record content on delete. If enabled, assures deleted records cannot be analyzed by parsing the raw files and backups will be more compressed, but it also makes deletes a little bit slower	Boolean	true
`command.timeout`	Default timeout for commands (in ms). On a cluster, a write a follower forwards to the leader carries the follower database's budget, and the leader enforces that budget in place of its own setting.	Long	0
`command.warningsEvery`	Reduce warnings in commands to print in console only every X occurrences. Use 0 to disable warnings with commands	Integer	100
`commitLockTimeout`	Timeout in ms to lock resources during commit. A bulk index build uses `index.buildCommitLockTimeout` instead for its own commits	Long	5000
`explicitLockTimeout`	Timeout in ms to acquire the resources of an explicit `LOCK`. Separate from `commitLockTimeout`, which bounds the locking the engine does for you at commit time	Long	5000
`cypher.algoMaxWorkingMemory`	Maximum heap, in bytes, that a single `algo.*` procedure call may reserve for its working set: the graph it loads without a view (the RID of every vertex, 16 bytes each, since v26.12.1), the adjacency lists, weighted adjacency lists and degree buffers, the edge arrays of `algo.mst`/`algo.msa`, the visited sets and traversal state of the walks, the neighbour bitsets of the clique, triangle and truss algorithms, walk buffers, label memory, embedding and feature matrices, `nodeCount x nodeCount` distance/similarity/capacity matrices, and terminal tables and terminal-pair arrays. A call over budget is refused before allocating, naming the offending component. Negative means no limit, a positive value is a fixed budget. 0 (the default since v26.11.1) sizes the budget at each call from the heap the call can get: half of the heap free after the last collection plus the page cache, which the call takes back from the cache before it allocates, and never less than 1/8 of the max heap (64MB at least), the fixed default it replaces. Before v26.11.1 the default was that fixed 1/8, which on a 2GB heap refused `algo.wcc()` on a graph of 34M edges whose 280MB adjacency fitted (issue #11011).	Long	0 (sized from the heap)

Name	Description	Type	Default Value
`opencypher.bulkCreateBatchSize`	Batch size of bulk `CREATE` after an `UNWIND`: the records are created in batches to reduce transaction overhead. `0` disables batching	Integer	20000
`opencypher.foreachEagerRead`	Runs `FOREACH` to completion before a later clause of the same query reads the graph (`MATCH`, `MERGE`, `CALL`), so every row sees the complete result of the `FOREACH`. The input rows are kept in memory meanwhile. `false` restores the streaming behavior for very large inputs, at the price of the later read seeing a partly applied `FOREACH`	Boolean	true
`opencypher.labelWriteDegreeLimit`	Maximum degree of a vertex a Cypher label change may rewrite; above it the command fails. `0` (default) disables the limit	Integer	0
`opencypher.labelWriteDegreeWarning`	Degree above which a Cypher label change (`SET n:Label`, `REMOVE n:Label`) logs a warning. A label change rewrites the vertex under its new type and re-creates every edge, so it costs O(degree) and gives the vertex and its edges new RIDs. `0` disables the warning	Integer	10000
`opencypher.loadCsv.allowRemoteUrls`	Allows `LOAD CSV` to read `http://` and `https://` URLs, still restricted by `opencypher.loadCsv.blockedIpRanges`. Since v26.10.2 changing it through `ALTER DATABASE` needs the `updateSecurity` permission	Boolean	true
`opencypher.loadCsv.blockedIpRanges`	Comma-separated CIDR ranges `LOAD CSV` remote fetches may not reach, checked on the resolved address and on every redirect. The default covers loopback, private, link-local (including `169.254.169.254`), carrier-grade NAT, multicast and reserved ranges. Empty disables the filter (not recommended). Since v26.10.2 changing it through `ALTER DATABASE` needs the `updateSecurity` permission	String	(loopback, private, link-local, reserved)
`opencypher.planCache`	Maximum number of OpenCypher execution plans to keep in cache (frequency-based eviction). Since v26.11.1 a change on an open database applies at once	Integer	300

Name	Description	Type	Default Value
`opencypher.statementCache`	Maximum number of parsed OpenCypher statements to keep in cache. A statement must be hit twice to be protected from a burst of one-off query texts. Queries that differ only in the values they embed share one entry (see `queryLiteralParameterization`, *since v26.10.2*), so raise this value only if you generate many queries of different shapes. This is the setting that sizes Cypher statement caching. A statement that holds more than 4,096 literal values, typically a bulk write with its data written in the text (`INSERT INTO T CONTENT [⋯]`, `UNWIND [⋯] AS row CREATE ⋯`), is parsed for its execution and not cached, so it does not keep its parse tree on the heap *(since v26.12.1)*. Since v26.11.1 a change on an open database applies at once	Integer	300
`opencypher.loadCsv.allowFileUrls`	Allow LOAD CSV to access local files via `file:///` URLs and bare file paths. Disable for security in multi-tenant server deployments. In `production` server mode, this is automatically set to `false` if not explicitly configured. See how-to/operations/server.pdf	Boolean	true
`opencypher.loadCsv.importDirectory`	Root directory for LOAD CSV `file:///` URLs. When set, file paths are resolved relative to this directory and path traversal (`../`) is blocked. Empty string means no restriction	String	(empty)
`dateFormat`	Default date format using Java SimpleDateFormat syntax. Textual fields (month or day names, e.g. `MMM`, `EEE`) always use the English locale, whatever the JVM locale of the server, so every node of a cluster writes and reads the same value	String	yyyy-MM-dd
`dateImplementation`	Default date implementation to use on deserialization. By default java.time.LocalDate is used, but the following are supported: java.util.Calendar, java.util.Date, java.time.LocalDateTime	Class	java.time.LocalDate
`dateTimeFormat`	Default date time format using Java SimpleDateFormat syntax. Textual fields always use the English locale, see `dateFormat`	String	yyyy-MM-dd HH:mm:ss

Name	Description	Type	Default Value
`dateTimeImplementation`	Default datetime implementation to use on deserialization. By default java.time.LocalDateTime is used, but the following are supported: java.util.Date, java.util.Calendar, java.time.LocalDateTime, java.time.ZonedDateTime, java.time.Instant. java.util.Date and java.util.Calendar cannot carry sub-millisecond precision, so with them DATETIME_MICROS and DATETIME_NANOS values are returned as java.time.LocalDateTime	Class	class java.time.LocalDateTime
`deleteTolerateBrokenChain`	When true, a `DELETE` of a record whose multi-page chunk chain is structurally corrupted removes the record anyway, skipping the index cleanup it cannot read and leaving any unreachable chunks for `CHECK DATABASE FIX` to reclaim. When false (the default) such a delete is refused with a `BrokenChunkChainException` naming the record, so corruption is never removed silently. Genuine transaction conflicts are unaffected by this setting: they keep retrying either way. `CHECK DATABASE FIX` removes a corrupted record regardless of this setting	Boolean	false
`serializer.fixedWidthFloatArrays`	Write float and double arrays (vector embeddings) in their fixed-width form: 4 bytes per float and 8 per double, against 5 and 9-10 in the varint form used before. A build before v26.11.1 cannot read the fixed-width form, so set it to `false` to keep the database files readable by such a build if you may have to roll back. Read when the database opens; records already written keep their form until they are updated. When it is `true`, a replicated database writes the fixed-width form only once every server of the cluster reads it. Since v26.12.1, while it is `true` a database opened read-write has its dictionary labelled format version 2, which builds from v26.8.1 to v26.11.x refuse to open (a database without float arrays included): set it to `false` before the database is first opened read-write by v26.12.1 to keep a rollback possible, since turning it off later does not remove the label. See Upgrade. *Since v26.11.1*	Boolean	true
`freePageRAM`	Percentage (0-100) of memory to free when Page RAM is full	Integer	50

Name	Description	Type	Default Value
`gavCchMaxArcsPerEdge`	Upper bound on the size of a Customizable Contraction Hierarchy (CCH) attached to a Graph Analytical View, as supergraph arcs (edges plus shortcuts) per edge of the routed graph. Road, logistics and utility networks need a small multiple of their edge count; graphs without small separators (social graphs, graphs with supernodes) need far more, and building them would exhaust the heap. A hierarchy that would exceed the bound is not built: the view reports it as `UNSUITABLE` and shortest-path queries keep answering through bidirectional Dijkstra. Graphs small enough to need fewer than 100,000 arcs are always accepted. Each arc costs about 52 bytes of heap (topology plus the directed metric) and 20 more once undirected (`BOTH`) queries are served too, so the worst case at the default is about 830 bytes, or 1.2 KB, per routed edge. See Shortest Paths with Contraction Hierarchies	Integer	16
`gavPersistCsr`	When true, a Graph Analytical View (GAV/CSR) that is READY (with no pending overlay changes) when the database closes cleanly writes its CSR to disk alongside a freshness certificate (the database's last committed transaction id at build time). If nothing was committed to the database between that close and the next open, the certificate still matches and the persisted CSR is reused as-is instead of being rebuilt by a full graph scan. Any commit in between invalidates the certificate and falls back to the previous behavior: an async rebuild triggered on open. Set to false to disable persisting the CSR file (e.g. to avoid its disk footprint or the extra write at close). See Graph OLAP Engine.	Boolean	true
`gavBuildReleasesPages`	When true, the pages a Graph Analytical View build reads from disk leave the page cache when the build ends, unless another reader used them meanwhile: the view answers what they hold, and the page cache is on the heap. While the build runs they are the first pages evicted. Pages the cache held before the build stay. Set to false to keep them, for a workload that reads the same records through the record API right after the build. See Sizing the Heap for a View. *Since v26.11.1*	Boolean	true

Name	Description	Type	Default Value
`gavRestoreAwaitTimeout`	Milliseconds `database.open()` blocks waiting for Graph Analytical Views (GAV/CSR) restored from persisted definitions to reach READY before returning. 0 (default) does not wait: when no persisted CSR plausibly applies, the full rebuild is still triggered immediately in the background, but `open()` returns before it completes and queries issued right after run unaccelerated until it does; when a persisted CSR does plausibly apply (see `gavPersistCsr`), nothing is even started at `open()` time - it is deferred to whichever query touches the view first (see Lazy Restore), so the fast path costs nothing for a session that never queries it. A positive value here forces the wait either way, trading a slower `open()` for the restored/rebuilt view being usable by the query that triggered the reopen. See Graph OLAP Engine	Long	0
`gavQueryRestoreAwaitTimeout`	Milliseconds a query waits, while it is planned or starts, for a Graph Analytical View (GAV/CSR) whose deferred restore from disk is still in flight, so the first count push-down, `MATCH`, shortest path or other graph function after a reopen runs on the restored view instead of reading every record. Covers the SQL and OpenCypher planners and the SQL graph functions; the `algo.*` procedures have their own budget (`gavAlgoRestoreAwaitTimeout`). The wait ends early when the restore finishes, and a view that is only rebuilding after a commit is never waited for. `0` does not wait: the call takes the record path. The wait is not tied to the command timeout, so keep it small. Since v26.10.2.	Long	5000
`gavAlgoRestoreAwaitTimeout`	Milliseconds a whole-graph algorithm (`algo.wcc`, `algo.pagerank`, `algo.bfs` and the other `algo.*` procedures) waits for a Graph Analytical View whose deferred restore from disk is still in flight, so the first call after a reopen runs on the restored view instead of scanning every record. The wait ends early when the restore finishes or the command timeout expires. `0` does not wait. A command without its own timeout can block for the whole budget, so lower it if such commands must answer quickly. Separate from `gavRestoreAwaitTimeout`, which makes `open()` itself block.	Long	60000

Name	Description	Type	Default Value
`graph.edgeAppendMerge`	At commit, when the only conflict on an edge-list page is concurrent in-chunk edge appends (which commute), re-apply the appends on top of the newer page version instead of failing the whole transaction with a retryable conflict. Removes the retry storm on super-node (hot vertex) edge insertion. See Super-Nodes.	Boolean	true
`graph.edgeListInitialChunkSize`	Size in bytes of the first chunk of a vertex's edge list. Each further chunk doubles the previous one up to 8192, so the space a vertex allocates is the sum of that series - a smaller first chunk does not necessarily use less space, it just takes more chunks (each with its own header) to reach the same capacity. The best value follows the degree distribution: around 128 suits an average degree near 10, the default suits very sparse graphs, and above degree 100 the setting barely matters. Values below 32 are clamped. See Lightweight Edges.	Integer	64
`graph.ghostEdgeProbe`	Whether the walks that follow adjacency entries without loading the edge record (variable-length Cypher hops, the BFS, DFS and path walks, `path.subgraphNodes` and `path.subgraphAll`) check once per vertex reached that the edge of the entry still has its record, and skip and report a dangling entry. The probe costs about a quarter to a third of a variable-length hop query, on every database, while only a database with a data-integrity anomaly can hold a dangling entry. Off, those walks follow every entry, as a fixed-length hop, `out()`/`in()` and the counts of a vertex's edges always do, and the repair is `CHECK DATABASE FIX`. Set to `true` on a database suspected to hold dangling entries until it is repaired, so that its answers do not depend on the projection. A hop served by a Graph Analytical View is not probed either way: the view answers from the adjacency entries it was built from. *Since v26.11.1;* `false` *by default since v26.12.1,* `true` *in 26.11.1*	Boolean	false

Name	Description	Type	Default Value
`graph.supernodeThreshold`	Approximate number of edges (per vertex, per direction) after which the vertex's edge list is promoted to the striped super-node layout, spreading further appends over multiple files so concurrent insertions on the same hot vertex do not contend. **Forward-incompatible on first use**: promotion writes a new record type, so once any vertex promotes the database can no longer be opened by older releases; promotion is one-way. Iteration order on promoted vertices is approximate (newest-generation-first) instead of strict reverse-insertion. 0 disables promotion entirely (databases stay fully readable by older releases). See Super-Nodes.	Integer	4096
`graph.supernodeStripes`	Number of stripes (separate edge-list files) a super-node's edge list is spread over at promotion. The stripes are hosted in a per-type bucket pool of this many files, created once per type at its first promotion (types without super-nodes cost no files). Write parallelism saturates at the number of concurrent writers, so values beyond the CPU cores rarely help. Values below 2 disable promotion entirely. Recorded per vertex at promotion time. See Super-Nodes.	Integer	16
`gremlin.engine`	Gremlin engine to use. By default the native `java` engine is used. The `auto` setting uses the legacy `groovy` engine in case parameters are set, otherwise the native `java` one, and is not recommended for security-critical deployments. If you have compatibility issues with gremlin statements that use lambdas or in general, switch to the `groovy` one	String	java
`gremlin.client.port`	Port of the Gremlin Server the remote `ArcadeGraph` client connects to. `0` uses the port the server advertises, falling back to `8182`. Set it when the port is reached through a mapping the server does not know about	Integer	0
`gremlin.timeout`	Default timeout for gremlin commands (in ms). Deprecated	Long	30000

Name	Description	Type	Default Value
`index.buildCommitLockTimeout`	Timeout in ms a bulk index build waits for the file locks of one of its commits, replacing `commitLockTimeout` for those commits only. A vector index build commits in chunks, and each of those commits is a step of a build that can cost tens of minutes, so losing one to the interactive timeout discarded the whole build - while the contender it was queued behind was an ordinary commit that would be done in milliseconds. This bounds only how long the build **waits**, never how long it holds a lock, so raising it cannot make any other transaction slower. 0 or less waits indefinitely. The effective value is clamped to be at least `commitLockTimeout`, so setting it lower has no effect.	Long	60000
`indexCompactionMinPagesSchedule`	Minimum number of mutable pages for an index to be schedule for automatic compaction. 0 = disabled	Integer	10
`indexCompactionRAM`	Maximum amount of RAM to use for index compaction, in MB	Long	300
`initialPageCacheSize`	Initial number of entries for page cache	Integer	65535
`queryMaxRangeSize`	Maximum number of elements a `range()` expression may produce; past it the query is refused before any element is generated. The range is lazy and takes heap only when it must be materialized. A negative value means no limit. Left at the default it scales with the JVM maximum heap, never below 1000000	Long	(heap dependent, min 1000000)
`rebuild.repartition.maxBufferedRids`	Maximum number of misplaced RIDs `REBUILD TYPE … WITH repartition = true` may buffer in heap (about 16 bytes each, so about 160MB at the default). Past it the command fails with an error suggesting smaller batches	Integer	10000000
`timeSeriesTagDictionaryMaxSize`	Maximum number of distinct values the tag dictionary of one time-series type may hold. Tags are dictionary-encoded in RAM, so this caps its footprint and turns a high-cardinality column wrongly declared as `TAG` into a clear error. 1 million values take roughly 100MB	Integer	1000000

Name	Description	Type	Default Value
`timeSeriesUndeclaredKeys`	What a time-series write (InfluxDB line protocol over HTTP, the gRPC `TimeSeriesWrite` calls, SQL `INSERT` and Prometheus remote write) does with a sample that carries a tag, field, property or label its type does not declare in that role. `reject` drops the sample and reports the key: a partial write for the network protocols (the valid samples are stored and the answer is a `400`), a failed statement for SQL `INSERT`. `ignore` stores the sample without the undeclared keys, as schema-less producers such as Telegraf need. Before v26.10.2 such keys were always dropped silently *(since v26.10.2)*	String	reject
`truncateBatchSize`	Number of records `TRUNCATE TYPE` and `TRUNCATE BUCKET` delete per committed transaction, which keeps each HA Raft entry small. Ignored when `TRUNCATE` runs inside a transaction opened by the caller: then a `ROLLBACK` restores every record	Integer	1000
`txPageSlotMerge`	At commit, when a page conflicts only because concurrent transactions changed different records on it, re-applies this transaction's record changes on the newer page instead of failing with a `ConcurrentModificationException`. Covers inserts, updates that stay in the page and deletes of plain records. A real conflict on the same record still raises the exception, and so does, under `READ_COMMITTED`, a page whose records the transaction read from its own copy and another transaction changed meanwhile (since v26.12.1)	Boolean	true
`txPageSlotMergeMaxBytes`	Per-transaction cap, in bytes, on the record images kept for `txPageSlotMerge`. Past it the merge is disabled for the rest of that transaction, which bounds the heap of very large transactions	Long	16777216
`vectorIndex.compactionBloatFactor`	How much larger than its live vectors an `LSM_VECTOR` data file may grow before it compacts itself (`3` means about two thirds of the file is garbage). Each compaction rebuilds the graph. `0` compacts only on an explicit `COMPACT INDEX`	Integer	3

Name	Description	Type	Default Value
`vectorIndex.graphBuildCacheSize`	Maximum number of vectors to cache in memory during HNSW graph building. Higher values speed up construction but use more RAM. RAM usage = cacheSize × (dimensions × 4 + 64) bytes. 0 (the default) sizes it automatically: the cache holds the whole corpus when it fits `vectorIndex.graphBuildCacheMaxHeapPercent`, for indexes whose vectors live in the documents (no quantization, or PRODUCT) and, for inline-quantized ones (INT8/BINARY) as well. Setting an explicit value opts out of that auto-sizing. *Since v26.11.1* an INT8/BINARY corpus that does not fit is not cached as float vectors: the build keeps the quantized rows themselves, so the same RAM holds 4 times (INT8) to 30 times (BINARY) as many vectors, and reads the others from the index pages	Integer	0
`vectorIndex.graphBuildCacheMaxHeapPercent`	Maximum share of the JVM heap the auto-sized graph-build cache may use. Only applies when `vectorIndex.graphBuildCacheSize` is left at 0. A corpus larger than this budget still builds: an INT8/BINARY index keeps as many quantized rows as the budget pays for and reads the rest from the index pages (*since v26.11.1*), any other index evicts instead of holding everything. It is measured against the `-Xmx` ceiling and then capped at 90% of the heap currently available, so the same corpus gets the same cache whether it was ingested into this JVM first or not, while an online rebuild holding the old graph resident still asks for less. Values above 90 are clamped to 90	Integer	25
`vectorIndex.graphBuildDiagnostics`	Logs memory and file size diagnostics while a vector graph is built	Boolean	true
`vectorIndex.graphBuildParallelism`	Number of threads that build the HNSW graph of a vector index, the most expensive part of a build. `0` (default) means the available cores minus one, leaving room for queries during an online rebuild. *Since v26.11.1* the PQ training and encoding of a `PRODUCT`-quantized index run on the same pool, so this bounds the CPU of the whole rebuild; before, PQ training ran on JVector's own default executors	Integer	0

Name	Description	Type	Default Value
`vectorIndex.pqRetrainGrowthFactor`	A graph rebuild of a `PRODUCT`-quantized vector index keeps its PQ codebook and encodes only the vectors added since, as long as the codebook still fits the data. This is the first of the two tests of that fit: the codebook is trained again, and every vector encoded again, once the index holds more than this many times the vectors it held when the codebook was trained. 0 or less retrains on every rebuild, the behavior before v26.11.1. `REBUILD INDEX` always trains a new codebook. See PQ codebook reuse. *Since v26.11.1*	Float	2.0
`vectorIndex.pqRetrainErrorRatio`	The second test of whether a `PRODUCT`-quantized vector index keeps its PQ codebook across a graph rebuild: the mean relative reconstruction error of a sample of up to 1024 vectors added since the codebook was trained, against the same error measured on the vectors it was trained on. Once the new vectors are encoded more than this many times worse, their distribution has drifted away from the codebook, and it is trained again. Applies once at least 32 vectors have been added since the training. 0 or less disables the test, leaving `vectorIndex.pqRetrainGrowthFactor` alone to decide. See PQ codebook reuse. *Since v26.11.1*	Float	1.25

Name	Description	Type	Default Value
`vectorIndex.searchCacheSize`	Maximum number of vectors kept in the per-index search cache. The cache is shared by every query on the index and survives across queries, so a working set that fits stays resident instead of being re-read from the documents (or from the quantized index pages) on every beam-search hop. Since 26.11.1 the vectors are held in flat float storage, allocated in chunks of at most 4 MB as vectors are loaded, and scored where they lie: RAM usage = cacheSize × 8 bytes, plus dimensions × 4 bytes per vector cached (before 26.11.1: cacheSize × (dimensions × 4 + 64) bytes). A BINARY index caches the one-bit codes its searches are scored on instead (since 26.11.1): cacheSize times (⌈dimensions / 64⌉ × 8 + 8) bytes, the code held in whole 64-bit words plus two ints per slot (about 200 MB for 1M slots of 1536 dimensions). 0 (the default) sizes it automatically from the range of vector ids (the number of vectors ever indexed, deleted ones included until the next rebuild), rounded up to a power of two to leave room for growth and capped by `vectorIndex.searchCacheMaxHeapPercent`: when the round-up does not fit the cap, the cache takes the cap itself, so every vector still gets a slot of its own whenever the cap covers the id range. -1 disables the cache entirely	Integer	0
`vectorIndex.searchCacheMaxHeapPercent`	Upper bound, as a percentage of the JVM heap currently **available** rather than of `-Xmx`, on the RAM the automatically sized search caches of a database may use together (see `vectorIndex.searchCacheSize`). Since 26.12.1 the share is no longer per index: a type has one vector index per bucket, and an index that finds the share already planned on by the other indexes' caches gets no cache, so its searches read the vectors from the pages. Ignored when the cache size is set explicitly. Values above 90 are clamped to 90	Integer	25

Name	Description	Type	Default Value
`vectorIndex.deltaCacheSize`	Maximum number of vectors written since the last graph rebuild that keep a copy in memory. Every write is appended to a delta buffer so the vector is searchable before it reaches the HNSW graph, and only a rebuild drains that buffer - so an ingest that outruns the rebuilds used to grow a second full copy of the corpus in RAM, which is what made a 4.2M x 768-dimension bulk load run out of memory at `-Xmx16g`. The vector is written to disk before it is buffered, so entries past this cap keep only their identity and the vector is read back from disk when a query needs it. RAM usage = deltaCacheSize × (dimensions × 4 + 64) bytes per index: a type has one vector index per bucket, so an explicit size is multiplied by the number of buckets of every type with a vector index. 0 (the default) sizes it automatically from `vectorIndex.deltaCacheMaxHeapPercent`, which bounds all the vector indexes of the database together. -1 keeps every buffered vector in memory, which is unbounded. Reported by the index statistics as `deltaResidentVectors` and `deltaResidentVectorsCapacity`.	Integer	0
`vectorIndex.deltaCacheMaxHeapPercent`	Share of the `-Xmx` ceiling the automatically sized delta buffer caches of all the vector indexes of a database may use together (since 26.12.1; before, every index took this share on its own, so a schema with dozens of vector indexes could exhaust the heap during a load). A type has one vector index per bucket and they all draw on the same share, so the bound does not grow with the number of buckets or of vector properties: an index that finds the share used up keeps no copy in memory and reads its buffered vectors back from disk. The share is then capped at 90% of the heap currently available - the same measure `vectorIndex.graphBuildCacheMaxHeapPercent` uses, so a rebuild holding the old graph and an ingest filling the buffer are judged against the same free heap. Ignored when `vectorIndex.deltaCacheSize` is set explicitly. This is a margin rather than a reservation: neither cache reserves heap from the other, and what keeps them in check is that memory one of them keeps lowers the next reading for both. Values above 90 are clamped to 90.	Integer	10

Name	Description	Type	Default Value
`vectorIndex.locationCacheSize`	Maximum number of vector locations to cache in memory per vector index. Set to -1 for unlimited. Each entry uses ~56 bytes. Recommended: 100000 for datasets with 1M+ vectors	Integer	-1
`vectorIndex.prefilterMaxSelectivity`	Maximum fraction of an index's live vectors that a query's RID allow-list may cover and still resolve the allow-list to its ordinals and score them directly, instead of paying for a `Bits`-filtered HNSW graph walk that gets more expensive, not less, as the allow-list narrows (a filter is only checked once a node is popped from the search beam, so a very selective filter can make the walk expand trying to fill `k` instead of shrinking). Applies to plain k-NN search and to `groupBy` search; see `vectorIndex.prefilterApproximateMaxSelectivity` for the separate threshold the PQ-approximate (zero-disk-I/O) search path uses. Set to 0 to always use the graph walk.	Float	0.2
`vectorIndex.prefilterApproximateMaxSelectivity`	The `vectorIndex.prefilterMaxSelectivity` threshold, but for PQ-approximate (zero-disk-I/O) vector search. Kept as a separate, lower setting because PQ scores a candidate from in-memory codes roughly an order of magnitude cheaper than the exact path's page/document read, so its graph walk stays the cheaper option down to a narrower allow-list than the exact path's. Set to 0 to always use the graph walk.	Float	0.05
`vectorIndex.mutationsBeforeRebuild`	Number of mutations (inserts/updates/deletes) before rebuilding the HNSW graph index. Higher values reduce rebuild cost but may return slightly stale results. Recommended: 50-200 for read-heavy, 200-500 for write-heavy workloads	Integer	100

Name	Description	Type	Default Value
`vectorIndex.rebuildGraphRatio`	Fraction of the current graph size that must accumulate as pending mutations before the HNSW graph is rebuilt, on top of the absolute `vectorIndex.mutationsBeforeRebuild` floor. A rebuild always reindexes the whole graph, so a fixed count makes it cost O(index size) for every few new vectors and turns bulk ingestion quadratic; scaling the threshold with the graph amortizes rebuilds instead. Vectors waiting for the next rebuild stay exactly searchable in memory meanwhile, so a higher ratio trades a slightly longer per-query scan of those vectors for far less rebuild CPU. Set to 0 to use only the absolute threshold	Float	0.2
`vectorIndex.maxPendingMutations`	Ceiling on the threshold computed from `vectorIndex.rebuildGraphRatio`. Set to 0 for no ceiling. Being a fixed count, it stops scaling once the ratio-derived threshold reaches it (at the defaults, from 250,000 vectors upward), so past that point the same absolute per-query scan cost lands on an index of any size. `vectorIndex.maxDeltaScanRatio` is the size-independent bound and is the setting to reach for when query latency, rather than rebuild frequency, is what needs protecting. Note that this bounds how often a rebuild is triggered, not how many vectors wait for one: writes keep arriving between rebuilds regardless. What bounds the **memory** those waiting vectors occupy is `vectorIndex.deltaCacheSize`.	Integer	50000

Name	Description	Type	Default Value
`vectorIndex.maxDeltaScanRatio`	How much work a query may spend scanning vectors written since the last graph rebuild, as a multiple of the work its graph search already does, before a rebuild is triggered to absorb them. Those vectors are answered by a straight scan, so that part of a query grows with how many are waiting while the graph search it supplements grows only with the logarithm of the index size: at the default `vectorIndex.maxPendingMutations` the scan has been measured at four fifths of query time. The count-based settings cannot bound it, so this one is measured rather than assumed - ArcadeDB records how many nodes its graph searches actually visit and compares the waiting vectors against that. 1.0 lets the scan cost about as much as the search. Lower it for latency-sensitive read-heavy workloads, raise it to rebuild less often, set 0 to disable. A rebuild is only triggered once the scans have also cost as much as the rebuild will, so the extra rebuild CPU this can introduce is bounded by the query CPU it removes. Because it is evaluated when queries run, a write-only workload never triggers it. Reported by the index statistics as `deltaScanBudget`, `deltaScanWorkSinceRebuild` and `deltaScanWorkTarget`.	Float	1.0

Name	Description	Type	Default Value
`vectorIndex.inactivityRebuildTimeoutMs`	Inactivity timeout in milliseconds before flushing buffered vectors and rebuilding the HNSW graph. When mutations exist but haven't reached the rebuild threshold, a timer starts after the last mutation. On a graph under 1,000 vectors the rebuild is cheap and fires for any pending mutation count; on a larger graph it only fires once pending mutations reach at least 10% of the effective rebuild threshold, so a single stray insert does not force a full graph rebuild. The size compared against is the number of vectors the index holds, not the part of the graph the session has loaded, so an ingest-then-idle process that never queries is gated the same way. The timer does not fire while a bulk load is open on the database (`GraphBatch`, and therefore `GraphImporter` and the HTTP batch endpoint): a loader that pauses for an index compaction or a garbage collection looks idle to this timer, and the rebuild it used to start covered only the part of the dataset loaded so far and was made obsolete by the rest of the load. The rebuild is now scheduled once, when the load finishes. Set to 0 to disable. Recommended: 10000-30000 for low-volume ingestion	Integer	15000

Name	Description	Type	Default Value
`vectorIndex.rebuildMaxHeapPercent`	Share of the currently **available** heap that an online vector graph rebuild's estimated peak footprint may occupy before the rebuild is deferred instead of attempted. An online rebuild keeps the old graph resident so searches keep working and pays for a full new build's working set on top of it; with no gate it simply attempts the rebuild and dies with an OutOfMemoryError when it does not fit. What keeping the old graph resident costs is measured rather than assumed, so a graph served from pages - the shape after a reopen - is not charged as a second copy of itself in memory, and the available heap it is judged against counts the page read cache as reclaimable: a rebuild that fits only by giving some of it up evicts exactly that shortfall first (oldest pages, across every open database, since the cache is shared) and is still deferred if the cache cannot hand those bytes over. A deferred cycle is not lost: pending vectors stay exactly searchable through the in-memory delta buffer, so the cost is a longer delta scan per query rather than wrong or missing results, and the deferral is logged and counted as `rebuildsDeferredForMemory` in the index statistics. A first build in the background, a rebuild on close, `REBUILD INDEX` and `COMPACT INDEX` are never declined. Since v26.12.1 the rebuild a **query** needs because the index has no usable graph is judged against the same share: it gives up page cache, then runs with a smaller build cache (slower, same graph, counted as `searchRebuildsWithReducedCache`), and when even the graph alone does not fit the query fails with a retryable `VectorGraphBuildRefusedException` (HTTP 503, counted as `searchRebuildsRefusedForMemory`) instead of building into an OutOfMemoryError. Since v26.10.2, when the rebuild does not fit beside the in-memory graph, the graph is first swapped for its persisted on-disk copy (searches read it from pages until the new graph is published, counted as `residentGraphDemotions`) and the rebuild is deferred only if it still does not fit. Values above 90 are clamped to 90. Set to 0 to disable the gate.	Integer	90

Name	Description	Type	Default Value
`vectorIndex.rebuildDeferralCooldownMs`	Minimum time in milliseconds before another online vector graph rebuild may be attempted after one was deferred for lack of heap (see `vectorIndex.rebuildMaxHeapPercent`). A deferral does not consume the pending mutations that triggered it - only a successful build does - so without a cooldown the next search re-triggers it immediately, and since a search checks on every query a large heap-constrained index would spawn a rebuild thread, take the JVM-wide rebuild permit and log a warning once per query. A rebuild that completes clears the cooldown. Set to 0 to retry on the next trigger.	Integer	30000
`vectorIndex.searcherPoolSize`	Maximum number of vector graph searchers kept per vector index for reuse across queries, which reduces garbage on dense search workloads. `0` (default) means twice the available cores; `-1` disables pooling	Integer	0
`vectorIndex.warmUpAfterSchemaReloadDelayMs`	A schema reload (on an HA replica, every schema entry the incremental refresh declines) replaces every index with a freshly built instance, and a vector index starts out with nothing resident: the next search reads all of its pages to rebuild the location map, then loads or rebuilds the graph on the query thread. This is how long, in milliseconds, the database waits after a reload, with no further reload, before warming in the background - on one thread, one index after the other - the vector indexes whose replaced instance had a graph resident. The wait is what makes a burst of DDL statements cost one warm-up and not one per statement. A search that arrives meanwhile joins the load in progress. Indexes that were never searched are not warmed. Set to 0 to start right after the reload, or to -1 to disable and leave it to the first search. *Since v26.10.2*	Integer	1000
`maxPageRAM`	Maximum amount of pages (in MB) to keep in RAM. When unset it is a quarter of the maximum JVM heap (4096 MB under `-Xmx16g`). A value above 80% of the heap is reduced to half of it	Long	25% of the max heap
`pageFlushQueue`	Size of the asynchronous page flush queue	Integer	512

Name	Description	Type	Default Value
`pageSnapshotEnabled`	When true (the default), a full backup, an HA database verify and an HA snapshot ship read a point-in-time image served from a page-level copy-on-write shadow, so writers keep running at full speed for the duration. When false, or when the shadow exceeds `pageSnapshotMaxSize`, they fall back to freezing the data files instead, which throttles every writer until the operation finishes	Boolean	true
`pageSnapshotMaxRAM`	Memory budget in MB for the copy-on-write shadow of a single point-in-time window. The shadow only holds the pages **modified** while the window is open, once each, so a short backup on a moderately busy database often never touches the disk at all. Beyond this budget the shadow spills to a scratch file (see `pageSnapshotSpillPath`)	Long	64
`pageSnapshotMaxSize`	Hard limit in MB (memory plus spill file) on a single copy-on-write shadow before the window is abandoned and its consumer falls back to freezing the data files. The default `-1` sizes the limit automatically when the window opens: the smaller of the size the page files occupy at that instant - which the shadow provably cannot exceed - and half the space free on the spill volume, but never less than `pageSnapshotMaxRAM`, since that part of the budget never touches the disk. Set a positive value to pin an absolute limit in MB, or `0` for no limit	Long	-1
`pageSnapshotTrackOpeners`	Debug aid: record the stack of the code that opens each point-in-time snapshot window, so that a database close still waiting for a window after 60 seconds names the code path that opened it. Off by default, the window is still listed with its opening thread, age and status. *Since v26.11.1*	Boolean	false

Name	Description	Type	Default Value
`pageSnapshotSpillPath`	Directory for the scratch file a copy-on-write shadow spills into once `pageSnapshotMaxRAM` is exhausted. Empty (the default) uses the database directory, where the shadow competes for space with the very files it is protecting. The file is pure scratch: created on demand, deleted when the window closes, and never read by recovery. Since v26.12.1 a value set for one database must lie inside the database directory or inside the directory the server-level value names (one that does not is ignored with a warning and the database directory is used), and `ALTER DATABASE` of it needs `updateSecurity`	String	
`polyglotCommand.timeout`	Default timeout for polyglot commands (in ms)	Long	10000

Name	Description	Type	Default Value
queryMaxHeapElementsAllowedPerOp	Maximum number of elements (records) allowed in a single query for memory-intensive operations (eg. ORDER BY in heap). If exceeded, the query fails with an OCommandExecutionException. Negative number means no limit. In OpenCypher it also bounds the rows a Cartesian product or a hash join of disconnected patterns buffers, ORDER BY, DISTINCT, UNION, the groups of an aggregation, collect() and the eager read ahead of a write. This setting is intended as a safety measure against excessive resource consumption from a single query (eg. prevent OutOfMemory). A GROUP BY computed by a parallel scan checks the limit on each thread and on the final groups, so its peak memory can reach the limit times the number of threads. A SQL time series aggregation (ts.timeBucket() with GROUP BY) that returns more buckets than this fails instead of exhausting the heap, as the HTTP, gRPC and Grafana time series endpoints already did. It also bounds the index the delete of a vertex keeps in heap to find the incoming edges of unidirectional edge types: past it, each delete scans the types instead. Since v26.10.2 it also bounds the result of a SQL LET subquery, the records a SQL MATCH prefetches for a small root, and what an aggregate that keeps its values holds: list(), set(), map(), unionAll(), intersect(), symmetricDifference(), percentile(), median(), mode() and the ts.* window functions, and in OpenCypher percentileCont(), percentileDisc(), count(DISTINCT ⋯) and the same SQL aggregates called as sql.*. An aggregate is counted by what it holds, not by the rows it is given: a set() of a few distinct values over millions of rows stays far below the limit. Such a query used to run past the limit, so one that holds more than it may now fail where it completed before: raise the limit for it if the heap allows. Since v26.11.1 it bounds a subquery written inline in an expression too (a projection, a function argument, a SET value, the endpoints of CREATE EDGE) and the rows of a user defined SQL function, as it does the same subquery in a LET: one that returned more rows than the limit now fails where it completed before. In OpenCypher it also bounds, since v26.11.1, the list a COLLECT { ⋯ } subquery or a pattern comprehension builds and the rows of apoc.do.when(), and the refusal of either heap limit is no longer turned into a null row by OPTIONAL CALL. Since v26.12.1 it also bounds the nodes a SQL MATCH while:/maxDepth: traversal holds along its	Long	500000

Name	Description	Type	Default Value
`parallelScanAbandonedTimeout`	Milliseconds a parallel scan waits for a result set that is neither read nor closed before giving up: the scan then frees its threads and the next read of that result set fails instead of silently returning fewer rows. A query whose `LIMIT` is reached releases its scan at once, without waiting for this timeout, and a result set left open never stalls other queries: they only run less in parallel until it is closed. Raise it for clients that keep cursors open with long pauses. 0 disables the timeout	Long	600000
`queryIndexMaxSelectivity`	Share of a type's records above which an index search gives way to a full scan of the type, when the scan runs on one thread. Before loading any record, the matching index entries are read alone: when more of them match than this share of the records the type holds, the rows come from a scan filtered by the same condition, otherwise the matching records are loaded in physical order rather than in key order. A scan split across W workers of a parallel scan gives way sooner, at this share divided by (1 + W) / 2, because the index entries are read by one thread whatever the parallelism: with the default, 60% of the type for a sequential scan, 24% on 4 workers, 6% on 18. Applies, in SQL and OpenCypher, only where the order the rows arrive in cannot show in the result - the statement aggregates them or sorts them with an `ORDER BY` the index does not serve - and where the scan answers the same rows. 0 disables it. Since v26.11.1 a change on an open database (`ALTER DATABASE`) reaches queries already run once: every database setting change clears the cached query plans	Float	0.6

Name	Description	Type	Default Value
`queryLiteralParameterization`	When true, the literal values written in a SQL or OpenCypher query are extracted into generated parameters before the statement and plan caches are looked up, so queries that differ only in those values (`… WHERE id = 12345`, `… WHERE id = 67890`) share one parsed statement and one execution plan instead of being parsed and planned again for every value. This is what queries generated by an application, a query builder or an LLM (text-to-SQL, text-to-Cypher, GraphRAG tools) typically look like. A literal is extracted only where a parameter means exactly what the literal meant, so the result, its column names and the Java types of its values are the same: the literals of a projection without an alias (the column is named after its text), of `ORDER BY`, `GROUP BY`, `SKIP`, `LIMIT` and `TIMEOUT`, of a SQL `FROM` target, of OpenCypher procedure arguments and of schema statements stay in the text, as do booleans, `null` and empty strings. A query that already uses parameters, or that is run again with the same text, costs the same as before, and the first query of a shape runs as written, so a workload that never repeats a shape pays one parse per query as before. The shared plan is the one built for the second query of the shape, as a plan with parameters is built for the first values it is run with: where values are so skewed that some need a plan of their own, turn the setting off for that database. `EXPLAIN` and `PROFILE` plan the text as written, so they show the values you wrote. A query that itself mentions a name starting with `__lit_` is not parameterized, and names with that prefix are reserved for the generated parameters: a parameter named `__lit_...` reads only the value bound under its plain name, never one bound as `:__lit_...`, so a caller cannot replace a literal of the query text. Set to false to key the caches on the exact query text *(since v26.10.2)*	Boolean	true

Name	Description	Type	Default Value
`queryParallelScan`	When true, a full scan of a type is read by several threads, for more throughput on multi-core machines: with or without a `WHERE` clause, and with `count`, `sum`, `avg`, `min`, `max` and their `GROUP BY` computed on those threads too, in SQL and in OpenCypher, as is a plain `SELECT DISTINCT` of fields over a type of at least 10,000 records. Rows and groups come back in the same order as a sequential scan, but the sum or average of floating-point values may differ in the last digits from one run to the next, as the threads add in no fixed order. A `GROUP BY` of many keys, where every thread would otherwise build its own group for nearly every key, switches to a key exchange once a thread has created a few thousand groups: from then on the rows of the keys that thread does not hold are aggregated in one set of groups shared by the threads and partitioned by key, so each of those keys is held and aggregated once rather than once per thread. This roughly halves the memory such a query reserves from `queryMaxHeapRAM` and makes it faster, and it changes nothing for a `GROUP BY` of few keys. `EXPLAIN`/`PROFILE` reports it on the aggregation step as `N groups by key exchange` *(since v26.10.2)*. Used only outside an explicit transaction	Boolean	true
`queryParallelScanMinBuckets`	Minimum number of buckets a type needs for its full scans to run in parallel. A type with fewer buckets is scanned sequentially, unless one of its buckets is large enough to be split in page ranges (see `queryParallelScanPagesPerUnit`)	Integer	2
`queryParallelScanPagesPerUnit`	Minimum number of pages of the unit of work a parallel scan cuts a bucket in: a bucket of at least twice as many pages is read by several threads, each on a range of its pages, so a type with a single bucket (the default) is scanned in parallel too. 0 disables the split: each bucket is then read by one thread	Integer	32
`queryParallelScanMaxBatchBytes`	Maximum bytes of record content one batch of a parallel scan holds in memory. The parallel scan reads records on its worker threads and hands them over in batches of up to 256 rows; with large records this bound closes a batch earlier. 0 or a negative value removes the bound (row count only)	Long	16777216

Name	Description	Type	Default Value
`queryBatchMaxBytes`	Maximum bytes a scan of a bucket reads ahead of the query in one batch. A scan prefetches up to 1,024 records per bucket; with records that span several pages (a vertex with a big nested document) that is a lot of memory no query budget accounts for, and many concurrent queries could exhaust the heap. The batch ends once the bytes it copied out of the pages reach this size, whatever the record count (records that fit their own page are views of the cached page and are not counted). The limit also shrinks as running queries take the heap budget (see `queryMaxHeapRAM`): a scan reads ahead at most a sixty-fourth of what is left of it, down to one record at a time. The shares of the pool and of the budget are read once per batch and are not reservations, so the worst case is the number of scans running times the buckets each reads times the smaller of this setting and a sixty-fourth of what is left of the budget. A batch always holds at least one record. The `low-ram` profile sets it to 200KB. A scan that read under this pressure is slower for it, and says so: `PROFILE` marks its step with `[read-ahead reduced in N batches by memory pressure]` and the profiler counts it as `queryHeapScanShrinks`. 0 or a negative value removes the bound and the shrinking *(since v26.10.2)*	Long	1048576

Name	Description	Type	Default Value
`queryScanReadAheadMaxRAM`	Maximum memory (in MB) the scans of all the queries running in the JVM may hold read ahead at once, across every bucket of every database. `queryBatchMaxBytes` bounds the batch of one scan; this limit bounds the sum of all of them, so many concurrent scans cannot hold that many times the bound. Each scan starts with its full batch and reads less as the pool fills (a thirty-second of what is left of it), down to one record at a time. The limit is soft: a batch always holds at least one record, and its bytes are reserved after it is read, so scans that start together can each read a full batch before any of them reserves (the pool can overshoot by about the number of scans times `queryBatchMaxBytes`), and a single record larger than `queryBatchMaxBytes` is still read whole. The bytes go back to the pool when a batch is handed over or when the query that reads it is closed before its end (for a Gremlin traversal, since v26.12.1), and, for a scan the application abandoned, when the garbage collector reclaims it. The pool is reported by the profiler as `scanReadAheadReserved`, `scanReadAheadReservedPeak` and `scanReadAheadLimit`. 0 or a negative value disables the pool. The `low-ram` profile sets it to 16 *(since v26.10.2)*	Long	200
`schemaBulkDDLScript`	Runs a SQL script of two or more statements that are ALL schema definition DDL (`CREATE TYPE`, `CREATE PROPERTY`, `CREATE INDEX` on a type the script creates, ...) as one schema session, so the schema file is written once for the whole script instead of once per statement and, under HA, the batch is replicated as one Raft entry. The database write lock is held for the whole script instead of once per statement; under HA a leader commit waits for the session at most `ha.quorumTimeout` before proceeding, so keep such scripts short. There is no rollback: if a statement fails, the ones before it stay applied and are published. A script that mixes in other statements (DML, `REBUILD`, `TRUNCATE`, `REFRESH`, `COMPACT INDEX`, materialized views, repartitioning, an index on a type that already existed) runs one session per statement regardless. Set to `false` to always use one session per statement. See reference/sql/sql-script.pdf and reference/storage.pdf.	Boolean	true

Name	Description	Type	Default Value
`sql.letSubqueryCacheSize`	Maximum number of distinct outer values whose result a per-record LET subquery (`SELECT ⋯ LET $x = (SELECT ⋯)`) remembers within one execution of the query. When many rows feed the subquery the same values (rows sharing one parent, office or category), it runs once per distinct value instead of once per row. The subquery is keyed on the outer values it actually reads, including a single field read as `$parent.$current.<field>`. Subqueries calling non-deterministic or user-defined functions, or that are not read-only, are never cached, and any change to the database drops the cache. Results larger than 1000 rows are not cached. 0 disables the cache	Integer	128
`sql.maxExpressionDepth`	Maximum nesting depth of parentheses, brackets, braces (map and JSON literals) and `CASE` expressions allowed in a single SQL statement. A query past the limit is refused at parse time, which stops a deeply nested expression from tying up a worker thread for a very long time. Raise it for one database with `ALTER DATABASE` when a legitimately deep query needs it *(the per-database value is actually read; before that only the server-wide value had any effect)*	Integer	200
`cypher.maxExpressionDepth`	The same limit for OpenCypher, covering nested parentheses, list and map literals, function arguments, pattern parentheses, `CALL`/`EXISTS`/`COUNT`/`COLLECT` subqueries (all counted together), and the length of a chain of `AND`/`OR` terms *(per-database, see `sql.maxExpressionDepth`)*	Integer	200
`cypher.maxClauses`	Maximum number of clauses (`MATCH`, `WITH`, `CREATE`, ...) in one OpenCypher query or subquery body. A longer chain is refused at parse time, because it would otherwise exhaust the thread stack while the query runs. Each `UNION` branch counts on its own. A value below 1 falls back to the default	Integer	1000

Name	Description	Type	Default Value
`sqlStatementCache`	Maximum number of parsed statements to keep in cache. A statement must be hit twice to be protected from a burst of one-off query texts. Statements that differ only in the values they embed share one entry (see `queryLiteralParameterization`, *since v26.10.2*), so raise this value only if you generate many statements of different shapes. It also sizes the cache of SQL execution plans. A statement that holds more than 4,096 literal values, typically a bulk write with its data written in the text (`INSERT INTO T CONTENT [⋯]`, `UNWIND [⋯] AS row CREATE ⋯`), is parsed for its execution and not cached, so it does not keep its parse tree on the heap *(since v26.12.1)*. Since v26.11.1 a change on an open database applies at once	Integer	300
`timeSeriesDecodedBlockCacheRAM`	Memory budget, in MB and **per shard**, for keeping decoded TimeSeries data in memory so that reading the same compacted block twice decodes it once. This is what makes a repeated query - a dashboard polling the latest point of a host, for example - answer without re-reading and re-decoding the block it already read. Since the budget is per shard, the worst case for one type is this value times its `SHARDS`, and it is only reached by queries that actually touch that many distinct blocks. A shard sizes its cache when it is opened, so changing this on a running database applies to shards opened afterwards; reopen the database to apply it everywhere. Set to 0 to disable caching. Left at the default it scales with the JVM maximum heap (heap/512, never below 4MB)	Long	(auto)
`txRetries`	Number of retries in case of MVCC exception. A follower's refusal of leader-only work that names the leader (a schema change, for example) is not retried, since every attempt on that node would be refused the same way: it reaches the caller on the first attempt *(since v26.10.2)*	Integer	3

Name	Description	Type	Default Value
`txRetryDelay`	Cap, in milliseconds, on the random wait before the next transaction retry: an exponential backoff with full jitter, drawn from `1` to `min(txRetryDelay, txRetryDelayBase * 2^attempt)`. Helpful in case of high concurrency on the same pages (multi-thread insertion over the same bucket). Applied by the embedded `transaction()`, by `COMMIT RETRY` and, by the remote `RemoteDatabase.transaction()` too. Since v26.10.2 it is also applied between the replays of an asynchronous batch commit that hit a conflict. 0 disables the delay	Integer	100
`txRetryDelayBase`	Starting size, in milliseconds, of the transaction retry backoff window, doubled on each further attempt up to `txRetryDelay`	Integer	2
`txStaleReadCheck`	Refuses, with a retryable `ConcurrentModificationException`, a property write on a record read in the current transaction when another transaction committed a change to that record in between, so a value computed from the older read can never overwrite the concurrent change (a lost update under `READ_COMMITTED`). Adding or removing edges on a vertex is never refused. Only direct property assignment and removal is checked, not an in-place change to a list, map or embedded document got from the record. SQL `UPDATE` and Cypher `SET`/`MERGE`/`REMOVE` already compute their values from the latest data and are not affected. Set to false to restore the previous last-writer-wins behavior	Boolean	true
`txStatementSavepointMaxEntries`	A statement that fails inside a transaction you keep is taken back on its own, and the transaction goes on, instead of being refused its commit. To do it the engine journals what the statement changes in the transaction, up to this many entries; a statement that changes more is not taken back when it fails, and the transaction can no longer be committed, as before statement savepoints. A bulk statement thus pays no memory for a journal it could not use, while the many small statements of an interactive transaction are all undone exactly. 0 disables statement savepoints. See A statement that fails halfway *(since v26.11.1)*	Integer	100000
`txWAL`	Uses the WAL	Boolean	true

Name	Description	Type	Default Value
txWalFiles	Maximum number of concurrent files to use for tx log. 0 or a negative value = available cores, which is also the default. Since v26.11.1 a WAL file is opened on the first commit that needs it, instead of all of them when the database opens, and txWalIdleTimeout closes the ones left idle (see concepts/transactions.pdf)	Integer	(machine dependent)
txWalFlush	Flushes the WAL on disk at commit time. It can be 0 = no flush, 1 = flush without metadata and 2 = full flush (fsync). In production server mode, this is automatically set to 1 if not explicitly configured. A replicated commit of an HA cluster is made durable by the Raft log instead: its WAL record is written without a flush whatever this setting says *(since v26.12.1)*. See concepts/transactions.pdf and concepts/transactions.pdf	Integer	0
txWalGroupCommit	With txWalFlush 1 or 2, lets concurrent commits share one WAL flush: a commit still returns only once its WAL is on disk, but a transaction is visible to others a few milliseconds before. Set to false to make a transaction visible only once its WAL is on disk. Only for databases that are not replicated, with the asynchronous page flush. Since v26.11.1. See concepts/transactions.pdf	Boolean	true
txWalIdleTimeout	Milliseconds a WAL file of the pool can stay without writes before the WAL housekeeping closes and removes it, once all its pages reached the data files. Together with the WAL files opened on demand, a database that is only read, or not written for this long, holds no WAL file open. A database written less often than this creates, locks and deletes a WAL file per write, and its WAL file counter grows with them. 0 or a negative value keeps every WAL file open until the database is closed. See concepts/transactions.pdf *(since v26.11.1)*	Long	60000
typeDefaultBuckets	Default number of buckets to create per type	Integer	1

Available Plugins

Name	server.plugins-String
Gremlin	GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugin
gRPC	gRPC:com.arcadedb.server.grpc.GrpcServerPlugin

Name	server.plugins-String
MongoDB	MongoDB:com.arcadedb.mongo.MongoDBProtocolPlugin
Postgres	Postgres:com.arcadedb.postgres.PostgresProtocolPlugin
Prometheus	Prometheus:com.arcadedb.metrics.prometheus.PrometheusMetricsPlugin
Redis	Redis:com.arcadedb.redis.RedisProtocolPlugin

gRPC

The gRPC server is implemented by the `GrpcServerPlugin`, which is bundled in the `full` distribution but **not started by default**. To enable it, register the plugin in `server.plugins` (see available plugins):

```
-Darcadedb.server.plugins=gRPC:com.arcadedb.server.grpc.GrpcServerPlugin
```

Once enabled, the server listens on port `50051` by default. Since v26.10.2 every `grpc.*` setting below is a registered server setting, so it can be supplied in the server configuration file, as an environment variable or as a JVM system property (`-Darcadedb.grpc.*`). Before that, only `grpc.port` was, and the others worked as system properties only: a `grpc.tls.enabled` written in the configuration file was ignored and the endpoint stayed in cleartext. An invalid `grpc.mode` now makes the server refuse to start instead of starting no listener. The `server.plugins` setting itself is a standard server setting and can be supplied the same ways.

Name	Description	Type	Default Value
grpc.enabled	Start the gRPC server when the plugin is registered. Set to `false` to keep the plugin loaded but the listener stopped	Boolean	true
grpc.port	Port for the standard gRPC server. A registered server setting (so it is also resolved from environment variables and listed in the settings API), because HA reads it to advertise a peer's gRPC endpoint - see the `grpc` field of `arcadedb.ha.serverList`	Integer	50051
grpc.host	Host/interface to bind to: every local address the name resolves to. Applies to the standard server only; the xDS server (mode `xds` or `both`) has no host setting and listens on every interface. Before 26.10.2 the setting was read but never applied	String	0.0.0.0
grpc.mode	Server mode: `standard`, `xds`, or `both`	String	standard

Name	Description	Type	Default Value
`grpc.xds.port`	Port for the xDS server (used when `grpc.mode` is `xds` or `both`)	Integer	50052
`grpc.tls.enabled`	Enable TLS for the gRPC server. Only `true` or `false` are accepted: any other value (for example `yes`) makes the server refuse to start instead of running without TLS	Boolean	false
`grpc.tls.cert`	Path to the TLS certificate chain file (required when `grpc.tls.enabled=true`)	String	(none)
`grpc.tls.key`	Path to the TLS private key file (required when `grpc.tls.enabled=true`)	String	(none)
`grpc.maxMessageSize`	Maximum inbound message size, in MB	Integer	100
`grpc.maxMetadataSize`	Maximum inbound metadata (headers) size, in KB (minimum 1)	Integer	16
`grpc.maxConcurrentTransactions`	Maximum number of gRPC transactions open at the same time. `0` or a negative value disables the bound	Integer	1000
`grpc.maxConcurrentTransactionsPerPrincipal`	Maximum number of gRPC transactions one user may have open at the same time. `0` or a negative value disables the bound	Integer	100
`grpc.reflection.enabled`	Enable the gRPC server reflection service (used by tools such as `grpcurl`)	Boolean	true
`grpc.health.enabled`	Enable the standard gRPC health-checking service	Boolean	true
`grpc.compression.enabled`	Understand compressed requests and offer compression. When `false` the endpoint answers `UNIMPLEMENTED` to a compressed request, never compresses a response, and `grpc.compression.force` is ignored with a warning *(since v26.10.2)*	Boolean	true
`grpc.compression.force`	Force compression on all outbound messages, to a client that accepts the codec. Needs `grpc.compression.enabled=true`	Boolean	false
`grpc.compression.type`	Compression algorithm used when `grpc.compression.force=true`	String	gzip
`grpc.tx.maxIdleMs`	Milliseconds a gRPC transaction may stay idle before the reaper rolls it back	Long	300000
`grpc.tx.maxAgeMs`	Maximum age in milliseconds of a gRPC transaction, busy or not, before the reaper rolls it back. `0` disables the bound	Long	0

Name	Description	Type	Default Value
`grpc.tx.reaperPeriodMs`	Milliseconds between two runs of the gRPC transaction reaper	Long	30000

In addition to the plugin-level keys above, the gRPC service honors four registered SERVER settings that bound result materialization and protect against limitless or slow clients pinning worker threads and exhausting heap: `server.grpcQueryMaxResultRows`, `server.grpcTimeSeriesMaxResultRows`, `server.grpcStreamMaxMaterializedRows`, and `server.grpcStreamWriteTimeoutMs` (see the SERVER settings table). Like the `grpc.*` keys, they can be supplied in the server configuration file, as environment variables or as JVM system properties.

12.3. SQL Syntax

ArcadeDB Query Language is an SQL dialect.

This page lists all the details about its syntax.

Comments

Comments in the ArcadeDB SQL scripts can be C-Style block comments, which are enclosed by `/*` and `*/`,

```
/* This is a single-line comment */

/* This is a multi-
   line comment */
```

as well as classic SQL end of line comments, started by `--` (note the space after the two dashes),

```
SELECT true; -- This is an end of line comment
```

Identifiers

An identifier is a name that identifies an entity in ArcadeDB schema. Identifiers can refer to

- type names
- property names
- index names
- aliases
- bucket names
- method names
- named parameters
- variable names (LET)

An identifier is a sequence of characters delimited by back-ticks `. Examples of valid identifiers are

- `surname`
- `name and surname`
- `foo.bar`
- `a + b`
- `select`

The back-tick character can be used as a valid character for identifiers, but it has to be escaped with a backslash, eg.

- `foo \` bar`

The following are reserved identifiers, they can NEVER be used with a different meaning (upper or lower case):

- `@rid`: record ID
- `@type`: document type
- `@cat`: type category
- `@props`: property types
- `@in`: incoming edges
- `@out`: outgoing edges
- `@this`: current record
- `@size`: current record size

Simplified identifiers

Identifiers that start with a letter or with `$` and that contain only numbers, letters and underscores, can be written without back-tick quoting. Reserved words cannot be used as simplified identifiers. Valid simplified identifiers are

- `name`
- `name_and_surname`
- `$foo`
- `name_12`

Examples of INVALID queries for wrong identifier syntax

```
/* INVALID - `from` is a reserved keyword */
SELECT from from from
/* CORRECT */
SELECT `from` from `from`

/* INVALID - simplified identifiers cannot start with a number */
```

```
SELECT name as 1name from Foo
/* CORRECT */
SELECT name as `1name` from Foo

/* INVALID - simplified identifiers cannot contain `-` character, `and` is a reserved
keyword */
SELECT name-and-surname from Foo
/* CORRECT 1 - `name-and-surname` is a single field name */
SELECT `name-and-surname` from Foo
/* CORRECT 2 - `name`, `and` and `surname` are numbers and the result is the
subtraction */
SELECT name-`and`-surname from Foo
/* CORRECT 2 - with spaces  */
SELECT name - `and` - surname from Foo

/* INVALID - wrong back-tick escaping */
SELECT `foo`bar` from Foo
/* CORRECT */
SELECT `foo\`bar` from Foo
```

Case sensitivity

In current version, type names are case **insensitive**, all the other identifiers are case sensitive, particularly also bucket names.

Reserved words

In ArcadeDB SQL the following are reserved words (since v26.11.1 a reserved word can still be the unquoted key of a map literal or of a JSON object, eg. `{timeout: 1000}`, see Expressions):

- AFTER
- AND
- AS
- ASC
- BATCH
- BEFORE
- BETWEEN
- BREADTH_FIRST
- BY
- BUCKET
- CONSOLE
- CONTAINS
- CONTAINSALL
- CONTAINSKEY

- CONTAINSTEXT
- CONTAINSVALUE
- CREATE
- DEFAULT
- DEFINED
- DELETE
- DEPTH_FIRST
- DESC
- DISTINCT
- DOCUMENT
- EDGE
- ERROR
- FROM
- ILIKE
- IN
- INCREMENT
- INSERT
- INSTANCEOF
- INTO
- IS
- LET
- LIKE
- LIMIT
- MATCH
- MATCHES
- MAXDEPTH
- NOCACHE
- NOT
- NULL
- OR
- PARALLEL
- POLYMORPHIC
- RETRY
- RETURN
- SELECT

- SKIP
- STRATEGY
- TIMEOUT
- TRAVERSE
- UNSAFE
- UNWIND
- UPDATE
- UPSERT
- VERTEX
- WAIT
- WHERE
- WHILE

Base types

Accepted base types in ArcadeDB SQL are:

- **integer numbers**:

Valid integers are

```
(32bit)
1
12345678
-45

(64bit)
1L
12345678L
-45L
```

- **floating point numbers**: single or double precision

Valid floating point numbers are:

```
(double precision, the default)
1.5
12345678.65432
-45.0
0.23D
.23D

(single precision)
1.5F
```

```
-45.0f
```

- **absolute precision, decimal numbers**: like BigDecimal in Java

Convert a string with the `asDecimal()` method (e.g. `'124.4'.asDecimal()`) to explicitly instantiate an absolute precision number.

- **strings**: delimited by `'` or by `"`. Single quotes, double quotes and back-slash inside strings can escaped using a back-slash

Valid strings are:

```
"foo bar"
'foo bar'
"foo \" bar"
'foo \' bar'
'foo \\ bar'
```

- **booleans**: boolean values are case sensitive

Valid boolean values are

```
true
false
```

Boolean value constants are case insensitive, so also `TRUE`, `True` and so on are valid.

- **links**: A link is a pointer to a document in the database

In SQL a link is represented as follows (short and extended notation):

```
#<bucket-id>:<bucket-position>

or

{"@rid": "#<bucket-id>:<bucket-position>"}
```

eg.

```
#12:15

or

{"@rid": "#12:15"}
```

The bracket notation is mandatory inside JSON, as the short notation is not a valid value in JSON.

The extended notation can hold any expression, a parameter included: `WHERE @rid = {"@rid": ?}` selects the record whose RID is bound to the parameter, exactly like `WHERE @rid = ?`. Since v26.10.2 this form is resolved like the others when the statement is planned; earlier versions failed it with `Cannot convert to RID` (or a `NullPointerException` for a parameter) in a `WHERE @rid = ⋯` condition, while accepting it in `@rid IN [⋯]`.

- **null**: case insensitive (for consistency with IS NULL and IS NOT NULL conditions, that are case insensitive)

Valid null expressions include

```
NULL
null
Null
nUll
...
```

Numbers

ArcadeDB can store five different types of numbers

- Integer: 32bit signed
- Long: 64bit signed
- Float: decimal 32bit signed
- Double: decimal 64bit signed
- BigDecimal: absolute precision

Integers are represented in SQL as plain numbers, eg. `123`. If the number represented exceeds the Integer maximum size (see Java java.lang.Integer `MAX_VALUE` and `MIN_VALUE`), then it's automatically converted to a Long.

When an integer is saved to a schemaful property of another numerical type, it is automatically converted.

Longs are represented in SQL as numbers with `L` suffix, eg. `123L` (L can be uppercase or lowercase). Plain numbers (without L prefix) that exceed the Integer range are also automatically converted to Long. If the number represented exceeds the Long maximum size (see Java java.lang.Long `MAX_VALUE` and `MIN_VALUE`), then the statement fails with a parsing error (`Invalid integer`);

Integer and Long numbers can be represented in base 10 (decimal), 8 (octal) or 16 (hexadecimal):

- decimal: `["-"] ("0" | ( ("1"-"9") ("0"-"9")* ) ["l"|"L"]`, eg.
- `15`, `15L`
- `-164`
- `999999999999`
- octal: `["-"] "0" ("0"-"7")+ ["l"|"L"]`, eg.

- `01`, `01L` (equivalent to decimal 1)
- `010`, `010L` (equivalent to decimal 8)
- `-065`, `-065L` (equivalent to decimal -53)
- hexadecimal: `["-"] "0" ("x"|"X") ("0"-"9"," a"-"f", "A"-"F")+ ["l"|"L"]`, eg.
- `0x1`, `0X1`, `0x1L` (equivalent to 1 decimal)
- `0x10` (equivalent to decimal 16)
- `0xff`, `0xFF` (equivalent to decimal 255)
- `-0xff`, `-0xFF` (equivalent to decimal -255)

an unquoted number written with a leading zero is octal wherever SQL reads a numeric value: in expressions, in the values written by `INSERT` and `UPDATE`, in `LIMIT` and `SKIP`, and in ranges such as `list[01..010]`. So `SET code = 010` stores 8 and `LIMIT 010` returns 8 records. Write the number without the leading zero, or quote it as a string, to keep it decimal. Numbers that name or size something stay decimal whatever their spelling: bucket ids (`bucket:010` is bucket 10), the parts of a RID (`#12:010` is `#12:10`), the `$N` parameters and the counts and sizes of DDL statements (`BUCKETS 010` creates 10 buckets). Before version 26.12.1 a leading zero was ignored and every such number was read as decimal.

Float numbers are represented in SQL as `[-][<number>].<number>F` (F can be uppercase or lowercase), eg. valid Float values are `1.5f`, `-1567.0F`, `.556767F`. If the number represented exceeds the Float maximum size (see Java java.lang.Float `MAX_VALUE` and `MIN_VALUE`), then the result is infinity.

Double numbers are represented in SQL as `[-][<number>].<number>[D]` (D can be uppercase or lowercase and is optional), eg. valid Double values are `1.5`, `-1567.0`, `.556767`, `1.5d`, `-1567.0D`, `.556767D`. A floating point literal written without a suffix is a Double: this is what `0.05` means in a comparison against a `DOUBLE` or `DECIMAL` property, and writing it as a single precision Float instead would move it to 0.05000000074505806 and silently exclude every record holding exactly 0.05. If the number represented exceeds the Double maximum size (see Java java.lang.Double `MAX_VALUE` and `MIN_VALUE`), then the result is `NULL`

a suffix-less literal with 18 or more significant digits that a Double cannot hold exactly (for example `12345678901234567890.123456789012345678`) is kept as an exact `DECIMAL` value, so it is stored and compared without losing digits. Up to 17 digits it stays a Double. A comparison between a `DOUBLE` property and such a literal is exact, and the result of `doubleProp * <long literal>` is a `DECIMAL`. Add the `D` suffix to force a Double.

Float and Double numbers can be represented as decimal, decimal with exponent, hexadecimal and hexadecimal with exponent. Here is the full syntax:

```
FLOATING_POINT_LITERAL: ["-"] ( <DECIMAL_FLOATING_POINT_LITERAL> |
<HEXADECIMAL_FLOATING_POINT_LITERAL> )

DECIMAL_FLOATING_POINT_LITERAL:
```

```
      (["0"-"9"])+ "." (["0"-"9"])* (<DECIMAL_EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<DECIMAL_EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <DECIMAL_EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<DECIMAL_EXPONENT>)? ["f","F","d","D"]

DECIMAL_EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+

HEXADECIMAL_FLOATING_POINT_LITERAL:
        "0" ["x", "X"] (["0"-"9","a"-"f","A"-"F"])+ (".")? <HEXADECIMAL_EXPONENT>
(["f","F","d","D"])?
      | "0" ["x", "X"] (["0"-"9","a"-"f","A"-"F"])* "." (["0"-"9","a"-"f","A"-"F"])+
<HEXADECIMAL_EXPONENT> (["f","F","d","D"])?

HEXADECIMAL_EXPONENT: ["p","P"] (["+","-"])? (["0"-"9"])+
```

Eg. - base 10 - `0.5` - `0.5f`, `0.5F`, `2f` (ATTENTION, this is NOT hexadecimal) - `0.5d`, `0.5D`, `2D` (ATTENTION, this is NOT hexadecimal) - `3.21e2d` equivalent to `3.21 * 10^2 = 321` - base 16 - `0x3p4d` equivalent to `3 * 2^4 = 48` - `0x3.5p4d` equivalent to `3.5(base 16) * 2^4`

BigDecimal in ArcadeDB is represented as a Java BigDecimal. The instantiation of BigDecimal can be done explicitly, using the `asDecimal()` method, eg. `'124.4'.asDecimal()`

Mathematical operations

Mathematical Operations with numbers follow these rules:

- Operations are calculated from left to right, following the operand priority.
- When an operation involves two numbers of different type, both are converted to the higher precision type between the two.

Eg.

```
15 + 20L = 15L + 20L     // the 15 is converted to 15L

15L + 20 = 15L + 20L     // the 20 is converted to 20L

15 + 20.3 = 15.0 + 20.3     // the 15 is converted to 15.0

15.0 + 20.3D = 15.0D + 20.3D     // the 15.0 is converted to 15.0D
```

an Integer operation that exceeds the Integer range is promoted to Long (`2147483647 + 1` returns `2147483648`, where Java would wrap around), while a Long operation that exceeds the Long range fails with an `ArithmeticErrorException: long overflow` (Java would wrap around silently).

The conversion of a string to BigDecimal can be done explicitly, using the `asDecimal()` method, eg. `'124.4'.asDecimal()`

Collections

ArcadeDB supports one type of collection:

- **Lists**: ordered, allow duplicates

The SQL notation allows to create `Lists` with square bracket notation, eg.

```
[1, 3, 2, 2, 4]
```

For OrientDB compatibility, the `.asSet()` method is available to remove duplicates from a `List`:

```
[1, 3, 2, 2, 4].asSet() = [1, 3, 2, 4] -- The order of the elements in the resulting
set is not guaranteed
```

Binary data

ArcadeDB can store binary data (byte arrays) in document fields. There is no native representation of binary data in SQL syntax, insert/update a binary field you have to use `decode(<base64string>, "base64")` function.

To obtain the base64 string representation of a byte array, you can use the function `encode(<byteArray>, "base64")`

Expressions

Expressions can be used as:

- single projections
- operands in a condition
- items in a GROUP BY
- items in an ORDER BY
- right argument of a LET assignment

Valid expressions are:

- `<base type value>` (string, number, boolean)
- `<field name>`
- `<@attribute name>`
- `<function invocation>`
- `<expression> <binary operator> <expression>`: for operator precedence, see below table.
- `<unary operator> <expression>`
- `( <expression> )`: expression between parenthesis, for precedences
- `( <query> )`: query between parenthesis
- `[ <expression> (, <expression>)* ]`: a list, an ordered collection that allows duplicates, eg. `["a", "b", "c"]`)

- `{ <expression>: <expression> (, <expression>: <expression>)* }`: the result is a map, with `<field>:<value>` values, eg. `{"a":1, "b": 1+2+3, "c": foo.bar.size() }`. The key name is converted to a string if it is not. A key can be written unquoted, and every SQL keyword is a valid unquoted key, eg. `{parallel: true, timeout: 1000, unique: true, index: 'x', select: 1}`, also without a space after the colon (`{bucket:1}`, `{schema:x}`) *(since v26.11.1; before, 71 keywords, among them* `PARALLEL`, `TIMEOUT`, `UNIQUE`, `STRATEGY`, `BATCH`, `INDEX`, `FROM` *and* `SELECT`*, were a syntax error as unquoted keys and had to be quoted)*
- `<expression> <modifier> ( <modifier> )*`: a chain of modifiers (see below)
- `<json>`: is translated to a map, nested JSON is allowed and is translated to nested maps
- `<expression> IS NULL`: check for null value of an expression
- `<expression> IS NOT NULL`: check for non null value of an expression

Modifiers

A modifier can be - a dot-separated field chain, eg. `foo.bar`. Dot notation is used to navigate relationships and document fields. eg.

```
john = {
          name: "John",
          surname: "Jones",
          address: {
             city: {
                name: "London"
             }
          }
       }

john.address.city.name = "London"
```

- a method invocation, eg. `foo.size()`.

Method invocations can be chained, eg. `foo.toLowerCase().substring(2, 4)`

- a square bracket filter, eg. `foo[1]` or `foo[name = 'John']`

Square bracket filters

Square brackets can be used to filter collections or maps.

`field[ ( <expression> | <range> | <condition> ) ]`

Based on what is between brackets, the square bracket filtering has different effects:

- `<expression>`: If the expression returns an Integer or Long value (i), the result of the square bracket filtering is the i-th element of the collection/map. If the result of the expression (K) is not a number, the filtering returns the value corresponding to the key K in the map field. If the field is not a collection/map, the square bracket filtering returns `null`. The result of this filtering is ALWAYS a single value.

- `<range>`: A range is something like `M..N` or `M⋯N` where M and N are integer/long numbers, eg. `fieldName[2..5]`. The result of range filtering is a collection that is a subet of the original field value, containing all the items from position M (included) to position N (excluded for `..`, included for `⋯`). Eg. if `fieldName = ['a', 'b', 'c', 'd', 'e']`, `fieldName[1..3] = ['b', 'c']`, `fieldName[1⋯3] = ['b', 'c', 'd']`. Ranges start from `0`. The result of this filtering is ALWAYS a list (ordered collection, allowing duplicates). If the original collection was ordered, then the result will preserve the order.
- `<condition>`: A normal SQL condition, that is applied to each element in the `fieldName` collection. The result is a sub-collection that contains only items that match the condition. Eg. `fieldName = [{foo = 1},{foo = 2},{foo = 5},{foo = 8}]`, `fieldName[foo > 4] = [{foo = 5},{foo = 8}]`. The result of this filtering is ALWAYS a list (ordered collection, allowing duplicates). If the original collection was ordered, then the result will preserve the order.

Nested projections

Colon prefixed curly braces can be used to project maps or JSON documents.

`map:{ <item> [, <item>]* }`, where each `<item>` is one of:

- `*`: every property of the value
- `[!]<identifier>*`, optionally followed by a nested projection: every property whose name starts with the prefix (or, with `!`, excludes them)
- `[!]<expression>`, optionally followed by a nested projection and `AS <identifier>`

As for projections in the `SELECT` statement, nested projection consist of a comma separated list of projections. A prefix wildcard takes no alias, since one alias cannot name several properties, and it goes only after a plain property name: `{se* AS k}` and `{a.b*}` (or `{!a.b*}`) fail with a parsing error. *(Since v26.10.2)* See Projections for examples.

Conditions

A condition is an expression that returns a boolean value.

An expression that returns something different from a boolean value is always evaluated to `false`.

Comparison Operators

- `=` **(equals)**: If used in an expression, it is the boolean equals (eg. `select from Foo where name = 'John'`. If used in an SET section of INSERT/UPDATE statements or on a LET statement, it represents a variable assignment (eg. `insert into Foo set name = 'John'`). A list on the right side is compared as a whole, so `name = ['a', 'b']` does not match a `name` of `'a'`; use `IN` to match any of several values *(since v26.10.2)*
- `==` **(equals)**: same as `=`
- `<=>` **(null-safe equals)**
- `!=` **(not equals)**: inequality operator.
- `<>` **(not equals)**: same as `!=`
- `>` **(greater than)**

- `>=` **(greater or equal)**
- `<` **(less than)**
- `<=` **(less or equal)**

Math Operators

- `+` **(plus)**: addition if both operands are numbers, string concatenation (with string conversion) if one of the operands is not a number. The order of calculation (and conversion) is from left to right, eg `'a' + 1 + 2 = 'a12'`, `1 + 2 + 'a' = '3a'`. If one of the operands is null (or a missing field) and the other is a number, a date/time value, a duration or null, the addition will evaluate to null, eg `1 + null = null`. Concatenating to a string, a list or a map ignores a null, eg `'a' + null = 'a'`. It can also be used as a unary operator (no effect).
- `-` **(minus)**: subtraction between numbers. If one of the operands is null (or a missing field) and the other is a number, a date/time value, a duration or null, the subtraction will evaluate to null, eg `1 - null = null`. Minus can also be used as a unary operator, to invert the sign of a number, and `-null` is null. Between a date/time value and another date/time value, `-` returns the `Duration` between the two. A list or a map minus null is returned unchanged.

Since v26.11.1, `` ` and `-` `` with a null number or date/time operand evaluate to null, as `` `*` ``, `` `/` `` and `` `%` `` always did and as in standard SQL. Before, the null was treated as zero (`` `1 - null = 1` ``, `` `-null = 0` ``) and a date/time value minus null returned the date/time value, so `` `WHERE age - 4 < 6` `` returned the records without `` `age` ``. Use `` `ifnull()` `` (or `` `??` ``) to keep the old result, eg `` `ifnull(discount, 0) + 1` ``. The `` `= `` and `-=` assignments still start a missing or null field from zero, see below.

- `*` **(multiplication)**: multiplication between numbers. If one of the operands is null, the multiplication will evaluate to null.
- `/` **(division)**: division between numbers. If one of the operands is null, the division will evaluate to null. Integer, long and decimal division (and modulo) by zero fail with an `ArithmeticErrorException`, while a double division by zero returns `Infinity`.

  Integer division that does not come out even produces a floating point result, so `10 / 4` is `2.5`, not `2`.

  When either operand is a `DECIMAL` or a big integer, the division is exact: the result keeps as many decimals as it needs, and never fewer than the left operand had. So `1 / 2` is `0.5` and `10.00 / 2.00` is `5.00`. A division with no exact result, such as `1 / 3`, is rounded to 34 significant digits.

- `%` **(modulo)**: modulo between numbers. If one of the operands is null, the modulo will evaluate to null.
- `>>` **(bitwise right shift)**: shifts bits on the right operand by a number of positions equal to the right operand. Eg. `8 >> 2 = 2`. Both operands have to be Integer or Long values, otherwise the result will be null.
- `>>>` **(unsigned bitwise right shift)** The same as `>>`, but with negative numbers it will fill with `1` on the left. Both operands have to be Integer or Long values, otherwise the result will be null.
- `[` **(bitwise right shift)** shifts bits on the left, eg. `2 [ 2 = 8`. Both operands have to be Integer or

Long values, otherwise the result will be null.

- `&` **(bitwise AND)** executes a bitwise AND operation. Both operands have to be Integer or Long values, otherwise the result will be null.
- `|` **(bitwise OR)** executes a bitwise OR operation. Both operands have to be Integer or Long values, otherwise the result will be null *(since v26.11.1 a null operand also gives null; before, it was treated as zero)*.
- `^` **(bitwise XOR)** executes a bitwise XOR operation. Both operands have to be Integer or Long values, otherwise the result will be null *(since v26.11.1 a null operand also gives null; before, it was treated as zero)*.
- `||`: array concatenation (see below for details).

Math Operators precedence

type	Operators
multiplicative	`* / %`
additive	`+ -`
shift	`[ >> >>>`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
array concatenation	`||`

Math + Assign operators

These operators can be used in UPDATE statements to update and set values. The semantics is the same as the operation plus the assignment, eg. `a = 2` is just a shortcut for `a = a + 2`, with one difference: `=` and `-=` start a missing or null field from zero (or, for a string or a collection, from nothing), so `SET counter += 1` sets `counter` to 1 on a record without it, and `SET counter -= 1` sets it to -1, while `SET counter = counter + 1` sets it to null. A null on the right is likewise nothing to add: `SET counter += null` leaves `counter` as it is.

- `+=` **(add and assign)**: adds right operand to left operand and assigns the value to the left operand. Returns the final value of the left operand. If one of the operands is not a number, then this operator acts as a `concatenate string values and assign`
- `-=` **(subtract and assign)**: subtracts right operand from left operand and assigns the value to the left operand. Returns the final value of the left operand
- `*=` **(multiply and assign)**: multiplies left operand and right operand and assigns the value to the left operand. Returns the final value of the left operand
- `/=` **(divide and assign)**: divides left operand by right operand and assigns the value to the left operand. Returns the final value of the left operand
- `%=` **(modulo and assign)**: calculates left operand modulo right operand and assigns the value to the left operand. Returns the final value of the left operand

Array concatenation

The `||` operator concatenates two arrays.

```
[1, 2, 3] || [4, 5] = [1, 2, 3, 4, 5]
```

If one of the elements is not an array, then it's converted to an array of one element, before the concatenation operation is executed

```
[1, 2, 3] || 4 = [1, 2, 3, 4]

1 || [2, 3, 4] = [1, 2, 3, 4]

1 || 2 || 3 || 4 = [1, 2, 3, 4]
```

To add an array, you have to wrap the array element in another array:

```
[[1, 2], [3, 4]] || [5, 6] = [[1, 2], [3, 4], 5, 6]

[[1, 2], [3, 4]] || [[5, 6]] = [[1, 2], [3, 4], [5, 6]]
```

The result of an array concatenation is always a List (ordered and with duplicates). The order of the elements in the list is the same as the order in the elements in the source arrays, in the order they appear in the original expression.

To transform the result of an array concatenation in a Set (remove duplicates), just use the `.asSet()` method

```
[1, 2] || [2, 3] = [1, 2, 2, 3]

([1, 2] || [2, 3]).asSet() = [1, 2, 3]
```

Specific behavior of NULL

Null value has no effect when applied to a `||` operation. eg.

```
[1, 2] || null = [1, 2]

null || [1, 2] = [1, 2]
```

To add null values to a collection, you have to explicitly wrap them in another collection, eg.

```
[1, 2] || [null] = [1, 2, null]
```

Boolean Operators

- `AND`: logical AND
- `OR`: logical OR
- `NOT` (unary): logical NOT
- `CONTAINS`: checks if the left collection contains the right element. The left argument has to be a collection, otherwise it returns FALSE. It's NOT the check of collection intersections, so `['a', 'b', 'c'] CONTAINS ['a', 'b']` will return FALSE, while `['a', 'b', 'c'] CONTAINS 'a'` will return TRUE. With a condition on the right, it is true if the condition holds for at least one element of the left collection, and the condition is evaluated against that element rather than against the row being filtered, eg. `SELECT FROM Order LET $items = out('Contains') WHERE $items CONTAINS (price > 100)` returns the orders having at least one item over 100. The condition can itself use a sub-query, eg. `$items CONTAINS (@rid IN (SELECT item FROM Discounted))`.
- `IN`: the same as CONTAINS, but with inverted operands.
- `CONTAINSKEY`: for maps, the same as for CONTAINS, but checks on the map keys
- `CONTAINSVALUE`: for maps, the same as for CONTAINS, but checks on the map values
- `CONTAINSANY`: for collections, true if the left collection contains at least one element of the right collection (or, with a condition on the right, if the condition holds for at least one element of the left collection)
- `CONTAINSALL`: for collections, true if the left collection contains every element of the right collection (or, with a condition on the right, if the condition holds for every element of the left collection)
- `CONTAINSTEXT`: for strings, checks if a string contains another string (meant for indexed fields).
- `LIKE`: for strings, checks if a string contains another string. `%` is used as a wildcard, eg. `'foobar CONTAINS '%ooba%''`
- `ILIKE`: for strings, checks if a string contains another string disregarding case. `%` is used as a wildcard, eg. `'FOOBAR CONTAINS '%ooba%''`
- `IS NULL` (unary): returns TRUE if a field is `null`
- `IS NOT NULL` (unary): returns TRUE if a field is not `null`
- `IS DEFINED` (unary): returns TRUE is a field is defined in a document
- `IS NOT DEFINED` (unary): returns TRUE is a field is not defined in a document
- `BETWEEN - AND` (ternary): returns TRUE is a value is between two values, eg. `5 BETWEEN 1 AND 10`
- `MATCHES`: checks if a string matches a regular expression
- `INSTANCEOF`: checks the type of a value, the right operand has to be the a String representing a type name, eg. `father INSTANCEOF 'Person'`

12.4. Graph OLAP Engine

The Graph OLAP Engine maintains a read-optimized, columnar representation of your graph alongside the live OLTP data. It uses **Compressed Sparse Row (CSR)** encoding and flat primitive arrays to deliver 5x–400x speedups on analytical workloads — multi-hop traversals, graph

algorithms, and property aggregations — without sacrificing transactional safety.

12.4.1. Why Graph OLAP?

ArcadeDB's OLTP engine is optimized for point lookups and ACID transactions. Analytical workloads — PageRank, community detection, multi-hop traversals — access millions of edges in tight loops. The row-oriented, pointer-chasing nature of OLTP storage causes cache misses, object overhead, and GC pressure.

The OLAP engine solves this by encoding graph topology as flat `int[]` arrays and properties as typed columns:

- **Sequential memory access** — cache-line friendly, no pointer chasing
- **Zero object allocation** — no GC pressure during traversal
- **SIMD-friendly** — enables JVM vectorized operations
- **9x more compact** — flat arrays vs. Java object overhead

12.4.2. Graph Analytical View (GAV)

A Graph Analytical View is a named, schema-persisted OLAP snapshot of selected vertex types, edge types, and properties.

```
GraphAnalyticalView gav = GraphAnalyticalView.builder(database)
    .withName("social")
    .withVertexTypes("Person", "Company")
    .withEdgeTypes("FOLLOWS", "WORKS_AT")
    .withProperties("name", "age", "status")
    .withUpdateMode(UpdateMode.SYNCHRONOUS)
    .build();
```

Named views are persisted in `schema.json` and automatically restored on database restart. A `READY` view's CSR snapshot itself can also be persisted to disk at a clean database close and reused on the next open without rescanning the graph; see CSR Persistence.

12.4.3. SQL

Creating a view

```
CREATE GRAPH ANALYTICAL VIEW social
  VERTEX TYPES (Person, Company)
  EDGE TYPES (FOLLOWS, WORKS_AT)
  PROPERTIES (name, age, status)
  UPDATE MODE SYNCHRONOUS
```

All clauses after the view name are optional. A minimal view covering the entire graph:

```
CREATE GRAPH ANALYTICAL VIEW fullGraph
```

Choosing the vertex properties

Without a `PROPERTIES` clause the view copies every vertex property. `PROPERTIES (a, b)` copies only the listed ones, and a list of names prefixed with `!`, quoted with backticks, copies every property except those: `PROPERTIES (`!Body`, `!Text`)`. Use one form or the other: when a list mixes them, the exclusions apply and the plain names are ignored.

(Since v26.10.2) an empty list, `PROPERTIES ()`, copies no vertex property: the view holds only the graph's structure, which is all that algorithms such as `algo.pagerank()` or `algo.wcc()` need, and it builds faster and takes less memory.

```
CREATE GRAPH ANALYTICAL VIEW topology
  PROPERTIES ()
  UPDATE MODE OFF
```

Use `IF NOT EXISTS` to avoid errors if the view already exists:

```
CREATE GRAPH ANALYTICAL VIEW IF NOT EXISTS social
  VERTEX TYPES (Person)
  EDGE TYPES (FOLLOWS)
  UPDATE MODE SYNCHRONOUS
```

You can also materialize edge properties (e.g., weights):

```
CREATE GRAPH ANALYTICAL VIEW weighted
  VERTEX TYPES (City)
  EDGE TYPES (ROAD)
  EDGE PROPERTIES (distance, toll)
  UPDATE MODE SYNCHRONOUS
  COMPACTION THRESHOLD 50000
```

An edge property is materialized only when the schema declares it on at least one of the view's edge types, with a type a column can hold (`INTEGER`, `SHORT`, `BYTE`, `LONG`, `FLOAT`, `DOUBLE` or `STRING`) and the same column type on every edge type that declares it (`INTEGER`, `SHORT` and `BYTE` share one, as do `FLOAT` and `DOUBLE`). Declare it before creating the view:

```
CREATE PROPERTY ROAD.distance DOUBLE
```

Since v26.12.1 `CREATE GRAPH ANALYTICAL VIEW` (and the builder's `build()` and `buildAsync()`) refuses a definition whose `EDGE PROPERTIES` or `CCH` weights name a property that would get no column, and the error names the property and the `CREATE PROPERTY` to run. Before, the view was created and listed the property in `edgePropertyFilter`, but held no value of it, so every algorithm weighted by it

read the edge records and ran no faster than without a view (issue #12177). A saved view whose property has no column (created by an earlier release, or after a `DROP PROPERTY`) is still restored as it was defined when the database opens, with the same message logged as a warning: declare the property, then `REBUILD GRAPH ANALYTICAL VIEW`.

Add `CCH (⋯)` to keep a Customizable Contraction Hierarchy per weight property, for fast point-to-point shortest paths. A weight listed there is materialized as an edge property even when `EDGE PROPERTIES` does not name it:

```
CREATE GRAPH ANALYTICAL VIEW roads
  EDGE TYPES (ROAD)
  UPDATE MODE SYNCHRONOUS
  CCH (distance, travelTime)
```

Edge sub-types

Listing an edge type in `EDGE TYPES` also materializes its sub-types, and a hop on that type answers with the sub-types' edges too, as it does without a view. The view keeps one slice per concrete type, so the order the types are listed in does not matter. Views persisted by an earlier release are rebuilt once.

Vertex sub-types

Since v26.11.1 listing a vertex type in `VERTEX TYPES` also holds the vertices of its sub-types, including sub-types created after the view was built, as a query on that type reads them without a view. A view over `V` with `W EXTENDS V` therefore covers both, and serves queries once `W` holds vertices. Before, a view held the listed types only, and once a sub-type held a vertex the view no longer covered every vertex, so the planner never used it (issue #10208). A CSR persisted by an earlier release is rebuilt once on the first open.

A type that already holds vertices and is attached to a listed type later (`ALTER TYPE Y SUPERTYPE +V`) holds vertices the view never read: the view does not use them until it is rebuilt, which it does by itself in **SYNCHRONOUS** and **ASYNCHRONOUS** mode, while an **OFF** view turns STALE.

The same holds for an edge type attached to a listed edge type later (`ALTER TYPE G SUPERTYPE +E`) when it already held edges between the view's vertices, lightweight ones included: a hop on `E` reads the edges of its sub-types, so the view is not used for any query until it has rebuilt (an **OFF** view turns STALE and is not used even with `useWhenStale`). An edge type that already existed when the view was built or restored is held only once a scan has read it, even when it looks empty, because a transaction open at that moment may still commit edges of it: attaching it rebuilds the view once. An edge type created after that, and not written while it was not covered, is used at once, as is a sub-type created after the build. Before v26.11.1 such a view answered hops on `E` without the edges of `G` (issue #10214).

The reverse is handled the same way since v26.12.1: when an edge sub-type the view holds leaves the listed types, because it is dropped (`DROP TYPE G UNSAFE`), detached (`ALTER TYPE G SUPERTYPE -E`) or renamed (`ALTER TYPE G NAME G2`), the view is not used for any query until it has rebuilt, which it does by itself in **SYNCHRONOUS** and **ASYNCHRONOUS** mode, while an **OFF** view turns STALE and is not used even with `useWhenStale`. In the meantime queries read the records, and

`schema:graphAnalyticalViews` reports the view STALE. Before, the view stayed READY with the old `edgeCount`: an openCypher `count(*)` on `E` could still count the edges of a dropped or detached sub-type, and a hop on a renamed sub-type found none of its edges until a restart (issues #12032 and #12039).

Since v26.12.1 the view tells a sub-type by the buckets it owned when the view read it, not by its name alone, so the same holds when another type takes the name before the view is read again: a type dropped and created again under the same name (`DROP TYPE G UNSAFE; CREATE EDGE TYPE G EXTENDS E`) or a rename swap (`ALTER TYPE G NAME G_old; ALTER TYPE H NAME G`). A sub-type detached, written to and attached again (`ALTER TYPE G SUPERTYPE -E`, a `CREATE EDGE G`, `ALTER TYPE G SUPERTYPE +E`) also rebuilds the view, since the view never heard the edge written while `G` was detached. Adding a bucket to a sub-type (`ALTER TYPE G BUCKET +…`) does not. Before, with no read of the view between the statements, the view stayed READY and answered from the slice of the old type (issue #12507). A CSR persisted by v26.11.x is rebuilt once on the first open.

Altering a view

Change the update mode or compaction threshold of an existing view:

```
ALTER GRAPH ANALYTICAL VIEW social UPDATE MODE ASYNCHRONOUS
ALTER GRAPH ANALYTICAL VIEW social COMPACTION THRESHOLD 20000
```

Rebuilding a view

Force a full rebuild of the CSR snapshot:

```
REBUILD GRAPH ANALYTICAL VIEW social
```

A view whose build failed when the database opened is no longer loaded, but its definition is kept: `REBUILD` builds it again from the definition, and reports the error to the caller if the build fails again. Before v26.11.1 `REBUILD` refused such a view as "not loaded", and the only way back was `DROP` and `CREATE` (issue #10368).

Dropping a view

```
DROP GRAPH ANALYTICAL VIEW social
DROP GRAPH ANALYTICAL VIEW IF EXISTS social
```

Types a view lists

A view stores the vertex and edge types it lists, and the edge types of its contraction hierarchies, by name. Like a materialized view does with its source types, it protects them:

- `DROP TYPE` of a type a view lists is refused, naming the view. Drop the view first with `DROP GRAPH ANALYTICAL VIEW`. The type is protected whichever of its names, or aliases, the view or the `DROP TYPE` uses.

- A view whose definition names a type that no longer exists (left by a rename on an earlier version) refuses every `DROP TYPE`, of any type: it cannot be told whether the type being dropped is the one the view meant. The message names the view; drop and create it again. Materialized views and continuous aggregates behave the same way.
- `ALTER TYPE … NAME` of a type a view lists renames it in the view's definition too, and the view is rebuilt from the renamed definition. Until the rebuild completes, queries take the ordinary path, as for a view just created.

Before v26.11.1 both were accepted. The view kept serving a CSR that no longer matched the schema, and the next database open failed to build it and left it unbuilt (issue #10368).

Listing views

```
SELECT FROM schema:graphAnalyticalViews
```

12.4.4. Builder Options

Method	Description	Default
`withName(String)`	Named registration + schema persistence	anonymous
`withVertexTypes(String…)`	Filter to specific vertex types	all
`withEdgeTypes(String…)`	Filter to specific edge types	all
`withProperties(String…)`	Materialize specific vertex properties (names prefixed with `!` are excluded instead); an empty array materializes none	all
`withEdgeProperties(String…)`	Materialize edge properties (e.g., weights); each must be declared in the schema on at least one of the view's edge types, or the build is refused	none
`withUpdateMode(UpdateMode)`	OFF, SYNCHRONOUS, or ASYNCHRONOUS	OFF
`withCompactionThreshold(int)`	Rebuild CSR after N accumulated delta edges	10,000
`withContractionHierarchy(String, String…)`	Keep a contraction hierarchy for a weight property, over the given edge types (all of the view's when none)	none

12.4.5. Async Build for Large Graphs

For large graphs, use `buildAsync()` to avoid blocking the calling thread:

```
GraphAnalyticalView gav = GraphAnalyticalView.builder(database)
    .withName("large-graph")
    .withUpdateMode(UpdateMode.ASYNCHRONOUS)
    .buildAsync();

// Wait for build completion
```

```
boolean ready = gav.awaitReady(30, TimeUnit.SECONDS);
```

12.4.6. Update Modes

The GAV supports three synchronization modes between OLTP and OLAP:

Mode	Behavior	Staleness	Use Case
OFF	Marks view STALE on commit; requires manual rebuild	Until rebuild	Batch analytics, static snapshots
SYNCHRONOUS	Applies an overlay on each commit	Zero	Real-time analytics, consistent reads
ASYNCHRONOUS	Applies an overlay on each commit, off the committing thread	Brief STALE window per commit	Write-heavy workloads, tolerable brief inconsistency

In **SYNCHRONOUS** mode, the engine captures transaction deltas (new/deleted vertices, added/removed edges, property changes) and merges them into an immutable overlay on top of the base CSR. Readers always see a consistent snapshot via an atomic volatile reference swap.

In **ASYNCHRONOUS** mode the same deltas are captured, but merged into the overlay by a background task instead of by the committing thread, so the commit does not wait for the merge; commits that arrive together are merged by one task. Until a commit's delta is merged the view reports STALE: by default it is then not used, so a query (including the next one of the committing thread) reads the records and sees the commit, while with `arcadedb.gavUseWhenStale` the view keeps serving the snapshot as of the previous merge. `awaitReady()` waits for the pending merges. Since v26.11.1; before, every relevant commit started a full rebuild of the CSR whatever the size of the change, and the view served no query until it finished, even with `arcadedb.gavUseWhenStale`, so under a steady stream of small commits almost no query used it (issue #10189). If the merge falls more than 100,000 commits behind, the pending changes are dropped and the view rebuilds once, STALE in the meantime.

Commit order. In both modes the changes reach the overlay in the order their transactions committed, however many threads commit at once. A thread whose commit follows another one on the same records returns from `commit()` once its change is in the view, which is after the change it follows; this is why a commit can wait a moment on a slower one. `arcadedb.gavCommitOrderMaxWait` (30 seconds by default, `0` to not wait) bounds that wait; a transaction that never reports costs the commits behind it one such wait, logged as a WARNING, and is then skipped. Since v26.11.1; before, two commits that conflicted could reach the overlay in the opposite order, and when an edge was created in the record slot a deleted edge had just freed, the deletion arriving second withdrew the new edge from the view for good, so the counts it served were a few edges short of the same queries without the view (issue #10249).

Writes during a build or rebuild. A view starts tracking changes before it scans the graph, so a transaction that commits while the view is being built or rebuilt is never lost: in **SYNCHRONOUS** mode it is applied on top of the finished view (checked against what the scan already read, so nothing is counted twice) in **SYNCHRONOUS** and **ASYNCHRONOUS** modes, and in **OFF** mode the view is published STALE. Writers are not blocked by a running build, including `REBUILD GRAPH`

`ANALYTICAL VIEW`.

A committed change that the view cannot merge, or that a build cannot replay on top of the CSR it just scanned, is not dropped: the view logs a warning, reports itself STALE and rebuilds, so it never keeps answering from a graph that never existed (issue #10338). Before 26.11.1 such a change was dropped with a warning, and a first build that hit one never published.

Before 26.11.1 a **SYNCHRONOUS** or **ASYNCHRONOUS** view could answer wrongly until the database was reopened after three kinds of commit: an edge created in the transaction that deletes one of its endpoints (a negative `algo.degree()`, or a lightweight edge that stayed counted; issues #10340 and #10339), a transaction committed right after `CREATE GRAPH ANALYTICAL VIEW` (issue #10338), and a vertex property that was removed, which kept its old value (issue #10335). Rebuild a view with `REBUILD GRAPH ANALYTICAL VIEW` if it served such commits on an earlier version.

In a cluster. A transaction that reaches a node through replication (a follower applying the leader's commits, or the leader applying a commit made in a transaction opened on a follower) carries no change set the view can merge. Since v26.12.1 such a commit makes the view on that node STALE, so queries read the records and see it. A **SYNCHRONOUS** or **ASYNCHRONOUS** view then rebuilds in the background when a query asks for it, at most once a second (or once per twice its last build time), waiting longer (up to 30 seconds) while new commits keep overtaking the rebuilds; a view nobody queries is not rebuilt. An **OFF** view stays STALE until `REBUILD GRAPH ANALYTICAL VIEW`. Before, a view a node loaded when it opened its database (after a restart, for example) stayed READY and kept answering SQL `out()`/`in()` and openCypher traversals from the graph as of that open, also after the node became leader (issue #12250). A node that has not reopened its database since the view was created has no view (`NOT_BUILT`) and answers from the records.

In **SYNCHRONOUS** and **ASYNCHRONOUS** modes the view being rebuilt keeps serving queries, and keeps absorbing commits, until the rebuild publishes its new CSR: only the caller of `REBUILD GRAPH ANALYTICAL VIEW` (or of `build()`) waits for the scan. The scan no longer runs while holding the view's lock, so neither a committing transaction nor a status check waits behind it (issue #8403); before, the view served during a blocking rebuild lagged the commits until the rebuild finished. A rebuild superseded by a newer one while it scans waits for the newer one and reports its outcome (success or error); a rebuild interrupted by the database closing fails with an error rather than reporting success.

A view created by an earlier version while writes were running can be missing transactions committed during its build (issue #8378): rebuild it if its answers must match the graph exactly. When the overlay accumulates too many changes (configurable threshold, default 10,000 edges), a background compaction rebuilds the full CSR.

Until then the overlay costs whole-graph reads (the Cypher count operators, the `algo.*` procedures) one copy of the adjacency per committed change, made on the first read after the change: the overlay applied to the base CSR, kept with that snapshot for every later read. Before v26.11.1 a view with any overlay sent those reads down the per-vertex path instead (a 2-hop count measured about a quarter slower), from the first committed edge until the compaction (issue #10411).

The copy is made for the reads that pay for it. A read that starts from a handful of vertices - a 2-hop count from one indexed vertex visits some 64 adjacency entries - does not copy the adjacency of the whole graph for itself: a commit replaces the snapshot, so the copy would serve no second read. Such reads walk the vertices one by one, as they did before the overlay, and the copy is made by the read that brings the entries the reads of the snapshot have visited to what the copy costs (a whole-graph read does it at once; point reads do it only when no commit comes first). Since v26.11.1 point reads while writes are committed take about as long as before the copy existed, where they had become three to six times slower at 20 to 100 commits per second (issue #10471).

Lightweight edges and bulk loads. A lightweight edge is tracked like any other edge in every mode: its creation and its deletion reach the overlay in **SYNCHRONOUS** and **ASYNCHRONOUS** modes, and mark an **OFF** view STALE. Each copy of a duplicated lightweight edge counts as an edge of its own, and a delete removes one copy. Copies have no identity to tell an old one from a new one, so when lightweight edges of a pair change while the view is being built or rebuilt, the view checks that pair against the graph before serving it; if a duplicated copy made them disagree, it rebuilds rather than serve the pair one copy off. Edges loaded with GraphBatch (and so by the graph importer, the HTTP batch endpoint and the gRPC batch load) are written in bulk, without an event per edge, so the view learns of them when the batch closes: **OFF** goes STALE, and **SYNCHRONOUS** and **ASYNCHRONOUS** rebuild rather than absorbing millions of edges into the overlay. While that rebuild runs the view is not READY, so queries take the ordinary path and never answer without the load. During the batch itself the view still serves what it held before the batch started.

Before v26.11.1, deleting a vertex that carried a lightweight self-loop reported the self-loop deleted twice, once from each of the vertex's edge lists. A view counted both reports, so its edge count went one lower per such self-loop, down to negative numbers (issue #10369). A vertex delete now reports each self-loop once, record-backed or lightweight, so a delete listener or trigger hears of it once. Rebuild a view that served such deletes on an earlier version.

Before 26.10.2 a view never learned about a lightweight edge created after its build, nor about any edge written by `GraphBatch`: a **SYNCHRONOUS** view answered as if those edges did not exist, and a transaction or batch that touched nothing else left an **OFF** view READY (issue #9572). Rebuild a view on such a graph after upgrading.

12.4.7. CSR Persistence

Available.

A `READY` view (with no pending overlay changes) writes its CSR to disk at a clean database close, alongside a freshness certificate: the database's last committed transaction id at the time the CSR was built. On the next open, if a persisted file plausibly applies (see arcadedb.gavPersistCsr below), the view is marked `READY` immediately, but the file itself is not read yet - see Lazy Restore. That deferred read either:

- finds the database's current last committed transaction id still matches the certificate - nothing

was committed to the database in between, so the persisted CSR is loaded from disk as-is instead of being rebuilt; or

- finds a mismatch - anything was committed in between, to any type, not only the ones the view covers - and falls back to an async rebuild.

This means the speedup only applies to reopening a database that was **not written to** since its last clean close. A database that receives writes between every session still pays the rebuild on every open.

Persistence is controlled by the `arcadedb.gavPersistCsr` database-level setting (default `true`). Set it to `false` to skip writing the CSR file, for example to avoid its disk footprint or the extra write at close.

Serving queries against a stale CSR while a delta is applied in the background (the way `SYNCHRONOUS` mode's overlay already does for live updates) is not part of this: a CSR's contiguous array layout makes partial coverage expensive to retrofit, so an invalidated certificate always means a full rebuild, not a partial one.

Lazy Restore

Available.

A persisted CSR that plausibly applies is not read from disk during `open()`. Reading and deserializing it is a real, if bounded, cost - up to roughly a second at 10M vertices - and a session that opens a database and never queries the view got nothing for paying it. Instead, `open()` marks the view `READY` immediately without touching the file, and the actual read is deferred to whichever of these happens first:

- a query that actually needs the view's data; or
- an explicit call to the view's `awaitReady()` method.

Reading the state of the view is an explicit wait too, so `SELECT FROM schema:graphAnalyticalViews` (and `GraphAnalyticalView.getStats()`) starts the restore and returns at once. Polling the row until `status` is `READY` therefore ends with the counts of the view, `nodeCount` and `edgeCount`, in the row; while the restore runs the row reads `BUILDING`. Before v26.11.1 the row of a view whose restore had not been triggered yet read `READY` with `nodeCount = 0` and `edgeCount = 0`, for as long as no query used the view (issue #10575).

Each of these re-verifies the freshness certificate from scratch at that point (not the one implied at `open()` time), so a commit that lands after `open()` but before the first query is still caught correctly and falls back to a rebuild rather than serving stale data. A session that opens and closes a database without ever touching the view now costs what the no-view baseline costs, not what a restore costs - `close()` does not wait for a restore that was never triggered either.

Since v26.10.2 a query that looks a view up while its restore is in flight (for example the first count push-down, `MATCH` or `shortestPath()` after a reopen) waits for the restore instead of taking the slower record path. The wait is bounded by the `arcadedb.gavQueryRestoreAwaitTimeout` database-level setting (default `5000` ms, `0` disables the wait).

A query that runs while a view is not ready yet (its restore past that wait, its first build still running,

or the view stale and not allowed to serve stale results) takes the record path. Since v26.10.2 that is no longer remembered: the OpenCypher plan cached for the query is planned again as soon as the view is ready, so the following runs of the same query use the view.

The `arcadedb.gavRestoreAwaitTimeout` database-level setting (default `0`) forces the wait at `open()` time instead, for either the deferred restore or a full rebuild: set it to a positive number of milliseconds to trade a slower `open()` for the view being immediately usable by the query that triggered the reopen, rather than the first query racing (and possibly missing) it.

12.4.8. How CSR Works

The graph topology is stored as two pairs of arrays (forward for outgoing edges, backward for incoming):

```
Forward CSR (outgoing edges):
  offsets:   [0, 3, 5, 8, ...]     -- one entry per vertex + sentinel
  neighbors: [1, 5, 7, 2, 6, ...]  -- dense neighbor IDs, contiguous per source

  Outgoing neighbors of vertex v = neighbors[offsets[v] .. offsets[v+1])
  Out-degree of vertex v         = offsets[v+1] - offsets[v]   -- O(1)
```

This layout enables sequential memory access (cache-line friendly) and O(1) degree lookups.

12.4.9. Columnar Property Storage

Properties are stored as typed flat arrays — `int[]`, `long[]`, `double[]`, or dictionary-encoded `int[]` for strings. Each column has a compact null bitmap (1 bit per vertex). Dictionary encoding maps unique string values to integer codes, achieving near-100% compression for low-cardinality fields.

A column takes its type from the schema, or for an undeclared property from the first values the build samples. Since v26.11.1 a value the column cannot give back as it is (a `2.5` or a `NaN` in an integer column, a `FLOAT` in a double one, a number in a string one) is not stored in it: a read of that vertex's property through the view falls back to the record, so queries answer with the value the record holds. Until v26.10.2 it was converted into the column's type, and a query over the view could read `2` where the record held `2.5` (issue #10255).

A property that the view's types declare with different column types (`A.p INTEGER` and `B.p DOUBLE`; `INTEGER`, `SHORT` and `BYTE` share one column type, as do `FLOAT` and `DOUBLE`) gets no column at all, and a warning names it: every read of it through the view goes to the records. Since v26.12.1 this holds whatever the order of the types and however many declare it, and the build no longer samples such a property back into a column. Before, a later type declaring it again (`C.p INTEGER`), or the sampling of undeclared properties, gave it an `INT` column that held no value for the `B` vertices (issue #12229).

Edge Properties and Pending Changes

Edge property columns (`EDGE PROPERTIES (…)`, used as weights by the shortest-path and spanning-tree algorithms) are laid out in the order the graph had when the view was built. In `SYNCHRONOUS` mode a commit is absorbed into a pending-changes overlay rather than rebuilding, so from that

moment the neighbour list and the columns describe two different orderings.

The view reconciles the two: an edge already in the base graph keeps answering from its column, and an edge the overlay added answers from the value recorded when it was committed. Weighted algorithms therefore keep the fast columnar path while a view is being updated.

Two cases remain that the view cannot answer, and for those the algorithm reads the edge records directly - exact, only slower:

- several parallel edges join the same two vertices and only some of them were deleted, since the overlay counts deletions per vertex pair and cannot say which of the parallel edges went;
- a property of an existing edge was changed while another edge of the same type joins the same two vertices, until the rebuild that refreshes the columns completes, since the view cannot tell which of the two changed. Since v26.10.2, a change to an edge that is the only one of its type between its two vertices is kept in the overlay and answered from there, with no rebuild (issue #9437). Adding and deleting edges never have this effect. Since v26.11.1, neither does saving an edge the view does not hold: an edge to a vertex of a type the view does not cover, or an edge whose vertex is deleted in the same transaction (issue #10350).

12.4.10. Memory Usage

The OLAP representation is significantly more compact than the OLTP equivalent:

- **CSR topology**: ~8 bytes per edge (bidirectional)
- **Node ID mapping**: ~8 bytes per vertex
- **Columnar properties**: 4–8 bytes per vertex per column
- **Null bitmaps**: 1 bit per vertex per column

Example: for a graph with 500K vertices and 8M edges, the GAV uses **134.6 MB** compared to an estimated **~1.2 GB** for the OLTP representation — **9.3x more compact**.

```
long bytes = gav.getMemoryUsageBytes();
```

Sizing the Heap for a View

Behavior changed in ArcadeDB v26.11.1.

The view lives on the JVM heap, and so does the page cache (`arcadedb.maxPageRAM`, a quarter of the heap by default). Three things decide the smallest heap a view needs:

- **The build.** Since v26.11.1 a build holds about the size of the CSR it produces, plus one forward slice while it renumbers it, at its peak: the edges are collected per edge type in fixed-size blocks, each forward slice is filled from them and sorted in place, and each backward slice is derived from the forward one. Before v26.11.1 the build held the edges as (source, target) pairs in doubling arrays next to the first CSR and the reordered one, about three to four times the size of the view, so on the LDBC graph `datagen-7_5-fb` (633K vertices, 34M edges) a restore needed a 3 GB heap for a view of about 300 MB (issue #11011).

- **The restore from a persisted CSR.** Since v26.11.1 the file is read and written as a stream of frames of 1 MB, each checksummed and, when the database is encrypted, encrypted on its own, so a clean close and the next open need no more heap than the view itself. Before v26.11.1 both held the whole payload in a buffer besides the view, and in a heap too small for that the clean close silently skipped persisting it and the next open rebuilt the view from a full scan. Files written before v26.11.1 are not read: the first open rebuilds the view and the next clean close writes the new format.
- **The page cache.** Since v26.11.1 the pages a build reads from disk are the first evicted while it runs, and leave the cache when it ends unless another reader used them meanwhile: the view answers what they hold, so keeping them would hold the graph twice. The pages the cache held before the build stay. The `arcadedb.gavBuildReleasesPages` database-level setting (default `true`) turns this off, for a workload that reads the same records through the record API right after the build.

A build or restore that runs out of heap, or fails for any other reason, ends the view in the `FAILED` status (or `STALE`, when it still serves an earlier snapshot), logs the failure at `SEVERE` with the view name, and reports the cause in the `buildError` column of `schema:graphAnalyticalViews`. Before v26.11.1 a build that died of an `OutOfMemoryError` left the view `BUILDING` for good. `REBUILD GRAPH ANALYTICAL VIEW` builds a `FAILED` view again from its definition.

12.4.11. Graph Algorithms

The module includes parallelized graph algorithms that operate directly on CSR arrays with zero GC pressure:

Algorithm	Description
PageRank	Pull-based, parallel, configurable damping factor and iterations
Connected Components	Parallel union-find for weakly connected components
BFS	Breadth-first search with distance arrays
SSSP (Dijkstra)	Single-source shortest path for weighted graphs
Label Propagation	Community detection
Triangle Counting	Count 3-cliques in the graph
Local Clustering Coefficient	Per-vertex clustering coefficients

```
GraphAlgorithms algos = new GraphAlgorithms();

// PageRank (20 iterations, damping 0.85)
double[] ranks = algos.pageRank(gav, 20, 0.85);

// Connected Components
int[] components = algos.connectedComponents(gav);

// BFS from a source vertex
```

```
int[] distances = algos.bfs(gav, sourceNodeId);
```

Algorithms and Pending Changes

In `SYNCHRONOUS` mode a committed change is absorbed into a pending-changes overlay rather than rebuilding the view. The overlay keeps the slot of a deleted vertex and numbers an added vertex above the base graph, so from that moment the view's vertex count and the range of its internal vertex numbers stop being the same figure.

The vertices a view holds are renumbered onto a gapless range before any algorithm sees them, so every `algo.*` procedure answers for exactly the vertices that are live at the time of the call - the added ones included, the deleted ones excluded.

While a view is holding pending changes the whole-graph procedures also stop using the parallel kernels listed above, which read the base CSR arrays directly and know nothing of the overlay: they would otherwise answer for the graph as it stood at the last build. Those calls still run off the view's adjacency, single-threaded rather than parallel, and the compaction that folds the overlay back into the base graph (by default after 10,000 pending edges) restores the parallel path.

Since v26.11.1, each `algo.*` call, and each SQL `shortestPath()` or `bellmanFord()` call, reads one fixed snapshot of the view from start to end, including the rows streamed after the call returns. A rebuild or compaction that completes during the call does not affect it. Before, a rebuild that landed mid-call numbered the vertices afresh while the call was still reading. One row could then come back with a null node, or a row could name the wrong vertex (issue #10350). A long call keeps the snapshot it started on in memory until it ends.

The same holds for a whole statement: every part of a Cypher query (its `MATCH` operators, count push-downs, `CALL` and `COUNT` sub-queries) and of a SQL `SELECT`, `MATCH` or `TRAVERSE` statement (its `MATCH` patterns, `out()`/`in()`/`both()`, `astar()`) reads the snapshot the statement first read, and so does each Gremlin traversal. Before v26.11.1 the Cypher `MATCH` operators read whichever snapshot the view served at each call: a rebuild while the rows streamed made the later rows name other vertices, or, for a fused chain of hops, failed the query with "was rebuilt while the query walked it" (issue #10415). Each statement of a script takes its own snapshot, so a statement that follows a `COMMIT` reads what was committed.

12.4.12. Shortest Paths with Contraction Hierarchies (CCH)

A view can keep a **Customizable Contraction Hierarchy** for a weight property: a precomputed structure that answers point-to-point shortest paths in a time that depends on the depth of the hierarchy rather than on how far apart the two vertices are. It is meant for road, logistics, utility and other infrastructure networks that serve many routing queries.

```
CREATE GRAPH ANALYTICAL VIEW roads EDGE TYPES (ROAD) UPDATE MODE SYNCHRONOUS CCH
(distance)
```

```
GraphAnalyticalView roads = GraphAnalyticalView.builder(database)
    .withName("roads")
```

```
    .withEdgeTypes("ROAD")
    .withUpdateMode(UpdateMode.SYNCHRONOUS)
    .withContractionHierarchy("distance")
    .build();
```

Queries go through algo.cch.shortestPath (Cypher) or cchShortestPath() (SQL). A query is answered by the hierarchy when its weight property and relationship types are exactly those of a hierarchy on a ready view; otherwise, and whenever the hierarchy cannot answer for the view's current state, it is answered by bidirectional Dijkstra. Answers are exact either way, and a transaction holding uncommitted changes always sees them.

How it works

1. **Order** - a nested-dissection order of the vertices, computed from the topology alone: small vertex separators (minimum vertex cuts) split the graph recursively, and each separator is ranked above the pieces it separates.
2. **Contraction** - eliminating the vertices in that order adds shortcut edges, giving the chordal supergraph of the hierarchy.
3. **Customization** - the weight of every edge and shortcut, computed bottom-up from the edge weights. This is the only step that depends on the weights.
4. **Query** - the two endpoints climb the hierarchy and meet at the top; shortcuts are then unpacked into the edges they stand for.

On grids, the hardest case for this technique, a query on 360,000 vertices takes about 0.5 ms for a route across the whole grid against about 50 ms for Dijkstra, and the gap grows with the size of the graph. Road networks have much smaller separators than grids and benefit more.

Keeping up with changes

The hierarchy follows the view: every change the view absorbs is followed by a background preparation, on the view's build executor. In SYNCHRONOUS mode most changes are caught up in milliseconds:

- A weight change, a closed road (an infinite weight) or a deleted edge re-customizes only the part of the hierarchy above the edges that changed. The view keeps its CSR: the new weight of an edge that is the only one of its type between its two vertices is served from the view's change overlay, without rebuilding anything.
- A new edge between two vertices the hierarchy already joins is handled the same way.
- A new edge between two vertices it does not join, or a new vertex with edges, needs a new contraction. The existing vertex order is kept, with new vertices ranked first, so this costs a contraction and a customization rather than a full build.
- A weight change on an edge that has a parallel twin of the same type between the same two vertices makes the view rebuild its CSR in the background, because the view cannot tell which of the two changed. The hierarchy then catches up from the rebuilt view.
- While a preparation is running, queries are answered by bidirectional Dijkstra, so an answer never reflects an older state than the view itself.

- Directed (`OUT`/`IN`) queries are prepared from the start. Undirected (`BOTH`) ones need a second metric, customized the first time a `BOTH` query asks for it, so that first query and any during its customization are answered by Dijkstra.

Measured on the California road network (DIMACS `USA-road-d.CAL`: 1.9M vertices, 4.7M arcs) through a database: a query takes 0.04 to 0.1 ms at any distance, against 18 to 230 ms for Dijkstra; updating 1, 10 and 100 weights takes about 0.7, 8 and 65 ms from commit until the hierarchy answers for it; closing a road about 4 ms; adding a road the hierarchy does not join about 0.9 s; the first build about 13 s. Each query needs a few tens of KB of scratch, sized to the depth of the hierarchy rather than to the graph.

The order is persisted next to the view's CSR at a clean close (see CSR Persistence) and reused when the database is reopened with nothing committed in between, so a reopen only contracts and customizes.

Suitability

The technique relies on small separators. Graphs without them, such as social graphs or graphs with supernodes, would need a supergraph many times larger than themselves. A hierarchy that would exceed `arcadedb.gavCchMaxArcsPerEdge` supergraph arcs per edge (default 16) is not built: it reports `UNSUITABLE`, and queries keep using bidirectional Dijkstra. A `REBUILD GRAPH ANALYTICAL VIEW` tries again.

Monitoring

`SELECT FROM schema:graphAnalyticalViews` and `GraphAnalyticalView.getStats()` list each hierarchy with its status (`PREPARING`, `READY`, `UNSUITABLE`, `UNAVAILABLE`), size in arcs and memory, how many topologies were built or restored, how many customizations ran, and how many queries it answered or handed back to Dijkstra.

The weight of an edge is the numeric value of its weight property; an edge without one weighs `1`, and an edge whose value is negative is not walked.

12.4.13. Query Planner Integration

The Cypher query planner automatically detects ready GAVs and substitutes OLTP traversal operators with CSR-based operators when:

- A named GAV is registered and in READY state
- The GAV covers the required vertex and edge types
- The query does not return edge variables as first-class records (edges in CSR have no RID; edge **properties** are fully supported)

No query changes are needed — the optimizer transparently accelerates matching traversal patterns.

Relationship uniqueness. Cypher binds a relationship at most once per `MATCH` clause, so `MATCH (a)-[:KNOWS]-(b)-[:KNOWS]-(c)` never walks back to `a` over the edge it just took. A view holds adjacency, not edge records, so a hop that could reuse a relationship of another hop in its clause - they share

an edge type, or either is untyped - is walked one edge type and direction at a time and remembers which relationship it took, and `EXPLAIN` marks it `unique relationships`. Such a clause is served from the view when every one of those hops is anonymous and has a fixed length; a named relationship variable or a variable-length relationship among them sends all of them to the records instead. Hops that cannot collide - `(a)-[:KNOWS]→(b)-[:WORKS_AT]→(c)` - are unaffected. Since v26.11.1 this also holds for an edge type with sub-types: a hop on `KNOWS` with `KNOWS_WELL EXTENDS KNOWS` walks the slice of each concrete type on its own and names a relationship after the type it actually has, so a hop on the parent and a hop on the sub-type recognize the same edge. Before, such a clause walked the edge records (issue #10213).

Views over some of the vertex types. A view built with `VERTEX TYPES (…)` holds only the vertices of those types, so the edges that leave them are missing from its adjacency. Edge types do not declare which vertex types they connect, so the planner cannot know that a traversal stays inside the view: since v26.10.2 a view is used for SQL `out()`, `in()`, `both()`, `MATCH`, `TRAVERSE`, shortest path, the count push-downs and the `algo.*` procedures only when it covers every vertex type of the database that holds a vertex, and these statements otherwise read the records. Before that they answered with the covered part of the graph, for example 0 rows for `MATCH (b:B)-[:E]→(c:C)` on a view over `(A, B)` (issue #9301). A type with no vertex of its own does not count, because no traversal can reach it: a parent type whose records all live in the sub-types the view lists (`Message` over `Post` and `Comment`), or a type not used yet. The first vertex created in such a type makes the view partial again, without any schema change. A view over a subset of the types still serves a one-hop Cypher scan whose labels it holds. To accelerate the other statements, list every vertex type that holds vertices.

Stale views. A view that is STALE (in `OFF` mode after a commit, in `ASYNCHRONOUS` mode until a commit is merged, in a cluster after a replicated commit until it rebuilds, or after a failed rebuild) is not used by the planner unless `arcadedb.gavUseWhenStale` is `true`, in which case queries read the snapshot as of its last build. The setting is per database, and applies to every view of that database immediately, including views already built and views restored at open:

```
ALTER DATABASE `arcadedb.gavUseWhenStale` true
```

The builder's `withUseWhenStale(boolean)` overrides it for one view, and that override is saved with the view's definition.

A one-hop pattern feeding an aggregation, such as

```
MATCH (p:Person)-[:KNOWS]->(f:Person)
WHERE p.city = f.city
RETURN p.city AS city, avg(f.age) AS avgAge, count(*) AS n
```

is read entirely from the view: the sources are the view's nodes carrying the source label, both endpoints read their properties from the view's columns, and every edge reaches the aggregation as a lightweight row rather than a copied record. `EXPLAIN` shows it as `GAV ONE-HOP SCAN`. This applies when the statement is a single `MATCH` of one fixed-length relationship feeding an aggregating `RETURN`, with no relationship or path variable and no inline property map, the optimizer would otherwise scan the source label (not seek it through an index), and the view is not stale and covers every vertex type the two labels can match. A count per endpoint alone (`RETURN p.city, count(*)`) keeps

its own, even cheaper, edge-count plan, and an order-sensitive aggregate such as `collect()`, or a grouped `LIMIT`/`SKIP` without `ORDER BY`, keeps the regular plan.

Counts that build no row. A statement whose only output is `count(*)` - and a `COUNT { }` subquery - is answered by a count push-down when its pattern is a chain, a star around one node, or one of the LSQB shapes; `EXPLAIN` shows it under `Using Count Push-Down`. Since v26.10.2 the push-downs also take:

- a property predicate on a node of a chain, a star, the anti-join of LSQB Q8/Q9 or the pair join of LSQB Q2, written inline (`(:Message {kind: 'Post'})`) or as a `WHERE` conjunct that reads that node alone (`WHERE m.kind = 'Post'`, `WHERE m.created > $since`). It is checked once per distinct vertex the position reaches, on the vertex itself, so it costs one record read per such vertex instead of the rows of the whole pattern (issues #9595, #9608). A conjunct that compares two nodes (`WHERE a.x < b.x`), a property value read off another node (`{kind: a.kind}`), `rand()` and pattern predicates still leave the count to the regular plan. A predicate written at one position of a variable holds at every position the variable is written at, and two nodes whose inline maps give one property two different string, boolean or integer values (`(c:Message {kind: 'Comment'})`, `(p:Message {kind: 'Post'})`) are known to be different vertices, which proves their relationships different edges the way two unrelated labels do.
- a `MATCH` made of parts that share no variable, such as `MATCH (a:Person), (b:Person), (c:Person)`: each part is counted on its own, by its own push-down or plan, and the counts are multiplied. An `OPTIONAL MATCH` part counts at least one row. Two parts of one `MATCH` clause are multiplied only when the schema proves their relationships cannot be the same edge, since a clause binds every relationship once. A product that does not fit a 64-bit integer is an error (issue #9596).
- a `WITH` that only passes variables on, `*` and aliases included (`WITH message, creator`, `WITH m AS x`), with no aggregation, `DISTINCT`, `WHERE`, `ORDER BY`, `SKIP` or `LIMIT`: it leaves the number of rows alone, so the statement is counted as if the clauses on both sides were one query part (issue #9597).
- `count(x)` of a node or relationship variable a non-optional `MATCH` binds, which counts the same rows as `count(*)`; a single relationship between two unconstrained nodes, `MATCH ()-[e:KNOWS]→() RETURN count(e)` (untyped, with several types or undirected too), read from the view's per-type totals, or without a view in one walk of the edge lists that loads no edge; and `MATCH (n) RETURN count(n)`, the sum of the vertex types' counters (issue #9600). Unlike the SQL `SELECT count(*) FROM KNOWS`, which counts edge records, the relationship count includes lightweight edges, which keep no record.
- the same pattern however it is written: the comma-separated parts of a `MATCH` are read as the graph they describe, so a chain cut into parts is counted as the chain, and a cycle such as LSQB Q2 (a `KNOWS` hop closed by a three-hop chain, or the same cycle written as four one-hop patterns starting from the post) is split into the probe hop and the chain by the statistics - the fan-out of each hop sampled on the label it leaves - rather than by the order of the text (issue #9599).
- an `OPTIONAL MATCH` only tested for absence (`OPTIONAL MATCH (c)-[h:HAS_TAG]→(t) WITH ⋯ WHERE h IS NULL`), which is planned as `WHERE NOT (c)-[:HAS_TAG]→(t)` and so reaches the anti-join push-down (issue #9598, see OPTIONAL MATCH).

Since v26.11.1 a triangle written as one `MATCH`, `MATCH (a:Person)-[:KNOWS]-(b:Person)-[:KNOWS]-(c:Person)-[:KNOWS]-(a) RETURN count(*)`, is a count push-down too, with no `WHERE` or with one that orders the three variables strictly on one property (`WHERE a.id < b.id AND b.id < c.id`, in any

spelling of `<` and `>` that orders all three). Each triangle is found once by the sorted intersection `algo.triangleCount` uses and weighed by the matches it stands for, so on a 10,000-person, 180,000-edge graph the ordered count takes milliseconds instead of seconds (issue #10255). The answer is the pattern's own, not the algorithm's simple graph: parallel relationships are matches of their own, the comparison is on the property values (two equal values, a null, or values of two types such as a number and a string drop the triangle), and without a `WHERE` the six walks of every triangle and the walks over self loops are counted. The hops must be undirected, anonymous and of one type, and the three variables carry one label or none; `EXPLAIN` shows `COUNT TRIANGLES (CSR sorted intersection)` with the order it counts.

Since v26.12.1 a chain count with an end pinned by a UNIQUE key, such as `MATCH (c:Chunk)-[:OF_RECORD]→(r:Record {external_id: $id}) RETURN count(c)`, looks that vertex up through the index and counts only its own paths, whichever end the key is written on (inline or in `WHERE`, as a literal or a parameter). Its cost no longer grows with the size of the labels; `EXPLAIN` shows `seeded anchor: unique key Record[external_id] = [⋯]` (issue #11594).

Since v26.12.1 any other count anchored by an index lookup (a non-unique key, an `IN` list or a range such as `WHERE p.id < 500`) runs as that index seek plus the expansion from what it finds when the lookup finds few vertices of the label, and as the push-down otherwise. The push-down walks the whole label, while the seek costs only what it finds, so the index entries the lookup finds are counted with the parameters of the execution, without reading a record, up to a cap: a key or an `IN` list is seeked when it finds fewer than one vertex in a hundred of the label, a range when it finds fewer than one in four (issues #11680, #11889). `EXPLAIN` then shows the index seek instead of `COUNT CHAIN PATHS` or `COUNT STAR JOIN`. Both plans return the same count; only the time differs.

Since v26.12.1 a one-hop pattern asked once per row - `COUNT { (p)-[:K]→() }` (with or without `MATCH`), `EXISTS { (p)-[:K]→() }`, the pattern predicate `WHERE (p)-[:K]→()` and `WHERE NOT (p)-[:K]→()`, and `size([(p)-[:K]→(q) | q])` - is answered from the edge list of the vertex the row binds, instead of planning and running a subquery for every row (issue #11891). `COUNT { }` reads the degree and `EXISTS { }` stops at the first edge, which runs two to three times faster than before; the predicate and `size()` also check that each relationship record is there, so they gain less. It applies to one fixed-length hop with no relationship variable, path variable or property map, where one end (either) is a vertex bound by the row with no label, any type list is allowed, and the other end is anonymous or a variable the row does not bind. A label on the far end still filters. The body of a `COUNT { }` or `EXISTS { }` has to be that `MATCH` alone, with no `WHERE`, `RETURN`, `ORDER BY`, `SKIP` or `LIMIT`; a pattern comprehension has to have no `WHERE` and project a variable or a literal. Anything else, a variable-length hop (`[:K*1..1]` included) too, keeps the general path. The answer is the one the general path gives, read permissions included: edges created or deleted earlier in the transaction are counted, a self loop of an undirected hop is one relationship, and a Graph Analytical View is not used, the edge list is read.

Multi-hop chains on the view. A chain of labeled hops such as `MATCH (a:P)-[:K]→(b:P)-[:K]→(c:P)` is walked on the view (`EXPLAIN` shows `GAVFusedChain`). Since v26.10.2 a `count(*)`, or `count(x)` of a variable of the chain, is counted during the walk with up to two grouping keys that are chain variables, in a `RETURN` or a `WITH` (`RETURN a, count(*) AS n`); `EXPLAIN` shows it on the chain as `fused count(*) AS n`. When the statement returns the paths' rows instead, they are produced in bounded batches, so memory no longer grows with the number of paths and a `LIMIT` stops the walk early (issue #9738). If the view is rebuilt while such a result is still being read, the query fails with a retryable error rather than mixing rows from two builds: run it again.

`WHERE` **on a chain walked on the view.** Since v26.10.2 each `AND`-ed condition of the `WHERE` is checked on the hop that binds the last variable it reads, so a path that fails it is not walked any further (issue #9938). In `MATCH (a:P)-[:K]→(b:P)-[:K]→(c:P) WHERE a.id < b.id AND b.id < c.id`, the first condition is checked as soon as `b` is reached and the second one once `c` is: `EXPLAIN` shows them on the chain as `filter on (b)=a.id < b.id` and `filter on (c)=b.id < c.id`. A pattern the view walks one hop at a time instead of as one chain - a cycle such as a triangle, closed by a `GAVExpandInto` hop, or a pattern that branches from one of its nodes - gets one `Filter` per condition right above the hop that binds its variables, as a pattern without a view does. A triangle counted with ordered ids (`WHERE a.id < b.id AND b.id < c.id`) so prunes its wedges before closing them, about three times faster on the LDBC SNB SF1 `KNOWS` graph. A condition that reads no node of the pattern is still checked on the complete path.

A transaction that holds uncommitted changes - any change, even to types the view does not cover - does not use the view: the view serves the committed graph only, and a query inside the transaction must see the transaction's own writes. Such a query, Cypher, Gremlin or a graph algorithm, runs on the records until the transaction commits, which `PROFILE` shows as a plan without the view. Run analytical queries outside write transactions, or after committing, to keep the acceleration.

12.4.14. Lifecycle

```
// Check status
if (gav.isReady()) { /* safe to query */ }

// Status values: NOT_BUILT, BUILDING, READY, STALE, FAILED (since v26.11.1: the build
threw, see getBuildError())
Status status = gav.getStatus();

// Drop (removes from registry + schema)
gav.drop();

// Shutdown (release resources, schema definition persists)
gav.shutdown();
```

12.4.15. Benchmark Results

On a graph with 500K vertices and ~8M edges:

Benchmark	OLTP	OLAP	Speedup
1-hop count	6.9 µs	1.2 µs	5.7x
2-hop	101.4 µs	5.1 µs	19.8x
3-hop	1,037 µs	56.4 µs	18.4x
5-hop	194,046 µs	5,141 µs	37.7x
Shortest Path	394 ms/pair	7.5 ms/pair	52.8x
PageRank (20 iter)	124,563 ms	316 ms	394.2x
Connected Components	5,591 ms	197 ms	28.4x

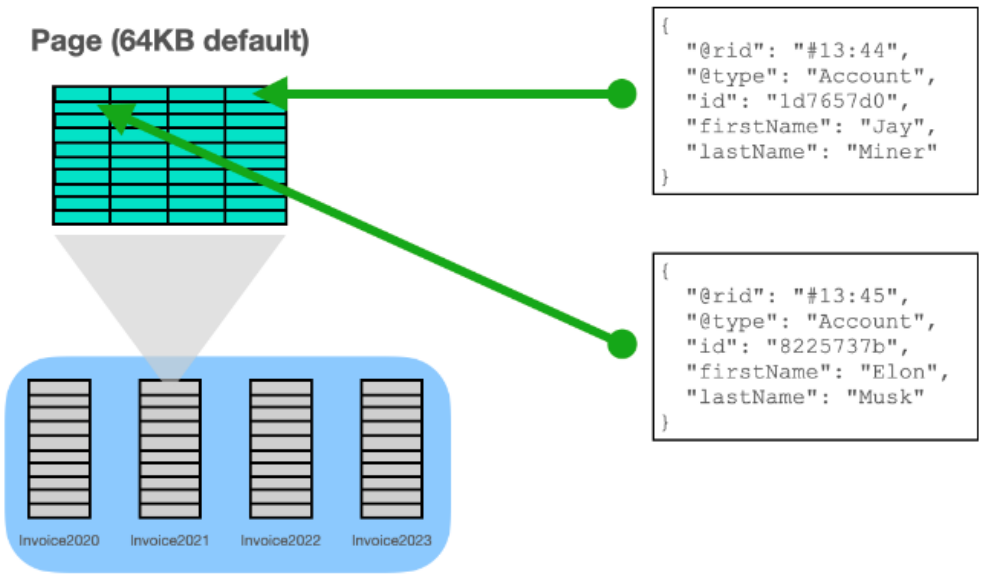
Benchmark	OLTP	OLAP	Speedup
Label Propagation	62,619 ms	645 ms	97.1x

12.4.16. Limitations

- CSR uses `int[]` arrays — maximum ~2.1 billion vertices per bucket and ~2.1 billion edges per direction
- Edges in CSR do not carry their own RID; the Cypher query planner falls back to OLTP only when the query returns an edge variable as a first-class record (e.g., `RETURN r`). Edge **properties** are fully supported via `withEdgeProperties()`
- Dictionary encoding applies only to string properties
- Initial build requires a full scan of selected vertex/edge types. A rebuild triggered by an invalidated persisted CSR certificate (or by `OFF` mode, or by exceeding the `SYNCHRONOUS`/`ASYNCHRONOUS` compaction threshold) also requires a full scan - only a clean-close-to-unchanged-reopen cycle skips it

12.5. Storage Internals

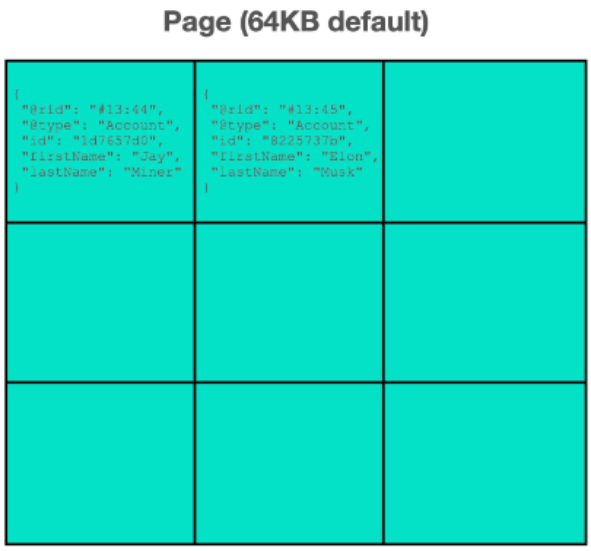
12.5.1. Pages and Record IDs



Records (Vertices, Edges and Documents) are stored in one or more pages. At creation time, the **unique** Record ID (**RID**) is assigned as the physical position where the beginning of record is stored.

RIDs cannot be changed and when a record is deleted are never recycled.

12.5.2. Zero Copy Objects



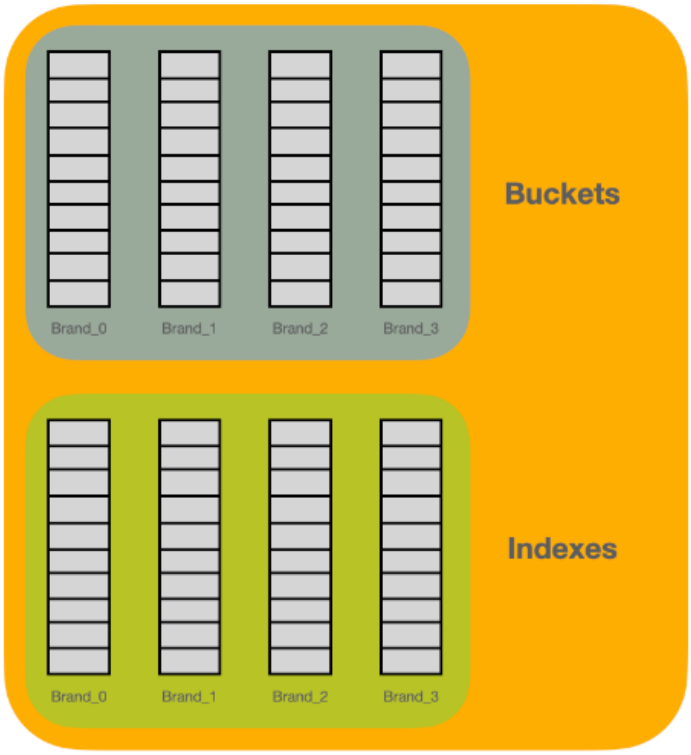
Pages are loaded in RAM as `ByteBuffer` objects (backed by byte arrays - `byte[]`). When a record is read, an in **immutable** vertex object is created which points to the portion of the `ByteBuffer` relative at the record.

All the immutable vertices point to the byte buffers with **zero copy** and no Java `Map` involved.

This greatly reduce the pressure on the Garbage Collection because most of the objects are big chunks of `byte[]`.

12.5.3. How Buckets Work

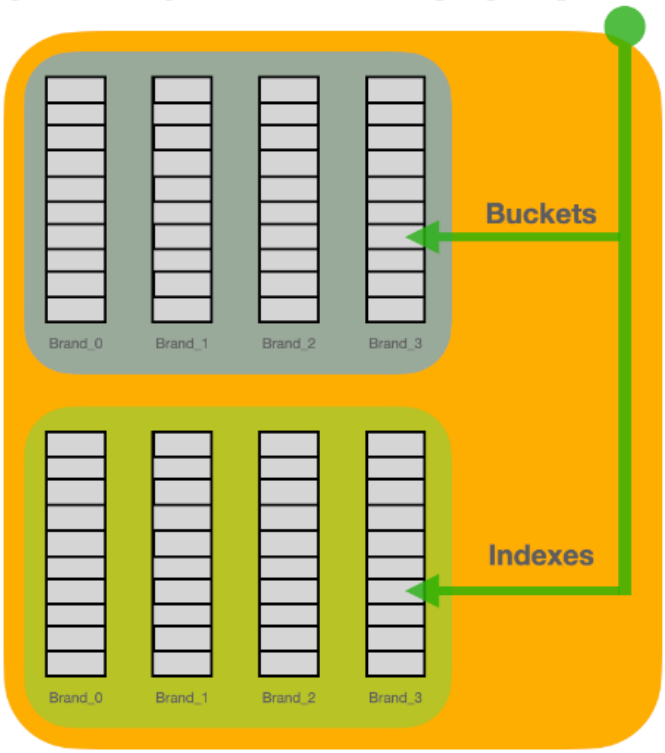
```
sql> create vertex type Brand
```



Buckets are physical files made of pages of 64KB (configurable). It's where the records are stored. Multiple buckets (files) per type allow to execute many jobs in parallel using all the CPU cores available.

Indexes, one per bucket, uses the **Log-structured merge-tree** (**LSM-Tree**) algorithm for efficient insertion speed and background concurrent compaction.

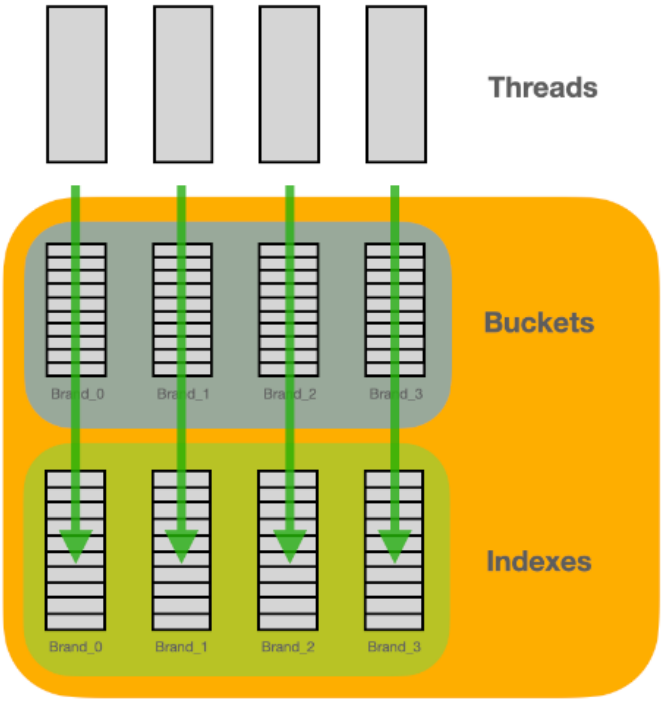
```
gremlin> g.addV('Brand').property('name', 'Maserati')
```



Round-Robin Strategy: **selected bucket = 3**

At insertion time the bucket is selected by using a (pluggable) strategy. By default is **Round-Robin**, but **Thread** and **Partitioned** strategies are available.


```
gremlin> g.addV('Brand').property('name', 'Maserati')
```

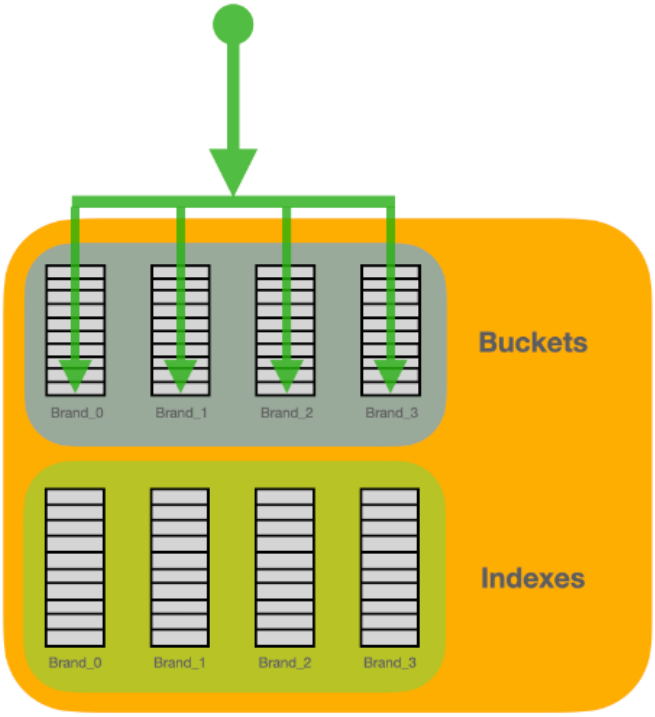



Thread strategy is preferable with multi-thread insertion where each thread insert on the same bucket and index files with zero contention with other threads.

```
bucket = threadId % available buckets
```

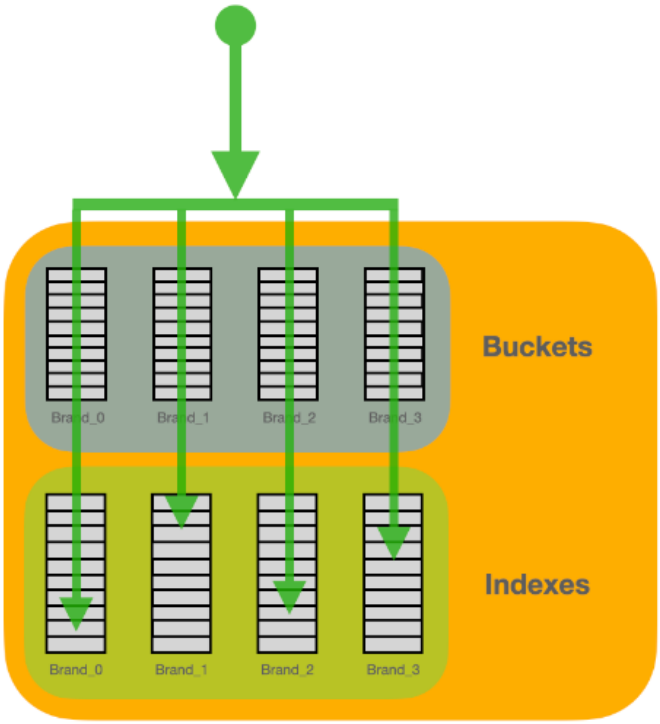
On a 2019 MacBook Pro ArcadeDB can insert **3.5 Million vertices/s** without index and **1.6 Million vertices/s** with one index

```
gremlin> g.V().hasLabel('Brand')
```



When no indexes are involved, the entire type is scanned. ArcadeDB browses all the buckets associated with the type.

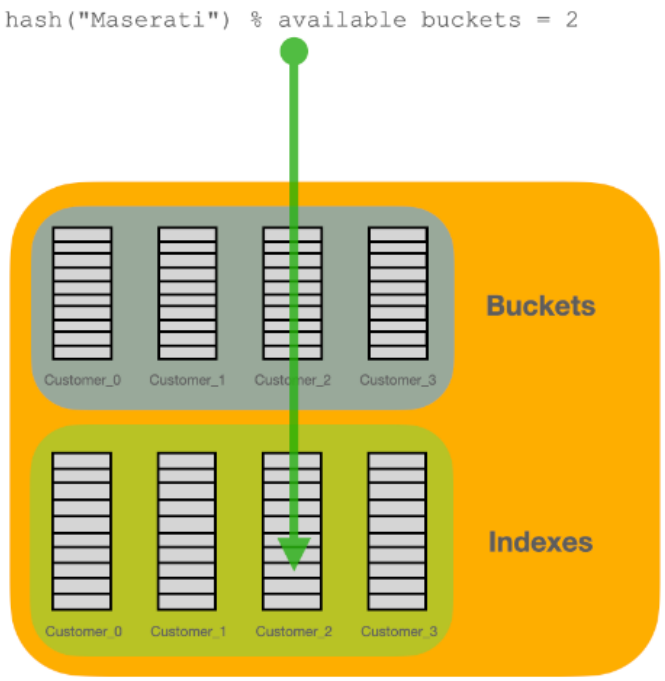
```
gremlin> g.V().hasLabel('Brand').has('name', 'Maserati')
```



When an index is used, a lookup on all the index per bucket is executed because the vertex could be in any of the configured buckets.

12.5.4. Bucket Partitioning

```
gremlin> g.V().hasLabel('Brand').has('name', 'Maserati')
```



It's possible to partition data based on one or more properties. This has the great advantage to involve only one index instead of all the indexes when a lookup by key is executed. It must be configured before the insertion:

1. Define a unique index on the property used for partitioning. In this example the brand's name:

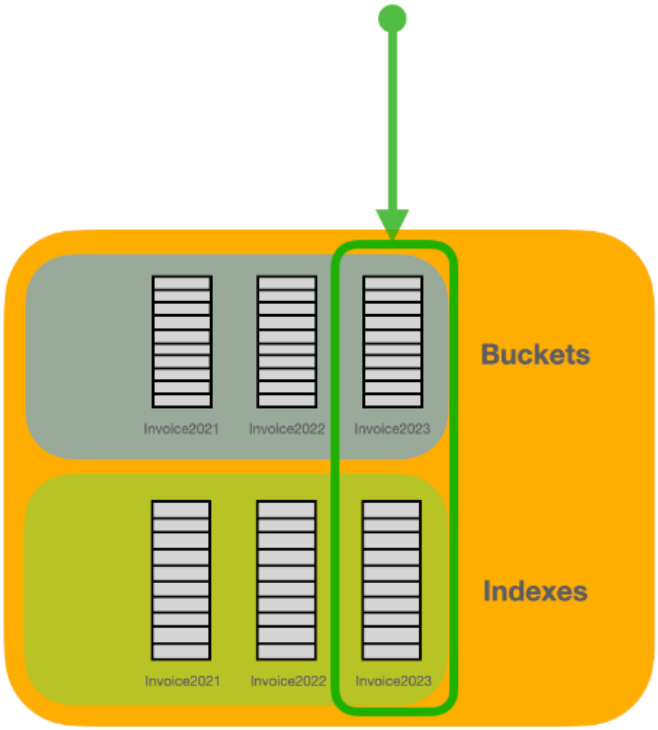
```
create index on Brand (name) unique
```

2. Use the bucket selection strategy "partitioned(name)":

```
alter type Brand BucketSelectionStrategy
`partitioned(name)`
```

12.5.5. Add and Remove Buckets from Types

```
sql> ALTER TYPE Invoice BUCKET +Invoice2023
```

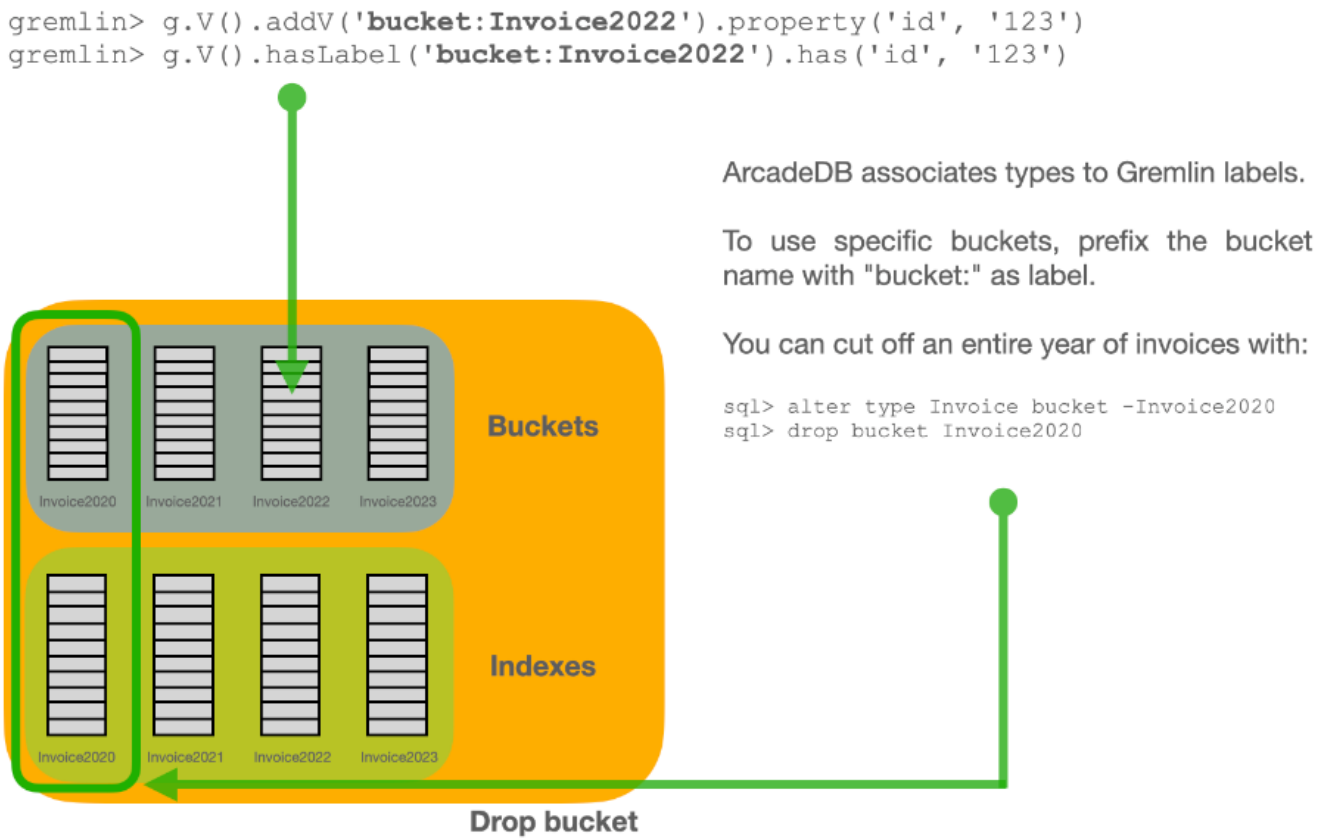


When a new bucket is added to a type, ArcadeDB creates automatically the underlying index.

Having one index per bucket is useful when a bucket is deleted and the entire sub-index is deleted with it.

Also, you could have the most recent data on a bucket stored on a fast SSD and move the old ones on a slower (but cheaper) HDD.

In the future could be possible to move buckets (and associated sub-indexes) to different servers for a distributed partitioning.



12.5.6. Page Version

Records are stored in pages. Each page has its own version number, which increments on each update. At creation the page version is zero. In optimistic transactions, ArcadeDB checks the version in order to avoid conflicts at commit time.

12.5.7. Schema File Persistence

The schema of a database lives in two JSON files in the database directory: `schema.json`, the current generation, and `schema.prev.json`, the generation it replaced. Every DDL statement rewrites them, once (see Cost of a Schema Change, and Batching Many).

Both files are published by an **atomic replacement**: the new content is written to a temporary sibling file, flushed to disk, and then renamed over its target. A process reading `schema.json` from the filesystem - an external backup, a monitoring script, a snapshot tool - therefore always observes one complete generation. It is never truncated, and it never disappears.

A full backup archives `schema.prev.json` as well, so a restored database keeps the same fallback copy as the database it came from. A database whose schema has never been re-saved has no previous generation yet, and the archive simply does not contain one.

The previous generation is saved as a byte copy of `schema.json`, never as a hard link: some file systems (a SeaweedFS FUSE mount, for one) accept a hard link and then lose its data once the original name is replaced, which would leave the fallback copy unreadable.

`schema.prev.json` is read only when `schema.json` is missing, empty or unreadable, and the atomic replacement above never leaves `schema.json` in any of those states. Since v26.12.1 the copy is

therefore not flushed to disk by every schema write: it is flushed once when the database is closed, and when a database is opened after a crash (it was not closed properly), a `schema.prev.json` whose content did not reach the disk is rebuilt from `schema.json`, and one that did is flushed. Until the database has seen one atomic replacement of `schema.json` since it was opened, and always on a file system that cannot rename atomically, the copy is still flushed by every write.

On a file system that cannot rename atomically at all, ArcadeDB falls back to a plain replacement and logs a warning once per JVM. The schema still saves; only the crash-time guarantee is lost.

12.5.8. Cost of a Schema Change, and Batching Many

Each schema write is durable: the new `schema.json` is flushed to disk and the directory that holds it is flushed once, after both files are published. On a real disk that is a few milliseconds to a few tens of milliseconds per write, depending on the device, against a fraction of a millisecond on `tmpfs`. Since v26.12.1 that is one file flush per write instead of two: the copy saved as `schema.prev.json` is flushed when the database closes rather than by every write (see Schema File Persistence). Every DDL statement writes the schema exactly once, however many bucket and index files and internal steps it involves. Before, a `CREATE TYPE` wrote it five or six times, a `CREATE PROPERTY` or a `CREATE INDEX` twice, and each write flushed the directory twice.

An application or a test suite that creates hundreds of types pays that write once per statement. To pay it once for a whole batch:

- **SQL**: send the statements as one `sqlscript`. A script made only of schema definition statements runs as one schema session and writes the schema once (setting `schemaBulkDDLScript`, on by default). Under HA the batch is also replicated as one Raft entry.
- **Java**: wrap the calls in `db.getSchema().bulkChange(() -> { ⋯ })`, which does the same for any sequence of schema API calls.
- **Any API**: run the DDL inside a transaction. The schema write is postponed to the end of the transaction.

Creating the bucket and index files themselves is cheap: about a millisecond per type with one index, once the schema writes are batched.

Since 26.11.1 a schema change no longer walks every type to work out which type owns which bucket, and neither does opening the database: creating 20,000 types in one `bulkChange()` went from minutes to under two seconds, and opening a database with 20,000 types from tens of seconds to about one second. The bucket-to-type maps follow the additive changes in place (a bucket attached to a type, an external or edge-list bucket paired with it) instead of being rebuilt over every type, and `schema.json` is streamed type by type without building a tree of the whole schema first (issue #10247). A detach, a drop, a rename or a stripe pool still rebuilds the maps. What remains proportional to the size of the schema is the write itself, since `schema.json` holds every type, so batching matters more the more types the database has.

Schema changes are not transactional. A DDL statement run inside a transaction takes effect immediately and is visible to every other session; a rollback does not undo it. Its types, properties and files stay, and the schema is written at the end of the transaction whether it commits or rolls back. A crash before that write leaves

> the new bucket and index files on disk without a schema entry naming them: the database opens normally without the new types, and running the same DDL again succeeds. The window lasts as long as the transaction, so it is worth keeping a transaction that creates types or indexes short, and creating them before a long import rather than inside it. Records another session committed into such a type before the crash are in those files too, and are not reachable after the restart: recreate the type and they stay unreferenced. Because a schema change is not transactional, the next commit on any session writes it, including a session that only writes records, and the type it names may belong to a transaction that has not ended yet. The transaction that ran the DDL is not exposed to this for its own records: the schema is written just before its commit becomes durable, so `begin(); CREATE TYPE; INSERT; commit()` never leaves acknowledged records in files the schema does not name. Under HA, each DDL statement inside a transaction is still replicated on its own, so prefer a script or `bulkChange()` there.

12.6. LSM-Tree Algorithm

ArcadeDB's default index algorithm is the LSM Tree. For equality-only lookups, ArcadeDB also provides a Hash Index based on extendible hashing.

12.6.1. Quick Overview

The LSM-Tree index is optimized for writes with a complexity of O(1) in writing because it does not require a re-balancing of the tree (like the B+Tree) and O(log(N)) to O(log(N)^2) with reads, depending on how fragmented (non compacted) the index is at a point of time.

The class LSMTreeIndex is the main entrypoint for the index. This class contains an instance of a LSMTreeIndexMutable class that manages the current mutable index and a subIndex in case there is a compacted index connected.

ArcadeDB uses pages to store index entries. Those pages are immutable once full and cannot be changed. This is why it is considered an append-only index. After a while, ArcadeDB runs the compactor that compacts the index by reading the content of the pages from disk, compresses the content and write them back into new pages to disk. This process happens in background while the index can be used by the application.

The LSMTreeIndexMutable class holds the current page in RAM so new entries can be quickly appended to the page. When the database is closed or when the page is full, the page is serialized to disk and becomes immutable. Since the pages are immutable, deletion are managed with special placeholders to mark the entries as removed.

Since the LSM-Tree is append only, the most updated entries are at the end of the index. If an entry is created and then deleted, the deletion will always be after the creation. For this reason cursors start from the last pages back to the first one available. While the cursor is browsing the pages back, it keeps track of the deleted values encountered.

Compaction

During compaction, new compressed version of the pages are stored. The pages are stored in segments where the root page is the first page of the segment. A file can have multiple segments. The compaction requires a lot of RAM to properly compact large indexes and make them efficient. To control the amount of RAM the compaction task is using, the setting `arcadedb.indexCompactionRAM` is used to determine the maximum amount of RAM allowed to use for compaction. By default is 300MB of RAM. Also, the setting `arcadedb.indexCompactionMinPagesSchedule` tells the index when it is time for scheduling a compaction. By default the minimum is 10 pages not compacted. If you set `0`, the automatic compaction is disabled.

If, for example, the compaction task finds 20 pages to compact, but the RAM requirements allow only 10 pages to be compacted, then the task will process the first 10 pages in one segment and the remaining 10 pages in the following segment. Having multiple segments in the same file allows to run the compaction with a minimum amount of RAM if needed.

Page layout

There are 2 types of pages: root page and data page. The pages are populated from the head of the page by storing the pointers to the pair of key/value that are written from the tail. A page is full when there is no space between the head (key pointers) and the tail (key/value pairs). When a page is full, another page is created, waiting for a compaction.

Root Page

It's the first page of a segment of pages. The header contains more information than the Data Page (see below). Both root pages and data pages store keys. With mutable indexes the only difference is that the root page has a larger header. For compacted indexes, the root page does not store entries, but contains pointers to the data pages with the first key indexed on the page of the current segment.

Header:

```
[offsetFreeKeyValueContent(int:4),numberOfEntries(int:4),mutable(boolean:1),compactedP
ageNumberOfSeries(int:4),subIndexFileId(int:4),numberOfKeys(byte:1),keyType(byte:1)*]
```

- **offsetFreeKeyValueContent** is the offset in the page of free space to store key/value pairs. When a page is new, the offset points to the end of the page because the pairs are filled from the end to the beginning
- **numberOfEntries** number of entries (pairs) in the current page
- **mutable** 1 means mutable, 0 immutable. Immutable pages cannot be modified, only compacted and then removed at the end of compaction cycle
- **compactedPageNumberOfSeries** number of pages in the compacted segment. This is stored in the last page of the segment and it is used to count the pages of the current segment and therefore finding the root page of the segment
- **subIndexFileId** file id of compacted page segment
- **numberOfKeys** number of keys. If you're using a composite index, multiple keys are used,

otherwise only 1

- **keyType** an array of key types. If the numberOfKeys is 3, then 3 bytes for keyType are written

Data Page

Header:

```
[offsetFreeKeyValueContent(int:4),numberOfEntries(int:4),mutable(boolean:1),compactedP
ageNumberOfSeries(int:4)]
```

Look the root page for the meaning of the header fields.

Mutable Structure

Before being compacted, the LSM-Tree index is conceptually simple: new entries are always inserted in the last page of the file that is the only mutable page of the index. If the page is full, a new mutable page is created and the old mutable page become immutable and stored on disk. The entries in the page are written already ordered by key. This means a cursor can just browse the page from the first entry to the last to have ordered items. The search of a key is executed with the dichotomy algorithm. Pages can have entries with the same key, because have been added later in the index. On writing, the order of the keys is maintained inside the same page. This means to have a true ordered iterator, a virtual iterator is created with pointers to all the pages. Every time you move forward in the iterator, the next available key in the page pointer is compared among each other and the minor is returned. This can become quite expensive in the case of millions of entries and hundreds of pages. For this reason a periodic compaction is needed to keep values ordered together in the same page and also to remove deleted items.

Chapter 13. Help

13.1. Community

Join our growing community around the world, for ideas, discussions and help regarding ArcadeDB.

- Chat live with us on Discord: https://discord.gg/w2Npx2B7hZ
- Follow us on Twitter: https://twitter.com/arcade_db
- or on Bluesky: https://bsky.app/profile/arcadedb.bsky.social
- Connect with us on LinkedIn: https://www.linkedin.com/products/arcadedb
- or on Facebook: https://www.facebook.com/arcadedb
- Questions tagged `#arcadedb` on Stack Overflow: https://stackoverflow.com/questions/tagged/arcadedb
- View our official Blog: https://blog.arcadedb.com/

13.1.1. Professional Support

Need guaranteed response times and dedicated assistance for your production environment? ArcadeDB offers professional support plans to help you get the most out of your deployment.

- View available plans: Pricing
- Service level details: SLA

13.1.2. Frequently Asked Questions

- What is ArcadeDB's release cycle?
  - There are typically monthly releases with occasional bi-monthly releases.
- How do I build or contribute to ArcadeDB?
  - See the contributing guide in the ArcadeDB github repository: CONTRIBUTING.md
- What is a good introductory book about NoSQL or Graph databases?
  - Try **SQL and NoSQL Databases**. It teaches basics on relational (SQL), document (Mongo), and graph (Cypher) databases.
- How do I protect prepared statements against SQL injection?
  - A good starting point is the OWASP SQL Injection Prevention Cheat Sheet. To use escaping as defense, the characters `\`, `"`, `'`, need to be escaped with `\\`, `\"`, `\'` respectively. Furthermore, the strings `;` and `--` need to be replaced, for example with similar unicode characters like `;` (greek question mark, `\u037E`) and – (en dash, `\u2013`). Please note, that this escaping and replacing is merely a minimal protection, and generally not sufficient.
- How are duplicate keys handled in map properties or inserted content?
  - As in the ECMAScript Specification (Note 2), duplicate keys in JSON objects are resolved by

keeping the last one read and discarding previous ones.

- How can strictly ascending unique identifiers be data modeled with ArcadeDB?
  - One can use a unique index together with transactions of `next_id = max(id) + 1`. See https://www.cybertec-postgresql.com/en/postgresql-sequences-vs-invoice-numbers/ for more details.
- How can "prepared statements" be formed for ArcadeDB?
  - Prepared statements can be directly used via the Postgres driver and the JDBC driver. Furthermore, the Java API and the HTTP/JSON API provide parameterized queries and commands which can manually be extended to prepared statements by wrapping the respective method or request calls in a function and forwarding the function arguments to the parameterized query or command.
- How can database schema be maintained?
  - One can use `.sql` files listing the SQL commands creating the schema. To keep this script idempotent the DDL commands have to have the suffix `IF NOT EXISTS`. Such a SQL script can then be parsed using the `LOAD` command of the console, which can also run commands passed as command-line arguments.

13.2. Get the Latest Build

ArcadeDB ships a new official release roughly once a month, but every commit merged into the `main` branch is also packaged as a development build (a **SNAPSHOT**). If your fix has already landed on `main` and you don't want to wait for the next monthly release, you have two options:

1. **Build it yourself locally** — fastest if you already have a JDK and Maven, and produces the same artifacts the CI does.
2. **Pull the SNAPSHOT from Maven Central** — no toolchain required; just wait ~15 minutes after the commit lands.

Both options give you the **exact same code** — the difference is whether the bytes are produced on your machine or on the GitHub Actions runner.

13.2.1. Option 1: Build it Yourself

You'll need:

- **JDK 21 or newer** — check with `java -version`.
- **Maven 3.9+** — check with `mvn -version`.
- (optional) **Docker Desktop** — only required if you want a local Docker image; skip the `-Pdocker` profile otherwise.

Steps:

```
# 1. Clone the repository
git clone https://github.com/ArcadeData/arcadedb.git
cd arcadedb
```

```
# 2. Build the project (skip tests for a faster build)
mvn clean install -DskipTests -Pdocker
```

The `-Pdocker` profile additionally builds and tags Docker images on your local Docker daemon. After the build completes you'll find them with:

```
docker images | grep arcadedata/arcadedb
```

You will see images such as `arcadedata/arcadedb:latest`, `arcadedata/arcadedb-minimal:latest`, and `arcadedata/arcadedb-headless:latest`, each also tagged with the snapshot version (for example `26.12.1-SNAPSHOT`). If you don't need Docker, just drop `-Pdocker` from the command.

The packaged distributions (`tar.gz` and `zip`) are produced in:

```
package/target/
```

Look for files named `arcadedb-<version>-SNAPSHOT-base.tar.gz`, `arcadedb-<version>-SNAPSHOT-base.zip`, and the `headless` variants. Unpack one of them and follow the same launch instructions as a stable release — see Installation from Binaries.

A clean build from scratch typically takes 2–4 minutes on a modern laptop. Re-runs are noticeably faster thanks to Maven's incremental compilation and dependency cache.

13.2.2. Option 2: Wait ~15 Minutes for the Maven Central SNAPSHOT

Every push to `main` triggers a GitHub Actions deploy workflow. As long as the build passes, it publishes the artifacts to Maven Central's snapshots repository — typically within ~15 minutes from the commit.

To consume the SNAPSHOT in your own project, point your build at the Maven Central snapshots repository.

Maven (`pom.xml`):

```
<repositories>
  <repository>
    <id>central-snapshots</id>
    <url>https://central.sonatype.com/repository/maven-snapshots/</url>
    <releases><enabled>false</enabled></releases>
    <snapshots><enabled>true</enabled></snapshots>
  </repository>
</repositories>

<dependencies>
```

```
  <dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-engine</artifactId>
    <version>26.11.1-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Gradle (`build.gradle`):

```
repositories {
  maven { url 'https://central.sonatype.com/repository/maven-snapshots/' }
}

dependencies {
  implementation 'com.arcadedb:arcadedb-engine:26.11.1-SNAPSHOT'
}
```

Replace `26.11.1-SNAPSHOT` with the version reported in the `pom.xml` of the `main` branch at the time of your build. The version is bumped after each official release, so the value changes over time.

To force Maven to re-resolve the latest snapshot (instead of using the locally cached one), pass the `-U` flag: `mvn -U clean install`.

13.2.3. Verifying You Have the Build You Expect

A SNAPSHOT distribution includes a `build.number` file at its root. Open it to confirm the exact commit hash and build timestamp:

```
cat arcadedb-<version>-SNAPSHOT/build.number
```

If you open an issue against a SNAPSHOT build, please include the contents of this file — see Report an Issue.

13.2.4. When to Use Which Option

If you…	Use
Need to test a fix immediately and have JDK/Maven installed	Build it yourself
Want a Docker image of the snapshot on your local machine	Build it yourself with `-Pdocker`
Don’t want to install a toolchain	Wait for the Maven Central snapshot

If you…	Use
Are integrating ArcadeDB as a library dependency in your own project	Maven Central snapshot (with the repository declared in your build)
Need to reproduce CI behavior exactly	Maven Central snapshot

13.3. Report an Issue

Very often when a new issue is open it lacks some fundamental information. This slows down the entire process because the first question from the ArcadeDB team is always "What release of ArcadeDB are you using?" and every time a Ferret dies in the world.

So please add more information about your issue:

1. ArcadeDB **release**? (If you're using a SNAPSHOT please attach also the build number found in "build.number" file)
2. What **steps** will reproduce the problem?
3. **Settings**. If you're using custom settings please provide them below (to dump all the settings run the application using the JVM argument `-Darcadedb.environment.dumpCfgAtStartup=true`)
4. What is the **expected behavior** or output? What do you get or see instead?
5. If you're describing a performance or memory problem the **profiler** dump can be very useful

Now you're ready to create a new issue on GitHub.

13.4. Requirements

13.4.1. Memory

ArcadeDB's memory (RAM) requirements depend on the configured profile:

- `default` profile: 800MB
- `low-ram` profile: 16MB

see also RAM Configuration.

13.4.2. Java

ArcadeDB runs on the Java Runtime Environment (JRE) in versions **21** (LTS) and **25** (LTS); tested JREs are:

- OpenJDK 17 is **NOT** compatible,
- OpenJDK 21 is compatible and the suggested version to use.
- Corretto 21 is also compatible.

The **headless** variants of JRE packages are compatible to ArcadeDB.

For now, a separate Java 17 branch and packages are maintained, see the README.

Java 17

Java 17 is supported through a special `java17` branch.

Query Languages

Chapter 14. SQL

14.1. SQL Introduction

ArcadeDB speaks SQL as its primary query language, extended for graph traversal, full-text search, time series, and vector similarity. Nine out of ten developers can write their first ArcadeDB query without learning anything new — the differences are covered in the ArcadeDB SQL dialect section below.

14.1.1. Introduction to SQL

When it comes to query languages, SQL is the most widely recognized standard. The majority of developers have experience and are comfortable with SQL. For this reason ArcadeDB uses SQL as its query language and adds some extensions to enable graph functionality. There are a few differences between the standard SQL syntax and that supported by ArcadeDB, but for the most part, it should feel very natural. The differences are covered in the ArcadeDB SQL dialect section of this page.

If you are looking for the most efficient way to traverse a graph, we suggest using MATCH instead.

Many SQL commands share the WHERE condition. Keywords are case insensitive, but type names, property names and values are case sensitive. In the following examples keywords are in uppercase but this is not strictly required.

For example, if you have a type `MyType` with a field named `id`, then the following SQL statements are equivalent:

```
SELECT * FROM MyType WHERE id = 1
select * from MyType where id = 1
```

The following is NOT equivalent. Notice that the field name 'ID' is not the same as 'id'.

```
SELECT * FROM MyType WHERE ID = 1
```

Also the following query is NOT equivalent because of the type 'mytype ' is not the same as 'MyType'.

```
SELECT * FROM mytype WHERE id = 1
```

14.1.2. Automatic usage of indexes

ArcadeDB allows you to execute queries against any field, indexed or not-indexed. The SQL engine automatically recognizes if any indexes can be used to speed up execution. You can also query any indexes directly by using `INDEX:<index-name>` as a target. Example:

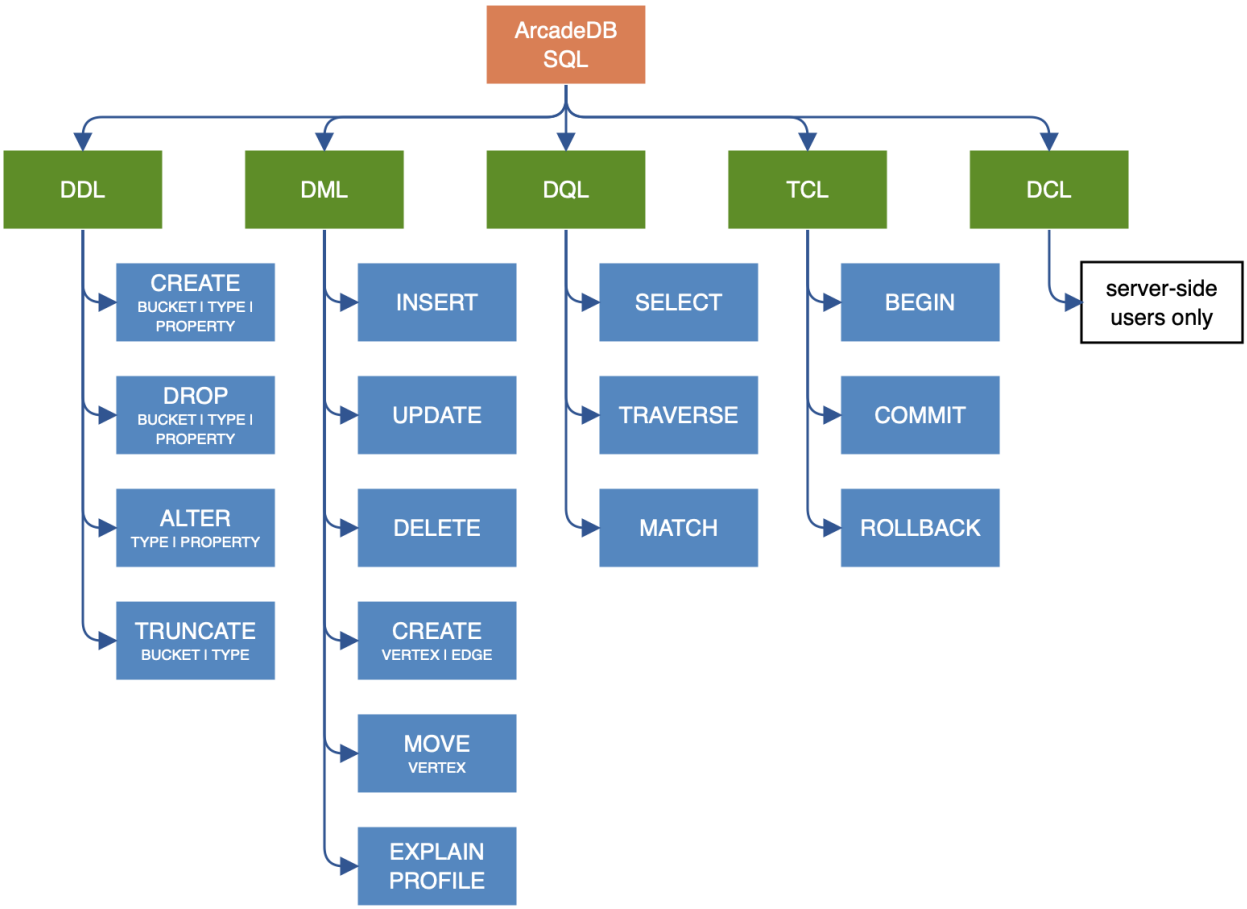
```
SELECT * FROM INDEX:myIndex WHERE key = 'Jay'
```

14.1.3. Extra resources

- Syntax
- Projections
- Where clause
- Operators
- Pagination
- SQL Script
- Match for traversing graphs

14.1.4. ArcadeDB SQL dialect

ArcadeDB supports SQL as a query language with some differences compared with SQL. The ArcadeDB team decided to avoid creating Yet-Another-Query-Language. Instead we started from familiar SQL with extensions to work with graphs. We prefer to focus on standards.



14.1.5. Learning SQL

If you want to learn SQL, there are many online courses such as:

- KhanAcademy
- Introduction to SQL at W3 Schools
- Beginner guide to SQL
- SQLCourse.com
- YouTube channel Basic SQL Training by Joey Blue

alternatively, order a book such as:

- SQL in 10 Minutes a Day
- or any of these.

For details on ArcadeDB's dialect, see ArcadeDB SQL Syntax. You can also download a SQL Command Cheat Sheet (pdf).

The simplest SQL query just returns a constant, and is given by:

```
SELECT 1
```

and could have practical use, for example, as a connection test. Furthermore, such constant queries can be used as a calculator:

```
SELECT sqrt(6.0 * 7.0)
```

A typical ArcadeDB SQL query has the following components:

```
SELECT projections
FROM type
WHERE predicate
GROUP BY property
ORDER BY projection
SKIP number
LIMIT number
```

for further components or details, see the SELECT command. As for classic SQL the execution order in ArcadeDB SQL is:

1. FROM
2. WHERE
3. GROUP BY
4. SELECT

5. `ORDER BY`
6. `SKIP`
7. `LIMIT`

14.1.6. No JOINs

The most important difference between ArcadeDB and a Relational Database is that relationships are represented by `LINKS` instead of JOINs.

For this reason, the typical JOIN syntax of relational databases is not supported. ArcadeDB uses the "dot (`.`) notation" to navigate `LINKS`. Example 1 : In SQL you might create a join such as:

```
SELECT *
FROM Employee A, City B
WHERE A.city = B.id
  AND B.name = 'Rome'
```

In ArcadeDB, an equivalent operation would be:

```
SELECT *
FROM Employee
WHERE city.name = 'Rome'
```

Another example:

```
SELECT *
FROM Employee A, City B, Country C
WHERE A.city = B.id
  AND B.country = C.id
  AND C.name = 'Italy'
```

In ArcadeDB, an equivalent operation would be:

```
SELECT *
FROM Employee
WHERE city.country.name = 'Italy'
```

Furthermore, RIDs can be resolved by nested projections.

14.1.7. Projection

In SQL, projections are mandatory and you can use the star character `*` to include all of the fields. With ArcadeDB this type of projection is optional. Example: In SQL to select all of the columns of Customer you would write:

```
SELECT * FROM Customer
```

In ArcadeDB, the * may be omitted:

```
SELECT FROM Customer
```

14.1.8. System Types

ArcadeDB exposes several read-only virtual types under the `schema:` prefix that let you introspect the database from SQL. They behave like any other type: projections, filters, `ORDER BY`, `LIMIT` and aggregations all work, and you can `expand()` collection-valued properties.

Virtual type	What it returns
`schema:types`	One row per defined document/vertex/edge/timeseries type. Properties: `name`, `type` (`document`/`vertex`/`edge`/`timeseries`), `records` (count), `buckets`, `bucketSelectionStrategy`, `parentTypes`, nested `properties` (with `name`, `type`, `mandatory`, `readOnly`, `notNull`, `min`, `max`, `default`, `regexp`, `compression`, `custom`, ...) and nested `indexes`. For external properties also `externalBuckets`. For time series types, additional time-series metadata.
`schema:buckets`	One row per bucket (the physical storage unit). Properties: `name`, `fileId`, `records` (count), `purpose` (e.g. `PRIMARY`; filter with `WHERE purpose = 'PRIMARY'` to hide internal buckets like external-property pairs).
`schema:indexes`	One row per index. Properties: `name`, `indexType`, `typeName`, `properties`, `keyTypes`, `unique`, `automatic`, `compacting`, `fileId`, `size`, `supportsOrderedIterations`, `associatedBucketId`, `nullStrategy`, `metadata`. `metadata` is a map with every setting of the `METADATA` clause the index type accepts and the value in force, defaults included (for example `addHierarchy`, `similarity` and `maxConnections` on an `LSM_VECTOR` index, `analyzer` and `defaultOperator` on a `FULL_TEXT` index). Enums are reported by name (`"similarity": "COSINE"`), the same value the `METADATA` clause takes. An index type with no settings, such as a plain `LSM_TREE` index, has no `metadata` key, and on a vector index `efSearch` appears only when it was set explicitly (otherwise searches use the adaptive beam). `SELECT FROM schema:index:<name>` returns the same `metadata` for a single index, for example `SELECT metadata FROM schema:index:`Doc[embedding]`` *(since v26.12.1, issue #13096).*
`schema:database`	A single row with database-wide metadata: `name`, `path`, `mode`, `size`, `dateFormat`, `dateTimeFormat`, `timezone`, `encoding`, plus a `settings` array of all `DATABASE`-scoped global settings (`key`, `value`, `default`, `description`, `overridden`). Use `SELECT expand(settings) FROM schema:database` to list current settings, or wrap it in a subquery to filter: `SELECT FROM ( SELECT expand(settings) FROM schema:database ) WHERE key = '<setting>'`. Password values are masked.

Virtual type	What it returns
`schema:materializedviews`	One row per materialized view. Properties: `name`, `query`, `backingType`, `refreshMode`, `refreshInterval`, `simpleQuery`, `lastRefreshTime`, `status`, `sourceTypes` (every type the defining query reads, at any depth; their subtypes are tracked too), plus runtime metrics: `refreshCount`, `refreshTotalTimeMs`, `refreshMinTimeMs`, `refreshMaxTimeMs`, `refreshAvgTimeMs`, `errorCount`, `lastRefreshDurationMs`.
`schema:graphanalyticalviews`	One row per graph analytical view. Properties: `name`, `vertexTypes`, `edgeTypes`, `propertyFilter`, `edgePropertyFilter`, `updateMode`, `compactionThreshold`, plus live stats when the view is built: `status`, `nodeCount`, `edgeCount`, `memoryUsageBytes`, `buildTimestamp`, `buildDurationMs`.
`schema:continuousaggregates`	One row per continuous aggregate (incremental rollup over a time-series source). Properties: `name`, `query`, `backingType`, `sourceType`, `bucketIntervalMs`, `bucketColumn`, `timestampColumn`, `watermarkTs`, `lastRefreshTime`, `status`, plus runtime metrics: `refreshCount`, `refreshTotalTimeMs`, `refreshMinTimeMs`, `refreshMaxTimeMs`, `refreshAvgTimeMs`, `errorCount`, `lastRefreshDurationMs`.
`schema:triggers`	One row per trigger. Properties: `name`, `typeName`, `timing` (`BEFORE`/`AFTER`), `event` (`CREATE`/`READ`/`UPDATE`/`DELETE`), `actionType` (`SQL`/`JAVASCRIPT`/`JAVA`) and `actionCode`. Triggers whose type the current user cannot read are omitted.
`schema:stats`	A single row with per-database runtime statistics (`databaseName` plus all entries returned by `Database.getStats()` — counters, cache hits/misses, page I/O and the like). Useful for ad-hoc diagnostics from the SQL console.
`schema:dictionary`	A single row describing the database property dictionary (the internal mapping of property names to numeric ids used to keep records compact): `totalEntries`, `totalPages`, and `entries` (a map of property name → dictionary id).

Examples:

```
SELECT FROM schema:types
```

```
SELECT name, records FROM schema:buckets WHERE purpose = 'PRIMARY' ORDER BY records
DESC
```

```
SELECT FROM ( SELECT expand(settings) FROM schema:database ) WHERE key =
'arcadedb.serverMetrics'
```

14.1.9. DISTINCT

You can use DISTINCT keyword exactly as in a relational database:

```
SELECT DISTINCT name FROM City
```

14.1.10. HAVING

ArcadeDB does not support the `HAVING` keyword, but with a nested query it's easy to obtain the same result. Example in SQL:

```
SELECT city, sum(salary) AS salary
FROM Employee
GROUP BY city
HAVING salary > 1000
```

This groups all of the salaries by city and extracts the result of aggregates with the total salary greater than 1000 dollars. In ArcadeDB the `HAVING` conditions go in a select statement in the predicate:

```
SELECT
FROM ( SELECT city, SUM(salary) AS salary FROM Employee GROUP BY city )
WHERE salary > 1000
```

14.1.11. Multiple targets

ArcadeDB allows only one type (types are equivalent to tables in this discussion) as opposed to SQL, which allows for many tables as the target. If you want to select from 2 types, you have to execute 2 sub queries and join them with the `UNIONALL` function:

```
SELECT FROM E, V
```

In ArcadeDB, you can accomplish this with a few variable definitions and by using the `expand` function to the union:

`DISTINCT` vs `distinct()`

Query results without duplicates can be realized either using the keyword `DISTINCT` or the function `distinct()`. The keyword `DISTINCT` is be used if the entire projection should be unique, while the `distinct` function returns unique elements only for its argument field

`UNWIND` vs `expand()`

Collections fields can be spread into a set of results by either using the keyword `UNWIND` or the function `expand()`. Both produce similar results but via different semantics and constraints.

14.1.12. SQL Differences

Following are some basic differences between ArcadeDB and PostgreSQL.

Postgres	ArcadeDB
`CREATE TABLE`	`CREATE TYPE`
`ALTER TABLE`	`ALTER TYPE`
`ADD COLUMN`	`CREATE PROPERTY`
`ALTER COLUMN`	`ALTER PROPERTY`
`SELECT * FROM information_schema.tables`	`SELECT FROM schema:types`
`SELECT * FROM pg_indexes`	`SELECT FROM schema:indexes`
`SELECT * FROM pg_database`	`SELECT FROM schema:database`

However, there is a SQL condition where the two systems do differ. When you use functions that do not affect the column name, SQL defines that function name as the column name. In ArcadeDB, since we're projecting from a record, the resulting name remains the property name.

For instance, in SQL:

```
SELECT count(*) FROM Employee
```

would result in a data set with one column named "count(*)".

The ArcadeDB equivalent is:

```
SELECT count(*) FROM Employee
```

resulting in a data set with one column name of "count". If you specify a name for a column in a projection using the AS keyword, both SQL and ArcadeDB would use the provided name.

If you prefer having the function name as the column name, like standard SQL, use AS, like:

```
SELECT count(*) as _count_ FROM Employee
```

14.1.13. Commands

While the ArcadeDB SQL syntax is very similar to SQL-92, commands are similar but different. For more commands, see the diagram above.

14.1.14. Comments

Use `-- ` (two dashes followed by a space) for single line comments:

```
-- This is a single line comment
```

Use `/* ⋯ */` for multi-line comments.

```
/* this is a
   multiline comment */
```

14.2. SQL - SELECT

ArcadeDB supports the SQL language to execute queries against the database engine. For more information, see operators and functions. For more information on the differences between this implementation and the SQL-92 standard, please refer to this section.

Syntax:

```
SELECT [ [DISTINCT] <Projections> ]
    [ FROM <Target> [ [AS] <Alias> ] [ LET <Assignment>* ] ]
    [ WHERE <Condition>* ]
    [ GROUP BY <Field>* ]
    [ ORDER BY <Fields>* [ ASC|DESC ] * ]
    [ UNWIND <Field>* ]
    [ SKIP <SkipRecords> ]
    [ LIMIT <MaxRecords> ]
    [ TIMEOUT <MilliSeconds> [ <STRATEGY> ] ]
```

- `Projections` Indicates the data you want to extract from the query as the result-set. Note: In ArcadeDB, this variable is optional. In the projections you can define aliases for single fields, using the `AS` keyword; in current release aliases cannot be used in the WHERE condition, GROUP BY and ORDER BY (they will be evaluated to null)
  - `DISTINCT` enforces unique projection results.
- `FROM` Designates the object to query. This can be a type, bucket, single RID, set of RID index values sorted by ascending or descending key order.
  - When querying a type, for `<target>` use the type name.
  - When querying a bucket, for `<target>` use `BUCKET:<bucket-name>` (eg. `BUCKET:person`) or `BUCKET:<bucket-id>` (eg. `BUCKET:12`). This causes the query to execute only on records in that bucket. The bucket name can also come from an input parameter: `BUCKET::name` (named) or `BUCKET:?` (positional), as in `SELECT FROM BUCKET::name` with `{"name": "person"}`. The same statement can be run with a different bucket each time *(since v26.12.1, issue #12480: earlier versions kept reading the bucket of the first execution).*
  - When querying record ID's, you can specific one or a small set of records to query. This is useful when you need to specify a starting point in navigating graphs.
  - To query the schema you can use one of the read-only `schema:*` virtual types (see System Types for the full list and the fields each one returns):
    - `select from schema:types` — defined types
    - `select from schema:buckets` — physical buckets
    - `select from schema:indexes` — defined indexes

        - `select from schema:database` — database metadata and settings
        - `select from schema:materializedviews` — materialized views
        - `select from schema:graphanalyticalviews` — graph analytical views
        - `select from schema:continuousaggregates` — continuous aggregates over time series
        - `select from schema:triggers` — defined triggers
        - `select from schema:stats` — per-database runtime statistics
        - `select from schema:dictionary` — property-name dictionary

        The `schema:*` targets support `WHERE`, `ORDER BY`, `SKIP` and `LIMIT` like any other target.

    - When querying indexes, use the following prefixes:
        - `INDEXVALUES:<index>` and `INDEXVALUESASC:<index>` sorts values into an ascending order of index keys.
        - `INDEXVALUESDESC:<index>` sorts the values into a descending order of index keys.
- `AS <Alias>` Optionally names the records the target returns, so the rest of the statement can refer to them by that name. The `AS` keyword is optional: `FROM Person p` and `FROM Person AS p` are the same. The alias can be used in the projections, `WHERE`, `LET`, `GROUP BY` and `ORDER BY`, and in `UPDATE` and `DELETE`, which take the same `FROM` clause. `p.name` is the record's `name` property and a bare `p` is the record itself, so `SELECT p FROM Person p` and `SELECT FROM Person` return the same thing. Qualifying a property with the alias does not cost you an index: `SELECT FROM Person p WHERE p.name = 'Luke'` uses an index on `Person.name` exactly as the unqualified form does. When a projection gives a column the same name as the target alias, `ORDER BY` and `GROUP BY` mean the column.
- `WHERE` Designates conditions to filter the result-set.
- `LET` Binds context variables to use in projections, conditions or sub-queries.
- `GROUP BY` Designates field on which to group the result-set.
- `ORDER BY` Designates the field with which to order the result-set. Use the optional `ASC` and `DESC` operators, or `true` (= `ASC`) and `false` (= `DESC`), to define the direction of the order. The default is ascending. Additionally, if you are using a projection, you need to include the `ORDER BY` field in the projection. An aggregate function can be used in `ORDER BY` together with `GROUP BY`, for example `SELECT k, sum(v) FROM T GROUP BY k ORDER BY sum(v) DESC`, and the groups are sorted by the aggregate. An `ARRAY_OF_*` value (such as `ARRAY_OF_INTEGERS`) is ordered element by element, as a list is *(since v26.11.1; before, arrays were ordered by their identity text)*.
- `GROUP BY` and `ORDER BY` may contain comments between the two words (`ORDER /* by id */ BY id`), like anywhere else in the statement.
- **Integer literals** can be written in decimal or hexadecimal (`0xFF`, or `0xFFL` for a long). A hexadecimal value above `2147483647` is a long, it is not read as a negative number.
- `UNWIND` Designates the field on which to unwind the collection.
- `SKIP` Defines the number of records you want to skip from the start of the result-set. You may find this useful in Pagination, when using it in conjunction with `LIMIT`.
- `LIMIT` Defines the maximum number of records in the result-set. You may find this useful in

Pagination, when using it in conjunction with SKIP.

- TIMEOUT Defines the maximum time in milliseconds for the query. By default, queries have no timeouts. If you don't specify a timeout strategy, it defaults to EXCEPTION. These are the available timeout strategies:
  - RETURN Truncate the result-set, returning the data collected up to the timeout.
  - EXCEPTION Raises an exception.

DISTINCT, GROUP BY and count(DISTINCT ⋯) key a temporal value as = reads it, the same rule openCypher grouping uses: a zone-less instant (a DATETIME / LocalDateTime, a java.util.Date, an Instant, a Calendar, a java.sql.Timestamp) is one group with the ZONED_DATETIME of the same instant in UTC, and an OffsetDateTime keys as its zoned form. A zoned datetime keeps its zone, so the same instant in two different zones stays two groups. The elements of a list key the same way.

(since v26.11.1) Before, one instant stored as a DATETIME in one record and as a ZONED_DATETIME in another (possible in a schemaless property) landed in two groups, and counted twice in count(DISTINCT ⋯).

Examples

- Return all records of the type Person, where the name starts with Luk:

```
ArcadeDB> SELECT FROM Person WHERE name LIKE 'Luk%'
```

- The same query, naming the target p and referring to it in the filter:

```
ArcadeDB> SELECT FROM Person p WHERE p.name LIKE 'Luk%'
```

Alternatively, you might also use either of these queries:

```
ArcadeDB> SELECT FROM Person WHERE name.left(3) = 'Luk'
ArcadeDB> SELECT FROM Person WHERE name.substring(0,3) = 'Luk'
```

- Return all records of the type !AnimalType where the collection races contains at least one entry where the first character is e, ignoring case:

```
ArcadeDB> SELECT FROM animaltype WHERE races CONTAINS( name.toLowerCase().subString(
            0, 1) = 'e' )
```

- Return all records of type !AnimalType where the collection races contains at least one entry with names European or Asiatic:

```
ArcadeDB> SELECT * FROM animaltype WHERE races CONTAINS(name in ['European',
            'Asiatic'])
```

- Return all records in the type `Profile` where any field contains the word `danger`:

```
ArcadeDB> SELECT FROM Profile WHERE @this.values().asString() LIKE '%danger%'
```

- Return all results on type `Profile`, ordered by the field `name` in descending order:

```
ArcadeDB> SELECT FROM Profile ORDER BY name DESC
```

- Return the number of records in the type `Account` per city:

```
ArcadeDB> SELECT SUM(*) FROM Account GROUP BY city
```

- Return only a limited set of records:

```
ArcadeDB> SELECT FROM [#10:3, #10:4, #10:5]
```

- Return three fields from the type `Profile`:

```
ArcadeDB> SELECT nick, followings, followers FROM Profile
```

- Return the field `name` in uppercase and the field country name of the linked city of the address:

```
ArcadeDB> SELECT name.toUppercase(), address.city.country.name FROM Profile
```

- Return records from the type `Profile` in descending order of their creation:

```
ArcadeDB> SELECT FROM Profile ORDER BY @rid DESC
```

- Return value of `email` which is stored in a JSON field `data` (type EMBEDDED) of the type `Person`, where the name starts with `Luk`:

```
ArcadeDB> SELECT data.email FROM Person WHERE name LIKE 'Luk%'
```

ArcadeDB can open an inverse cursor against buckets. This is very fast and doesn't require the typical ordering resources, CPU and RAM.

Projections

In the standard implementations of SQL, projections are mandatory. In ArcadeDB, the omission of projects translates to its returning the entire record. That is, it reads no projection as the equivalent of the `*` wildcard.

```
ArcadeDB> SELECT FROM Account
```

For all projections except the wildcard `*`, it creates a new temporary document, which does not include the `@rid` fields of the original record.

```
ArcadeDB> SELECT name, age FROM Account
```

The naming convention for the returned document fields are:

- Field name for plain fields, like `invoice` becoming `invoice`.
- First field name for chained fields, like `invoice.customer.name` becoming `invoice`.
- Function name for functions, like `MAX(salary)` becoming `max`.

In the event that the target field exists, it uses a numeric progression. For instance,

```
ArcadeDB> SELECT MAX(incoming), MAX(cost) FROM Balance

------+------
 max  | max2
------+------
 1342 | 2478
------+------
```

To override the display for the field names, use the `AS`.

```
ArcadeDB> SELECT MAX(incoming) AS max_incoming, MAX(cost) AS max_cost FROM Balance

---------------+----------
 max_incoming  | max_cost
---------------+----------
 1342          | 2478
---------------+----------
```

With the dollar sign `$`, you can access the context variables. Each time you run the command, ArcadeDB accesses the context to read and write the variables. For instance, say you want to display the path and depth levels up to the fifth of a `TRAVERSE` on all records in the `Movie` type.

```
ArcadeDB> SELECT $path, $depth FROM ( TRAVERSE out() FROM Movie MAXDEPTH 5 )
```

14.2.1. `LET` Block

The `LET` block contains context variables to assign each time ArcadeDB evaluates a record. It destroys these values once the query execution ends. You can use context variables in projections,

conditions, and sub-queries.

Assigning Fields for Reuse

ArcadeDB allows for crossing relationships. In single queries, you need to evaluate the same branch of the nested relationship. This is better than using a context variable that refers to the full relationship.

```
ArcadeDB> SELECT FROM Profile WHERE address.city.name LIKE '%Saint%"' AND
          ( address.city.country.name = 'Italy' OR
            address.city.country.name = 'France' )
```

Using the `LET` makes the query shorter and faster, because it traverses the relationships only once:

```
ArcadeDB> SELECT FROM Profile LET $city = address.city WHERE $city.name LIKE
          '%Saint%"' AND ($city.country.name = 'Italy' OR $city.country.name =
'France')
```

In this case, it traverses the path till `address.city` only once.

14.2.2. Sub-query

The `LET` block allows you to assign a context variable to the result of a sub-query.

```
ArcadeDB> SELECT FROM Document LET $temp = ( SELECT @rid, $depth FROM (TRAVERSE
          V.OUT, E.IN FROM $parent.$current ) WHERE @type = 'Concept' AND
          ( id = 'first concept' OR id = 'second concept' )) WHERE $temp.SIZE() > 0
```

`LET` Block in Projection

You can use context variables as part of a result-set in projections. For instance, the query below displays the city name from the previous example:

```
ArcadeDB> SELECT $temp.name FROM Profile LET $temp = address.city WHERE $city.name
          LIKE '%Saint%"' AND ( $city.country.name = 'Italy' OR
          $city.country.name = 'France' )
```

14.2.3. Unwinding

ArcadeDB allows unwinding of collection fields and obtaining multiple records as a result, one for each element in the collection:

```
ArcadeDB> SELECT name, OUT("Friend").name AS friendName FROM Person

--------+------------------
```

```
 name   | friendName
--------+-----------------
 'John' | ['Mark', 'Steve']
--------+-----------------
```

In the event if you want one record for each element in `friendName`, you can rewrite the query using `UNWIND`:

```
ArcadeDB> SELECT name, OUT("Friend").name AS friendName FROM Person UNWIND friendName

--------+------------
 name   | friendName
--------+------------
 'John' | 'Mark'
 'John' | 'Steve'
--------+------------
```

`UNWIND` works the same way on the array types (`SHORT[]`, `INTEGER[]`, `LONG[]`, `FLOAT[]`, `DOUBLE[]`), giving one record per element. A `BINARY` property is left untouched: it is an opaque blob, not a sequence of values, so it is never split into one record per byte.

14.3. SQL - INSERT

The `INSERT` command creates a new record in the database. Records can be schema-less or follow rules specified in your model.

Syntax:

```
INSERT INTO [TYPE:]<type>|BUCKET:<bucket>|INDEX:<index>
  [(<field>[,]*) VALUES (<expression>[,]*)[,]*]|
  [SET <field> = <expression>|<sub-command>[,]*]|
  [CONTENT {<JSON>}|[{<JSON>}[,]*]]
  [ON DUPLICATE KEY SKIP]
  [RETURN <expression>]
  [FROM <query>]
  [RETURNING <expression>]
```

- `SET` Abbreviated syntax to set field values.
- `CONTENT` Defines JSON data as an option to set field values of one (object) or multiple (in an array of objects) records.
- `ON DUPLICATE KEY SKIP` When inserting multiple records, skips any record that would violate a **unique** index instead of aborting the whole command. Each result row tells you whether that record was inserted or skipped. Any other kind of error (for example, a missing mandatory field) still aborts the command as usual.
- `RETURN` Defines an expression to return instead of the number of inserted records. You can use

any valid SQL expression. The most common use-cases,

  - `@rid` Returns the Record ID of the new record.
  - `@this` Returns the entire new record.

- `FROM` Defines where you want to insert the result-set.
- `RETURNING` PostgreSQL's spelling of `RETURN`, written at the end of the statement as PostgreSQL does: it returns the inserted records projected by `<expression>`, and `RETURNING *` returns all their properties. `RETURN` and `RETURNING` cannot be combined in the same statement. *(Since v26.12.1, issue #11392.)*

To insert only documents, vertices, or edges with a certain **unique** property, use a unique index.

(Since v26.12.1.) In the `SET`, `VALUES` and `CONTENT` expressions the new record has no RID yet: `@rid` reads `null` there (it used to fail with a `NullPointerException`), and `RETURN @rid` gives the RID once the record is saved. For the same reason a record cannot hold a link to itself while it is created: `INSERT INTO D SET me = @this` is refused (it used to store a link to a position that is no record). Insert the record, then set the link with `UPDATE D SET me = @this WHERE …`.

`INSERT INTO <timeseries type>` is the one `INSERT` target that is **not atomic with the transaction containing it**. The samples are committed as they are appended: they are durable and visible to every other connection as soon as the statement returns, and rolling the transaction back does not take them back. Every other `INSERT` target behaves normally, which is exactly why this one surprises people. See Appends and Transactions for why, and for the same statement about the other ingestion paths.

If multiple documents are inserted as JSON "CONTENT", and one document causes an error, for example due to a schema error, the remaining documents in the JSON array are not inserted. Add `ON DUPLICATE KEY SKIP` if you only want to skip records that clash with a unique index, instead of aborting the rest of the batch.

Examples

- Inserts a new record with the name `Jay` and surname `Miner`.

As an example, in the SQL-92 standard, such as with a Relational database, you might use:

```
ArcadeDB> INSERT INTO Profile (name, surname)
           VALUES ('Jay', 'Miner')
```

Alternatively, in the ArcadeDB abbreviated syntax, the query would be written as,

```
ArcadeDB> INSERT INTO Profile SET name = 'Jay', surname = 'Miner'
```

In JSON content syntax, it would be written as this,

```
ArcadeDB> INSERT INTO Profile CONTENT {"name": "Jay", "surname": "Miner"}
```

- Insert a new record of the type `Profile`, but in a different bucket from the default.

In SQL-92 syntax:

```
ArcadeDB> INSERT INTO Profile BUCKET profile_recent (name, surname) VALUES
            ('Jay', 'Miner')
```

Alternative, in the ArcadeDB abbreviated syntax:

```
ArcadeDB> INSERT INTO Profile BUCKET profile_recent SET name = 'Jay',
            surname = 'Miner'
```

- Insert several records at the same time:

```
ArcadeDB> INSERT INTO Profile (name, surname) VALUES ('Jay', 'Miner'),
            ('Frank', 'Hermier'), ('Emily', 'Sout')
```

again in JSON content syntax, a JSON array of the to-be-inserted objects is passed:

```
ArcadeDB> INSERT INTO Profile CONTENT [{"name": "Jay", "surname": "Miner"},
            {"name": "Frank", "surname": "Hermier"},{"name": "Emily", "surname":
"Sout"}]
```

- Insert several records, skipping any that clash with a unique index:

```
ArcadeDB> INSERT INTO Product CONTENT [{"sku": "A1", "name": "Widget"},
            {"sku": "A2", "name": "Gadget"}] ON DUPLICATE KEY SKIP
```

If `Product.sku` is a unique index and a record with `sku = 'A1'` already exists, that record is skipped and the rest are still inserted.

- Insert a new record, adding a relationship.

In SQL-92 syntax:

```
ArcadeDB> INSERT INTO Employee (name, boss) VALUES ('jack', #11:9)
```

In the ArcadeDB abbreviated syntax:

```
ArcadeDB> INSERT INTO Employee SET name = 'jack', boss = #11:99
```

- Insert a new record, add a collection of relationships.

In SQL-93 syntax:

```
ArcadeDB> INSERT INTO Profile (name, friends) VALUES ('Luca', [#10:3, #10:4])
```

In the ArcadeDB abbreviated syntax:

```
ArcadeDB> INSERT INTO Profiles SET name = 'Luca', friends = [#10:3, #10:4]
```

- Inserts using `SELECT` sub-queries

```
ArcadeDB> INSERT INTO Diver SET name = 'Luca', buddy = (SELECT FROM Diver
            WHERE name = 'Marko')
```

- Inserts using `INSERT` sub-queries:

```
ArcadeDB> INSERT INTO Diver SET name = 'Luca', buddy = (INSERT INTO Diver
            SET name = 'Marko')
```

- Inserting into a different bucket:

```
ArcadeDB> INSERT INTO BUCKET:asiaemployee (name) VALUES ('Matthew')
```

However, note that the document has no assigned type. To create a document of a certain type, but in a different bucket than the default, instead use:

```
ArcadeDB> INSERT INTO BUCKET:asiaemployee (@type, content) VALUES
            ('Employee', 'Matthew')
```

That inserts the document of the type `Employee` into the bucket `asiaemployee`.

- Insert a new record, adding it as an embedded document:

```
ArcadeDB> INSERT INTO Profile (name, address) VALUES ('Luca', { "@type": "d",
            "street": "Melrose Avenue"})
```

- Insert a record with a list property:

```
ArcadeDB> INSERT INTO Profile SET friends = list("Joe","Jack","John")
```

- Insert a record with a map property:

```
ArcadeDB> INSERT INTO Profile SET address = map("home","Avenue Lane")
```

- Insert from a query.

To copy records from another type, use:

```
ArcadeDB> INSERT INTO GermanyClient FROM SELECT FROM Client WHERE
            country = 'Germany'
```

This inserts all the records from the type `Client` where the country is Germany, in the type `GermanyClient`.

To copy records from one type into another, while adding a field:

```
ArcadeDB> INSERT INTO GermanyClient FROM SELECT *, true AS copied FROM Client
            WHERE country = 'Germany'
```

This inserts all records from the type `Client` where the country is Germany into the type `GermanClient`, with the addition field `copied` to the value `true`.

Each new record holds exactly the properties of the row the `SELECT` produced, so the projection decides what is copied: `SELECT *, !secret` leaves `secret` out and `SELECT *, upper(name) AS name` writes the computed `name`. A row read from a vertex still creates a vertex.

```
ArcadeDB> INSERT INTO PublicClient FROM SELECT *, !password FROM Client
```

Before v26.10.2 a row backed by a record (every `SELECT *` shape) copied the record instead of the row, so an excluded column was copied anyway and an overriding alias was written with the stored value.

The `SELECT` is evaluated on the data as it was when the statement started, as SQL requires, so it never reads the records the statement inserts. That matters when it reads the type it inserts into, or one of its super types:

```
ArcadeDB> INSERT INTO Client FROM SELECT id + 1000 AS id, name FROM Client WHERE id >=
4
```

copies each matching client once. ArcadeDB drains such a `SELECT` before the first insert, holding its rows in memory, and only when the `SELECT` can read a bucket the statement writes to (or nests a

subquery): a copy between unrelated types streams the rows as before. The rows held count against the heap budget all queries share, so copying a very large type into itself can be refused with a heap-limit error; copy it in ranges (`WHERE id BETWEEN …`) or through an intermediate type instead.

Before v26.11.1 such a `SELECT` could read its own inserts: with an `OR` filter a row was copied twice, and with an index range on the filtered property (`WHERE id >= 4` over an index on `id`) the statement never ended (issue #10192).

- Insert a vertex.

Besides the specialized command CREATE VERTEX, vertices and edges can be inserted via the `INSERT` command:

```
ArcadeDB> INSERT INTO MyVertexType SET name = 'John Doe'
```

However, edges have to be created with CREATE EDGE.

- Insert a record and return its Record ID and name, PostgreSQL style:

```
ArcadeDB> INSERT INTO Customer (name, age) VALUES ('Jay', 31) RETURNING @rid, name
```

This is what PostgreSQL clients rely on to read back generated keys: the PostgreSQL JDBC driver appends `RETURNING *` to a statement prepared with `Statement.RETURN_GENERATED_KEYS` (and `RETURNING "col"` for the columns named in `prepareStatement(sql, String[])`), so Hibernate `IDENTITY` ids, Spring's `KeyHolder` and jOOQ's generated keys work over the PostgreSQL protocol. `returning` can still be used as a property name.

14.4. SQL - UPDATE

Update one or more records in the current database. Remember: ArcadeDB can work in schema-less mode, so you can create any field on-the-fly. Furthermore, the command also supports extensions to work on collections.

Syntax:

```
UPDATE <type>|BUCKET:<bucket>|<recordID>
  [SET|REMOVE <field-name> = <field-value>[,]*]|[CONTENT|MERGE <JSON>]
  [UPSERT]
  [APPLY DEFAULTS]
  [RETURN <returning> [<returning-expression>]]
  [WHERE <conditions>]
  [LIMIT <max-records>] [BATCH <batch-size>] [TIMEOUT <MilliSeconds>]
  [RETURNING <returning-expression>]
```

- `BUCKET:<bucket>` Restricts the update to the records of one bucket, given by name or id. The name can also be an input parameter, `BUCKET::name` or `BUCKET:?`, also with `UPSERT`, which then

creates the record in that bucket. An unknown bucket is refused with an error that names it. *(Since v26.12.1, issue #12480: earlier versions failed with a* `NullPointerException` *on a bucket parameter.)*

- **`SET`** Defines the fields to update. On an edge, `SET @in = <rid>` / `SET @out = <rid>` re-points the edge to another vertex. The other metadata names (`@rid`, `@type`, `@cat`, `@props`, `@this`) are not properties and are refused (since v26.10.2: earlier versions stored them as properties no query could read back, and reported the record as updated).

  Every right-hand side reads the record as it was before the statement, as in SQL, so `UPDATE T SET a = b, b = a` swaps the two columns and `UPDATE T SET b = 7, a = b` gives `a` the old value of `b` (*since v26.11.1*: earlier versions applied the items left to right, so a later item saw the new value of an earlier one and the swap left both columns equal). Two items on the same property still apply in order, and a compound operator extends the value its property holds by then: `SET a = 1, a += 2` gives 3.

- **`REMOVE`** Removes an item in collection and map fields or a property.

  A property holding an `ARRAY_OF_*` value (such as `ARRAY_OF_LONGS`, which is what a list of numbers nested in a map parameter sent over HTTP becomes) is a list for `SET l += x`, `SET l -= x`, `l + [x]`, `l - x`, `REMOVE l[i]` and `REMOVE l = x`. The changed value is stored as a `LIST`, or as an array again on a property declared with an `ARRAY_OF_*` type; a `REMOVE` that takes nothing out leaves the array as it is. A `BINARY` value is not a list. *(Since v26.12.1, issue #12896: before,* `SET l += 3` *stored a string made of the array's identity,* `SET l -= 1` *removed the property and both* `REMOVE` *forms failed.)*

- **`CONTENT`** Replaces the record content with a JSON document.
- **`MERGE`** Merges the record content with a JSON document. Like `CONTENT`, a key starting with `@` is metadata and is skipped, except an edge's `@in` / `@out`, which re-point it. So a document read back as JSON, `@rid`, `@type` and `@cat` included, can be merged back as it is (since v26.10.2: `MERGE` refused a reserved key, and no JSON literal could carry a `"@rid"` key at all).
- **`UPSERT`** Updates a record if it exists or inserts a new record if it doesn't. This avoids the need to execute two commands, (one for each condition, inserting and updating). `UPSERT` requires a `WHERE` clause and a type target. There are further limitations on `UPSERT`, explained below. Practically `UPSERT` means: `UPDATE` if the `WHERE` condition is fulfilled, otherwise `INSERT`. `UPSERT` works on document types **and on vertex types**: when the target is a vertex type and no existing record matches, a brand new vertex (not a plain document) is created, with all the graph capabilities (incoming/outgoing edges, `@in`/`@out` projections, etc.). For edge types, `UPSERT` is not available because edges always require `FROM` and `TO` endpoints; use `CREATE EDGE … IF NOT EXISTS` instead.
- `APPLY DEFAULTS` applies a properties' default attribute on an update again.
- `RETURN` Specifies an expression to return instead of the record and what to do with the result-set returned by the expression. The available return operators are:
  - `COUNT` Returns the number of updated records. This is the default return operator.
  - `BEFORE` Returns the records before the update.
  - `AFTER` Return the records after the update.
- `WHERE` Defines the subset of records to be updated.

- `LIMIT` Defines the maximum number of records to update.

When an index serves the `WHERE` and the statement changes a property of that index (for example `UPDATE V SET a = a + 10 WHERE a BETWEEN 0 AND 1000`), the set of records to update is fixed before the first one is changed. Each record is updated once, even when its new key falls inside the range. Records are then updated in storage order rather than in index order, which can change the order of the rows returned by `RETURN BEFORE` and `RETURN AFTER`.

- `BATCH` Defines the number of records to update in each transaction batch. When set, the command commits the current transaction every `<batch-size>` records and begins a new one. This is essential for updating large datasets (e.g., millions of records) that would otherwise consume too much memory in a single transaction.

Because `BATCH` commits as it goes, an `UPDATE ⋯ BATCH` is not atomic: a failure part way through leaves the records committed up to the last batch boundary in place. For the same reason, running it inside a transaction opened by your application makes that transaction no longer atomic either, and a rollback cannot undo what the batches already committed. A `transaction()` block with retries does not retry in that case: the conflict is propagated to your application instead, since repeating the block would apply the already committed records a second time. A SQL Script `BEGIN ⋯ COMMIT RETRY <n>` block behaves the same way.

- `TIMEOUT` Defines the time you want to allow the update run before it times out.
- `RETURNING` PostgreSQL's trailing form of `RETURN AFTER`: returns the records as the update left them, projected by `<returning-expression>` (`RETURNING *` for all their properties). `RETURN` and `RETURNING` cannot be combined in the same statement. *(Since v26.12.1, issue #11392.)*

Examples

- Update to change the value of a field:

```
ArcadeDB> UPDATE Profile SET nick = 'Luca' WHERE nick IS NULL
```

- Update to remove a field from all records:

```
ArcadeDB> UPDATE Profile REMOVE nick
```

- Update to remove a value from a collection, if you know the exact value that you want to remove:

Remove an element from a link list or set:

```
ArcadeDB> UPDATE Account REMOVE address = #12:0
```

Remove an element from a list or set of strings:

```
ArcadeDB> UPDATE Account REMOVE addresses = 'Foo'
```

Append an element to a list or set of strings:

```
ArcadeDB> UPDATE Account SET addresses += 'Foo'
```

- Update to remove a value, filtering on value attributes.

Remove addresses based in the city of Rome:

```
ArcadeDB> UPDATE Account REMOVE addresses = addresses[city = 'Rome']
```

This works on a list of links, of embedded documents and of maps. A map can also be removed by writing it: it removes the first element of the list equal to it, entry by entry (`{'city': 'Rome'}` removes `{'city': 'Rome'}`, not `{'city': 'Rome', 'zip': '00100'}`). *(Since v26.12.1; before, a list of maps was left unchanged and the keys of the map were removed from the list instead)*

```
ArcadeDB> UPDATE Account REMOVE addresses = {'city': 'Rome'}
```

- Update to remove a value, filtering based on position in the collection.

```
ArcadeDB> UPDATE Account REMOVE addresses = addresses[1]
```

This remove the second element from a list, (position numbers start from `0`, so `addresses[1]` is the second element).

- Update a map entry

```
ArcadeDB> UPDATE #87:0 SET pages += { "homePage": "doctor.html" }
```

- Update a single map key or list position in place

```
ArcadeDB> UPDATE Account SET settings.theme = 'dark'
ArcadeDB> UPDATE Account SET settings['theme'] = 'dark'
ArcadeDB> UPDATE Account SET addresses[0] = 'Foo'
```

A positional assignment past the end of a collection, such as `addresses[5] = 'Foo'` on a 3-element list, pads the missing positions with `null`. A property whose value is a **set** is the exception: a set cannot hold `null` more than once, so several missing positions collapse into one and the value lands right after it - `[a, b, c, null, Foo]`

> rather than `[a, b, c, null, null, Foo]`. Declare the property as `LIST` if every position must be addressable.

- Update to remove a value from a map

```
ArcadeDB> UPDATE Account REMOVE addresses = 'Luca'
```

- Update to remove a property values from records

```
ArcadeDB> UPDATE Account REMOVE addresses WHERE addresses = 'unknown'
```

- Update an embedded document. The `UPDATE` command can take JSON as a value to update.

```
ArcadeDB> UPDATE Account SET address={ "street": "Melrose Avenue", "city": {
            "name": "Beverly Hills" } }
```

- Update the first twenty records that satisfy a condition:

```
ArcadeDB> UPDATE Profile SET nick = 'Luca' WHERE nick IS NULL LIMIT 20
```

- Update millions of records in batches of 10,000 to avoid loading all records into a single transaction:

```
ArcadeDB> UPDATE Profile SET verified = true WHERE verified IS NULL BATCH 10000
```

- Update a record or insert if it doesn't already exist:

```
ArcadeDB> UPDATE Profile SET nick = 'Luca' UPSERT WHERE nick = 'Luca'
```

- Upsert a vertex by a natural key. With a `UNIQUE` index on `Person.email`, this either updates the existing person or creates a new vertex (not a plain document) of type `Person`:

```
ArcadeDB> UPDATE Person SET name = 'Luca', email = 'luca@example.com'
            UPSERT WHERE email = 'luca@example.com'
```

The newly created record is a real vertex, so it can immediately participate in graph traversals and be the source/target of `CREATE EDGE`.

- Updates using the `RETURN` keyword:

```
ArcadeDB> UPDATE #7:0 SET gender='male' RETURN AFTER @rid
ArcadeDB> UPDATE #7:0 SET gender='male' RETURN AFTER @this
```

```
ArcadeDB> UPDATE #7:0 SET gender='male' RETURN AFTER
$current.exclude("really_big_field")
```

- The same with the PostgreSQL `RETURNING` clause, at the end of the statement:

```
ArcadeDB> UPDATE Profile SET nick = 'Luca' WHERE nick IS NULL RETURNING @rid, nick
```

In the event that a single field is returned, ArcadeDB wraps the result-set in a record storing the value in the field `result`. This avoids introducing a new serialization, as there is no primitive values collection serialization in the binary protocol. Additionally, it provides useful fields like `version` and `rid` from the original record in corresponding fields. The new syntax allows for optimization of client-server network traffic.

For more information on SQL syntax, see `SELECT`.

Limitations of the `UPSERT` Clause

The `UPSERT` clause only guarantees atomicity when you use a `UNIQUE` index and perform the look-up on the index through the `WHERE` condition. This applies to both document and vertex types.

```
ArcadeDB> UPDATE Client SET id = 23 UPSERT WHERE id = 23
```

Here, you must have a unique index on `Client.id` to guarantee uniqueness on concurrent operations.

The same rule applies when upserting vertices: define a `UNIQUE` index on the natural key first.

```
ArcadeDB> CREATE INDEX ON Person (email) UNIQUE
ArcadeDB> UPDATE Person SET name = 'Luca', email = 'luca@example.com'
            UPSERT WHERE email = 'luca@example.com'
```

Additional restrictions:

- the `WHERE` condition must be a non-null equality on **every** property of a `UNIQUE` index. A non-unique index, a partial match of a composite index (key prefix), a range, an `IN` list or a null value is rejected with the error "Upsert must involve a UNIQUE index to retrieve the records", because it could create duplicate records. *(Since v26.10.2)*
- the `WHERE` condition cannot use `OR`. Only an `AND` chain of equality conditions is supported, because each equality term is also reused as the initial value of the property when a new record has to be created.
- `UPSERT` is not available on edge types. The statement is refused with an error that names the edge type, whether the target is given as a type or as one of its buckets, also when a matching edge already exists, and nothing is written (`EXPLAIN` refuses it too). To create an edge only if it does not already exist, use `CREATE EDGE … IF NOT EXISTS`. *(Since v26.12.1)*

14.5. SQL - DELETE

Removes one or more records from the database. You can refine the set of records that it removes using the WHERE clause.

Syntax:

```
DELETE [VERTEX] FROM <Type>|BUCKET:<bucket>|INDEX:<index> [RETURN <returning>]
  [WHERE <Condition>*] [LIMIT <MaxRecords>] [BATCH <batch-size>]
  [TIMEOUT <MilliSeconds>] [UNSAFE] [RETURNING <projection>]
```

- VERTEX Deletes only vertices, each with its edges. A target type that is not a vertex type, or an INDEX: target, is refused before anything runs. A record that is not a vertex (reached through a RID, a bucket or a subquery) is not deleted: the statement fails on it, and the vertices it deleted before are rolled back with the statement, except the batches a BATCH already committed. Without it, DELETE FROM deletes any record, and a vertex is deleted with its edges all the same. *(Since v26.12.1, issue #11620: the keyword used to be accepted and ignored.)*
- RETURN Defines what values the database returns. It takes one of the following values:
  - COUNT Returns the number of deleted records. This is the default option.
  - BEFORE Returns the number of records before the removal.
- WHERE Filters to the records you want to delete.
- LIMIT Defines the maximum number of records to delete.
- BATCH Defines the number of records to delete in each transaction batch. When set, the command commits the current transaction every <batch-size> records and begins a new one. This is essential for deleting large datasets (e.g., millions of records) that would otherwise consume too much memory in a single transaction.
- TIMEOUT Defines the time period to allow the operation to run, before it times out.
- UNSAFE no use, is kept only for compatibility with OrientDB SQL and it could be removed in the future versions of the SQL language.
- RETURNING PostgreSQL's trailing form: returns the deleted records as they were before the delete, projected by <projection> (RETURNING * for all their properties), where RETURN BEFORE returns each whole record. RETURN and RETURNING cannot be combined in the same statement. *(Since v26.12.1, issue #11392.)*

The statement deletes the set of records its source names, each once. A source that names one record in several rows, such as DELETE VERTEX FROM (SELECT expand(out()) FROM V WHERE k = 0) over two parallel edges to the same neighbour, a list of RIDs or a parameter, deletes, counts and returns that record once: the later rows naming it are skipped. Parallel lightweight edges share one RID but are distinct edges, so each is still deleted. *(Since v26.12.1, issue #12256: the second row naming a record failed the whole statement with a not-found error.)*

On a TIMESERIES type (since 26.11.1) DELETE removes samples, and it does not take part in the transaction it runs in: the samples are gone, durably, when the statement returns, and a ROLLBACK does not bring them back (the result carries

> `transactional: false`). Its `WHERE` is a time range on the timestamp column and equalities on `TAG` columns joined by `AND`; a predicate on a `FIELD` column, an `OR`, `LIMIT`, `BATCH` and `RETURN BEFORE` are refused. See Deleting Samples.

Examples:

- Delete all records with the surname `unknown`, ignoring case:

```
ArcadeDB> DELETE FROM Profile WHERE surname.toLowerCase() = 'unknown'
```

- Delete all records of the type `Document`, due to an improper JSON import, or record creation command (note the use of the backticks):

```
ArcadeDB> DELETE FROM `Document`
```

- Delete all records from a large type in batches of 20,000 to avoid loading millions of records into a single transaction:

```
ArcadeDB> DELETE FROM LargeDataset BATCH 20000
```

- Delete matching records from a large type with a condition, committing every 10,000 records:

```
ArcadeDB> DELETE FROM LogEntry WHERE createdAt < '2025-01-01' BATCH 10000
```

- Delete the profiles with the surname `unknown` and return the Record ID and name each one had:

```
ArcadeDB> DELETE FROM Profile WHERE surname = 'unknown' RETURNING @rid, name
```

14.6. SQL - MATCH

Queries the database in a declarative manner, using pattern matching (inspired by Cypher).

Let's start with some examples.

The following examples are based on this sample data-set from the type `People`:

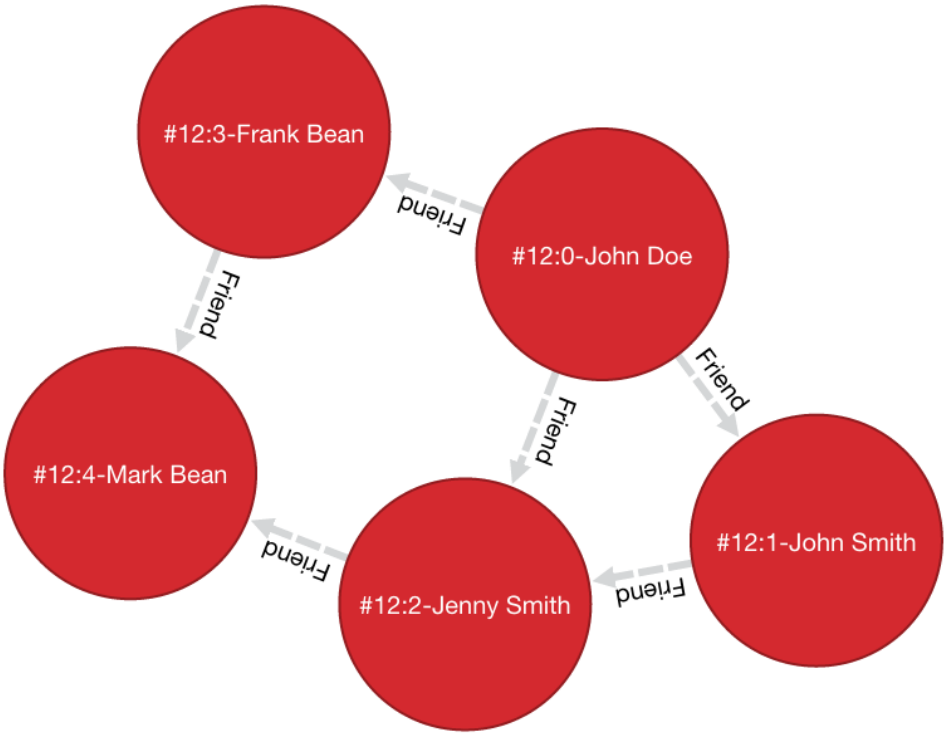
COMMAND

select from person

METADATA			PROPERTIES			IN	OUT
@rid	@version	@class	surname	name	fullName	Friend	Friend
#12:0	8	Person	Doe	John	#12:0-John Doe		#12:1 #12:3 #12:2
#12:1	7	Person	Smith	John	#12:1-John Smith	#12:0	#12:2
#12:2	8	Person	Smith	Jenny	#12:2-Jenny Smith	#12:1 #12:0	#12:4
#12:3	7	Person	Bean	Frank	#12:3-Frank Bean	#12:0	#12:4
#12:4	7	Person	Bean	Mark	#12:4-Mark Bean	#12:3 #12:2	

10 25 50 100 1000 5000

Query executed in 0.009 sec. Returned 5 record(s). Limit: 20 (change it) Table Raw



- Find all people with the name John:

```
ArcadeDB> MATCH {type: Person, as: people, where: (name = 'John')}
            RETURN people

---------
  people
---------
  #12:0
```

```
  #12:1
---------
```

- Find all people with the name John and the surname Smith:

```
ArcadeDB> MATCH  {type: Person, as: people, where: (name = 'John' AND surname =
'Smith')}
            RETURN people

-------
people
-------
 #12:1
-------
```

- Find people named John with their friends:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John')}.both('Friend') {as:
friend}
            RETURN person, friend

--------+---------
 person | friend
--------+---------
 #12:0  | #12:1
 #12:0  | #12:2
 #12:0  | #12:3
 #12:1  | #12:0
 #12:1  | #12:2
--------+---------
```

- Find friends of friends:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John' AND surname = 'Doe')}
            .both('Friend').both('Friend') {as: friendOfFriend}
            RETURN person, friendOfFriend

--------+----------------
 person | friendOfFriend
--------+----------------
 #12:0  | #12:0
 #12:0  | #12:1
 #12:0  | #12:2
 #12:0  | #12:3
 #12:0  | #12:4
--------+----------------
```

- Correlate two patterns by a property, here the files whose `path` equals the `filePath` stored on the matched source:

```
ArcadeDB> MATCH {type: Source, as: s, where: (id = 's1')},
            {type: File, as: f, where: (path = $matched.s.filePath)}
          RETURN f.path AS path, $matched.s.filePath AS sourcePath
```

- Find people, excluding the current user:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John' AND
            surname = 'Doe')}.both('Friend').both('Friend'){as: friendOfFriend,
            where: ($matched.person != $currentMatch)}
            RETURN person, friendOfFriend

--------+----------------
 person | friendOfFriend
--------+----------------
 #12:0  | #12:1
 #12:0  | #12:2
 #12:0  | #12:3
 #12:0  | #12:4
--------+----------------
```

- Find friends of friends to the sixth degree of separation:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John' AND
            surname = 'Doe')}.both('Friend'){as: friend,
            where: ($matched.person != $currentMatch), while: ($depth < 6)}
            RETURN person, friend

--------+---------
 person | friend
--------+---------
 #12:0  | #12:0
 #12:0  | #12:1
 #12:0  | #12:2
 #12:0  | #12:3
 #12:0  | #12:4
--------+---------
```

- Finding friends of friends to six degrees of separation, since a particular date:

```
ArcadeDB> MATCH {type: Person, as: person,
            where: (name = 'John')}.(bothE('Friend'){
            where: (date < ?)}.bothV()){as: friend,
            while: ($depth < 6)} RETURN person, friend
```

In this case, the condition `$depth < 6` refers to traversing the block `bothE('Friend')` six times.

- Find friends of my friends who are also my friends, using multiple paths:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John' AND
            surname = 'Doe')}.both('Friend').both('Friend'){as: friend},
            { as: person }.both('Friend'){ as: friend }
            RETURN person, friend

--------+--------
 person | friend
--------+--------
 #12:0  | #12:1
 #12:0  | #12:2
--------+--------
```

In this case, the statement matches two expression: the first to friends of friends, the second to direct friends. Each expression shares the common aliases (`person` and `friend`). To match the whole statement, the result must match both expressions, where the alias values for the first expression are the same as that of the second.

- Find common friends of John and Jenny:

```
ArcadeDB> MATCH {type: Person, where: (name = 'John' AND
            surname = 'Doe')}.both('Friend'){as: friend}.both('Friend')
            {type: Person, where: (name = 'Jenny')} RETURN friend

--------
 friend
--------
 #12:1
--------
```

The same, with two match expressions:

```
ArcadeDB> MATCH {type: Person, where: (name = 'John' AND
            surname = 'Doe')}.both('Friend'){as: friend},
            {type: Person, where: (name = 'Jenny')}.both('Friend')
            {as: friend} RETURN friend
```

14.6.1. Simplified Syntax

```
MATCH
  {
    [type: <type>],
    [bucket: <bucket>],
```

```
    [rid: <rid>],
    [as: <alias>],
    [where: (<whereCondition>)]
  }
  .<functionName>(){
    [type: <typeName>],
    [as: <alias>],
    [where: (<whereCondition>)],
    [while: (<whileCondition>)],
    [maxDepth: <number>],
    [depthAlias: <identifier> ],
    [pathAlias: <identifier> ],
    [optional: (true | false)]
  }*
  [,
    [NOT]
    {
      [as: <alias>],
      [type: <type>],
      [where: (<whereCondition>)]
    }
    .<functionName>(){
      [type: <typeName>],
      [as: <alias>],
      [where: (<whereCondition>)],
      [while: (<whileCondition>)],
      [maxDepth: <number>],
      [depthAlias: <identifier> ],
      [pathAlias: <identifier> ],
      [optional: (true | false)]
    }*
  ]*
RETURN [DISTINCT] <expression> [ AS <alias> ] [, <expression> [ AS <alias> ]]*
[ GROUP BY <expression> [, <expression>]* ]
[ ORDER BY <expression> [, <expression>]* ]
[ UNWIND <Field>* ]
[ SKIP <number> ]
[ LIMIT <number> ]
```

- **`<type>`** Defines a valid target type.
- **`bucket: <bucket>`** Restricts the node to the records of one bucket.
- **`rid: <rid>`** Binds the node to the record with that RID, written as a literal (`#12:0`) or a parameter (`:rid`, `?`). A node bound by RID is the preferred starting point of the pattern. When the same node also declares a `type:` or a `bucket:`, the record must satisfy them as well: `{type: Person, rid: #12:0}` matches only if `#12:0` is a `Person` or a subtype of it, and matches nothing otherwise.
- **`as: <alias>`** Defines an alias for a node in the pattern. An alias written in several places (in more than one comma-separated pattern, or twice in a cyclic path) is one node, so the `where:` of every occurrence applies to it, whichever occurrence the engine reaches it through: `MATCH {type: V,`

`as: z}.out(){as: y}, {as: y, where: (c = 0)}` returns only the rows where `y.c = 0`. The one exception is a `where:` that reads `$depth`, which belongs to the `while:` or `maxDepth:` hop it is written on. *Since v26.12.1*: earlier versions applied such a `where:` only when the engine reached the node through the occurrence that carried it, and could return rows that violate it.

- **<whereCondition>** Defines a filter condition to match a node in the pattern. It supports the normal SQL `WHERE` clause. You can also use the `$currentMatch` and `$matched` context variables. A condition that is a single value rather than a comparison, such as a function call (`where: (isActive(@this))`) or a boolean property (`where: (active)`), matches when the value is `true`, as in a SQL `WHERE`; the same applies to `while:`. *Since v26.12.1*: earlier versions silently ignored such a `where:` or `while:`.
- **<functionName>** Defines a graph function to represent the connection between two nodes. For instance, `out()`, `in()`, `outE()`, `inE()`, etc. For out(), in(), both() also a shortened *arrow* syntax is supported:
- `{…}.out(){…}` can be written as `{…}-->{…}`
- `{…}.out("EdgeType"){…}` can be written as `{…}-EdgeType->{…}`
- `{…}.in(){…}` can be written as `{…}<--{…}`
- `{…}.in("EdgeType"){…}` can be written as `{…}<-EdgeType-{…}`
- `{…}.both(){…}` can be written as `{…}--{…}`
- `{…}.both("EdgeType"){…}` can be written as `{…}-EdgeType-{…}`, or as `{…}<-EdgeType->{…}`, a head on both sides reading as either direction like openCypher's `<-[:EdgeType]->` Every arrow form means the same inside a nested path `.( )` as at the top level of the pattern. *Since v26.12.1*: earlier versions read `.(<-EdgeType->)` as `.(in("EdgeType"))`.
- **<whileCondition>** Defines a condition that the statement must meet to allow the traversal of this path. It supports the normal SQL `WHERE` clause. You can also use the `$currentMatch`, `$matched` and `$depth` context variables. For more information, see Deep Traversal While Condition, below.
- **<maxDepth>** Defines the maximum depth for this single path.
- **<depthAlias>** This is valid only if you have a `while` or a `maxDepth`. It defines the alias to be used to store the depth of this traversal. This alias can be used in the `RETURN` block to retrieve the depth of current traversal.
- **<pathAlias>** This is valid only if you have a `while` or a `maxDepth`. It defines the alias to be used to store the elements traversed to reach this alias. This alias can be used in the `RETURN` block to retrieve the elements traversed to reach this alias.
- **`RETURN <expression> [ AS <alias> ]`** Defines elements in the pattern that you want returned. It can use one of the following:
- Aliases defined in the `as:` block.
- `$matches` Indicating all defined aliases.
- `$paths` Indicating the full traversed paths.
- `$elements` Indicating that all the elements that would be returned by the $matches have to be returned flattened, without duplicates.
- `$pathElements` Indicating that all the elements that would be returned by the $paths have to be returned flattened, without duplicates.

- `optional` if set to true, allows to evaluate and return a pattern even if that particular node does not match the pattern itself (ie. there is no value for that node in the pattern). In current version, `optional` is supported only on right terminal nodes, eg. `{} --> {optional:true}`. On any other node (the first node or a node in the middle of the pattern, eg. `{optional:true} <-- {}`) the engine does not raise an error, but the flag is not applied, so do not rely on it there.
- `NOT` **patterns** Together with normal patterns, you can also define negative patterns. A result will be returned only if it also DOES NOT match any of the negative patterns, ie. if it matches at least one of the negative patterns it won't be returned.
- `UNWIND` Designates the field on which to unwind the collection.

14.6.2. Arrow notation

`out()`, `in()` and `both()` operators can be replaced with arrow notation `-->`, `<--` and `--`

Eg. the query

```
MATCH {type: V, as: a}.out(){}.out(){}.out(){as:b}
RETURN a, b
```

can be written as

```
MATCH {type: V, as: a}-->{}-->{}-->{as:b}
RETURN a, b
```

Eg. the query (things that belong to friends)

```
MATCH {type: Person, as: a}.out('Friend'){as:friend}.in('BelongsTo'){as:b}
RETURN a, b
```

can be written as

```
MATCH {type: Person, as: a}-Friend->{as:friend}<-BelongsTo-{as:b}
RETURN a, b
```

Using arrow notation the curly braces are mandatory on both sides. eg:

```
MATCH {type: Person, as: a}-->{}-->{as:b} RETURN a, b  -- is allowed

MATCH {type: Person, as: a}-->-->{as:b} RETURN a, b  -- is NOT allowed

MATCH {type: Person, as: a}.out().out(){as:b} RETURN a, b  -- is allowed

MATCH {type: Person, as: a}.out(){}.out(){as:b} RETURN a, b  -- is allowed
```

Negative (NOT) patterns

Together with normal patterns, you can also define negative patterns. A result will be returned only if it also DOES NOT match any of the negative patterns, ie. if the result matches at least one of the negative patterns it won't be returned.

As an example, consider the following problem: given a social network, choose a single person and return all the people that are friends of their friends, but that are not their direct friends.

The pattern can be calculated as follows:

```
MATCH
  {type:Person, as:a, where:(name = "John")}-FriendOf->{as:b}-FriendOf-> {as:c},
  NOT {as:a}-FriendOf->{as:c}
RETURN c.name
```

DISTINCT

The `MATCH` statement returns all the occurrences of a pattern, even if they are duplicated. To have unique, distinct records as a result, you have to specify the `DISTINCT` keyword in the `RETURN` statement.

Example: suppose you have a dataset made like following:

```
INSERT INTO V SET name = 'John', surname = 'Smith';
INSERT INTO V SET name = 'John', surname = 'Harris'
INSERT INTO V SET name = 'Jenny', surname = 'Rose'
```

This is the result of the query without a `DISTINCT` clause:

```
ArcadeDB> MATCH {type: Person, as:p} RETURN p.name as name

--------
 name
--------
 John
--------
 John
--------
 Jenny
--------
```

And this is the result of the query with a DISTINCT clause:

```
ArcadeDB> MATCH {type: Person, as:p} RETURN DISTINCT p.name as name

--------
```

```
 name
--------
 John
--------
 Jenny
--------
```

ORDER BY on the rows of a `RETURN`

When `RETURN` lists projections (or returns `$paths` or `$patterns`), every row is a new object built from the matched aliases: it is not a record and has no `@rid`, `@type` or other record attribute of its own. Order by the attribute of an alias, or by a projected column:

```
MATCH {type: Note, as: n}-HasPatient->{as: p}
RETURN n.@rid AS noteRid, p.name AS patient
ORDER BY noteRid           -- or ORDER BY n.@rid
```

A bare record attribute such as `ORDER BY @rid` is refused on these rows with an error that suggests the two forms above (a `CommandSQLParsingException`, answered `400` over HTTP since v26.12.1, `500` before). With `RETURN $elements` or `$pathElements` the rows are the records themselves, so `ORDER BY @rid` sorts them as expected.

Starting point and indexed functions

The engine starts the pattern from the node it expects to produce the fewest records. A node bound by `rid:` is the strongest candidate, followed by nodes whose `type:` and `where:` are estimated to be small. Small sets (under 100 records) are loaded upfront. A node filtered by an indexed function that cannot estimate its result, such as `SEARCH_INDEX()` on a full-text index or a geospatial predicate, is still preferred over a scan of a large type, but its records are streamed, never loaded upfront, so a `LIMIT` stops the pattern early instead of waiting for the whole index result.

Context Variables

When running these queries, you can use any of the following context variables:

Variable	Description
`$matched`	Gives the current partial match: a record whose attributes are the aliases matched so far, so `$matched.<alias>` is the record bound to that alias and `$matched.<alias>.<property>` one of its properties. You can use this in the `where:` and `while:` conditions to refer to current partial matches or as part of the `RETURN` value. A `where:` that reads another alias through `$matched` is evaluated once that alias is bound, whether the two nodes are connected by a path or belong to different comma-separated patterns: in the latter case the pattern reading the alias runs once per match of the pattern binding it, which correlates the two patterns by property.
`$currentMatch`	Gives the current complete node during the match.
`$depth`	Gives the traversal depth, following a single path item where a `while:` condition is defined.

Use Cases

Expanding Attributes

You can run this statement as a sub-query inside of another statement. Doing this allows you to obtain details and aggregate data from the inner `SELECT` query.

```
ArcadeDB> SELECT person.name AS name, person.surname AS surname,
          friend.name AS friendName, friend.surname AS friendSurname
          FROM (MATCH {type: Person, as: person,
          where: (name = 'John')}.both('Friend'){as: friend}
          RETURN person, friend)

--------+----------+------------+---------------
 name   | surname  | friendName | friendSurname
--------+----------+------------+---------------
 John   | Doe      | John       | Smith
 John   | Doe      | Jenny      | Smith
 John   | Doe      | Frank      | Bean
 John   | Smith    | John       | Doe
 John   | Smith    | Jenny      | Smith
--------+----------+------------+---------------
```

As an alternative, you can use the following:

```
ArcadeDB> MATCH {type: Person, as: person,
          where: (name = 'John')}.both('Friend'){as: friend}
```

```
          RETURN
          person.name as name, person.surname as surname,
          friend.name as friendName, friend.surname as friendSurname

--------+----------+------------+---------------
 name   | surname  | friendName | friendSurname
--------+----------+------------+---------------
 John   | Doe      | John       | Smith
 John   | Doe      | Jenny      | Smith
 John   | Doe      | Frank      | Bean
 John   | Smith    | John       | Doe
 John   | Smith    | Jenny      | Smith
--------+----------+------------+---------------
```

Incomplete Hierarchy

Consider building a database for a company that shows a hierarchy of departments within the company. For instance,

```
          [manager] department
          (employees in department)

                [m0]0
                 (e1)
                 /   \
                /     \
               /       \
           [m1]1        [m2]2
          (e2, e3)     (e4, e5)
             / \          / \
            3   4        5   6
          (e6) (e7)    (e8)  (e9)
          /  \
      [m3]7    8
      (e10)   (e11)
       /
      9
  (e12, e13)
```

This loosely shows that, - Department `0` is the company itself, manager 0 (`m0`) is the CEO - `e10` works at department `7`, his manager is `m3` - `e12` works at department `9`, this department has no direct manager, so `e12's manager is `m3` (the upper manager)

In this case, you would use the following query to find out who's the manager to a particular employee:

```
ArcadeDB> MATCH {type:Employee, where: (name = ?)}.out('WorksAt').out
('ParentDepartment')
          {while: (out('Manager').size() == 0),
```

```
          where: (out('Manager').size() > 0)}.out('Manager')
          {as: manager} RETURN expand(manager)
```

Deep Traversal

Match path items act in a different manners, depending on whether or not you use `while:` conditions in the statement.

For instance, consider the following graph:

```
[name='a'] -FriendOf-> [name='b'] -FriendOf-> [name='c']
```

Running the following statement on this graph only returns `b`:

```
ArcadeDB> MATCH {type: Person, where: (name = 'a')}.out("FriendOf")
          {as: friend} RETURN friend

--------
 friend
--------
 b
--------
```

What this means is that it traverses the path item `out("FriendOf")` exactly once. It only returns the result of that traversal.

If you add a `while` condition:

```
ArcadeDB> MATCH {type: Person, where: (name = 'a')}.out("FriendOf")
          {as: friend, while: ($depth < 2)} RETURN friend

---------
 friend
---------
 a
 b
---------
```

Including a `while:` condition on the match path item causes ArcadeDB to evaluate this item as zero to *n* times. That means that it returns the starting node, (`a`, in this case), as the result of zero traversal.

To exclude the starting point, you need to add a `where:` condition, such as:

```
ArcadeDB> MATCH {type: Person, where: (name = 'a')}.out("FriendOf")
          {as: friend, while: ($depth < 2), where: ($depth > 0)}
```

```
          RETURN friend
```

As a general rule,

- `while` **Conditions:** Define this if it must execute the next traversal, (it evaluates at level zero, on the origin node).
- `where` **Condition:** Define this if the current element, (the origin node at the zero iteration the right node on the iteration is greater than zero), must be returned as a result of the traversal.

A `while:` or `maxDepth:` item returns one result per **path**, not per node: a node reached through several routes is returned once for each route, so the number of results can grow exponentially with the depth. The paths are produced one at a time, depth first (a node, then the paths through each of its neighbors), so a `LIMIT` stops the traversal as soon as it has its rows, and the command timeout (`arcadedb.command.timeout`) is checked at every node reached. A traversal around a cycle with no bound, such as `while: (true)`, never ends by itself: bound it with `$depth` or `maxDepth:`. The nodes the traversal holds along its current path, with the neighbors still to visit, count against `arcadedb.queryMaxHeapElementsAllowedPerOp` and `arcadedb.queryMaxHeapRAM`. To know only which nodes are reachable, `TRAVERSE` visits each node once. *Since v26.12.1*: earlier versions built every path before returning the first row, so `LIMIT`, the timeout and the heap limits did not bound the traversal (#11501).

For instance, suppose that you have a genealogical tree. In the tree, you want to show a person, grandparent and the grandparent of that grandparent, and so on. The result: saying that the person is at level zero, parents at level one, grandparents at level two, etc., you would see all ancestors on even levels. That is, `level % 2 == 0`.

To get this, you might use the following query:

```
ArcadeDB> MATCH {type: Person, where: (name = 'a')}.out("Parent")
          {as: ancestor, while: (true), where: ($depth % 2 = 0)}
          RETURN ancestor
```

Best practices

Queries can involve multiple operations, based on the domain model and use case. In some cases, like projection and aggregation, you can easily manage them with a `SELECT` query. With others, such as pattern matching and deep traversal, `MATCH` statements are more appropriate.

Use `SELECT` and `MATCH` statements together (that is, through sub-queries), to give each statement the correct responsibilities. Here,

Filtering Record Attributes for a Single Type

Filtering based on record attributes for a single type is a trivial operation through both statements. That is, finding all people named John can be written as:

```
ArcadeDB> SELECT FROM Person WHERE name = 'John'
```

You can also write it as,

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John')}
          RETURN person
```

The efficiency remains the same. Both queries use an index. With `SELECT`, you obtain expanded records, while with `MATCH`, you only obtain the Record ID's. If you want to return expanded records from the MATCH, use the `expand()` function in the return statement:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John')}
          RETURN expand(person)
```

Filtering on Record Attributes of Connected Elements

Filtering based on the record attributes of connected elements, such as neighboring vertices, can grow trick when using `SELECT`, while with `MATCH` it is simple.

For instance, find all people living in Rome that have a friend called John. There are three different ways you can write this, using `SELECT`:

```
ArcadeDB> SELECT FROM Person WHERE BOTH('Friend').name CONTAINS 'John'
          AND out('LivesIn').name CONTAINS 'Rome'

ArcadeDB> SELECT FROM (SELECT BOTH('Friend') FROM Person WHERE name =
          'John') WHERE out('LivesIn').name CONTAINS 'Rome'

ArcadeDB> SELECT FROM (SELECT in('LivesIn') FROM City WHERE name = 'Rome')
          WHERE BOTH('Friend').name CONTAINS 'John'
```

In the first version, the query is more readable, but it does not use indexes, so it is less optimal in terms of execution time. The second and third use indexes if they exist, (on `Person.name` or `City.name`, both in the sub-query), but they're harder to read. Which index they use depends only on the way you write the query. That is, if you only have an index on `City.name` and not `Person.name`, the second version doesn't use an index.

Using a `MATCH` statement, the query becomes:

```
ArcadeDB> MATCH {type: Person, where: (name = 'John')}.both("Friend")
          {as: result}.out('LivesIn'){type: City, where: (name = 'Rome')}
          RETURN result
```

Here, the query executor optimizes the query for you, choosing indexes where they exist. Moreover, the query becomes more readable, especially in complex cases, such as multiple nested `SELECT` queries.

`TRAVERSE` Alternative

There are similar limitations to using TRAVERSE. You may benefit from using MATCH as an alternative.

For instance, consider a simple TRAVERSE statement, like:

```
ArcadeDB> TRAVERSE out('Friend') FROM (SELECT FROM Person WHERE name = 'John')
          WHILE $depth < 3
```

Using a MATCH statement, you can write the same query as:

```
ArcadeDB> MATCH {type: Person, where: (name = 'John')}.out("Friend")
          {as: friend, while: ($depth < 3)} RETURN friend
```

Consider a case where you have a since date property on the edge Friend. You want to traverse the relationship only for edges where the since value is greater than a given date. In a TRAVERSE statement, you might write the query as:

```
ArcadeDB> TRAVERSE bothE('Friend')[since > date('2012-07-02', 'yyyy-MM-dd')].bothV()
          FROM (SELECT FROM Person WHERE name = 'John') WHILE $depth < 3
```

Unfortunately, this statement DOESN'T WORK in the current release. However, you can get the results you want using a MATCH statement:

```
ArcadeDB> MATCH {type: Person, where: (name = 'John')}.(bothE("Friend")
          {where: (since > date('2012-07-02', 'yyyy-MM-dd'))}.bothV())
          {as: friend, while: ($depth < 3)} RETURN friend
```

Projections and Grouping Operations

Projections and grouping operations are better expressed with a SELECT query. If you need to filter and do projection or aggregation in the same query, you can use SELECT and MATCH in the same statement.

This is particular important when you expect a result that contains attributes from different connected records (cartesian product). For instance, to retrieve names, their friends and the date since they became friends:

```
ArcadeDB> SELECT person.name AS name, friendship.since AS since, friend.name
          AS friend FROM (MATCH {type: Person, as: person}.bothE('Friend')
          {as: friendship}.bothV(){as: friend,
          where: ($matched.person != $currentMatch)}
          RETURN person, friendship, friend)
```

The same can be also achieved with the MATCH only:

```
ArcadeDB> MATCH {type: Person, as: person}.bothE('Friend')
          {as: friendship}.bothV(){as: friend,
          where: ($matched.person != $currentMatch)}
          RETURN person.name as name, friendship.since as since, friend.name as friend
```

RETURN expressions

In the RETURN section you can use:

multiple expressions, with or without an alias (if no alias is defined, ArcadeDB will generate a default alias for you), comma separated

```
MATCH
  {type: Person, as: person}
  .bothE('Friend'){as: friendship}
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN person, friendship, friend

result:

| person | friendship | friend |
-------------------------------
| #12:0  | #13:0      | #12:2  |
| #12:0  | #13:1      | #12:3  |
| #12:1  | #13:2      | #12:3  |
```

```
MATCH
  {type: Person, as: person}
  .bothE('Friend'){as: friendship}
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN person.name as name, friendship.since as since, friend.name as friend

result:

| name | since | friend |
-------------------------
| John | 2015  | Frank  |
| John | 2015  | Jenny  |
| Joe  | 2016  | Jenny  |
```

```
MATCH
  {type: Person, as: person}
  .bothE('Friend'){as: friendship}
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN person.name + " is a friend of " + friend.name as friends

result:
```

```
| friends                    |
------------------------------
| John is a friend of Frank  |
| John is a friend of Jenny  |
| Joe is a friend of Jenny   |
```

$matches, to return all the patterns that match current statement. Each row in the result set will be a single pattern, containing only nodes in the statement that have an `as:` defined

```
MATCH
  {type: Person, as: person}
  .bothE('Friend'){}  -- no 'as:friendship' in this case
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN $matches

result:

| person |  friend |
--------------------
| #12:0  |  #12:2  |
| #12:0  |  #12:3  |
| #12:1  |  #12:3  |
```

$paths, to return all the patterns that match current statement. Each row in the result set will be a single pattern, containing all th nodes in the statement. For nodes that have an `as:`, the alias will be returned, for the others a default alias is generated (automatically generated aliases start with `$ORIENT_DEFAULT_ALIAS_`)

```
MATCH
  {type: Person, as: person}
  .bothE('Friend'){}  -- no 'as:friendship' in this case
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN $paths

result:

| person | friend | $ORIENT_DEFAULT_ALIAS_0 |
-------------------------------------------
| #12:0  | #12:2  | #13:0                   |
| #12:0  | #12:3  | #13:1                   |
| #12:1  | #12:3  | #13:2                   |
```

$elements the same as `$matches`, but for each node present in the pattern, a single row is created in the result set (no duplicates)

```
MATCH
```

```
  {type: Person, as: person}
  .bothE('Friend'){}  -- no 'as:friendship' in this case
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN $elements

result:

| @rid   |  @type | name   |  .....   |
---------------------------------------
| #12:0  |  Person | John   |  .....   |
| #12:1  |  Person | Joe    |  .....   |
| #12:2  |  Person | Frank  |  .....   |
| #12:3  |  Person | Jenny  |  .....   |
```

$pathElements the same as $paths, but for each node present in the pattern, a single row is created in the result set (no duplicates)

```
MATCH
  {type: Person, as: person}
  .bothE('Friend'){}  -- no 'as:friendship' in this case
  .bothV(){as: friend, where: ($matched.person != $currentMatch)}
RETURN $pathElements

result:

| @rid   |  @type | name   | since  |  .....   |
-----------------------------------------------
| #12:0  |  Person | John   |        |  .....   |
| #12:1  |  Person | Joe    |        |  .....   |
| #12:2  |  Person | Frank  |        |  .....   |
| #12:3  |  Person | Jenny  |        |  .....   |
| #13:0  |  Friend |        |  2015  |  .....   |
| #13:1  |  Friend |        |  2015  |  .....   |
| #13:2  |  Friend |        |  2016  |  .....   |
```

NOTE: When using the MATCH statement in ArcadeDB Studio Graph panel you have to use $elements or $pathElements as return type, to let the Graph panel render the matched patterns correctly.

14.7. SQL - TRAVERSE

Retrieves connected records crossing relationships. This works with both the Document and Graph API's, meaning that you can traverse relationships between say invoices and customers on a graph, without the need to model the domain using the Graph API.

In many cases, you may find it more efficient to use SELECT, which can result in shorter and faster queries. For more information, see TRAVERSE versus SELECT below.

Syntax

```
TRAVERSE <projection> [, <projection>]*   (a field, or a function such as out(), in()
or both())
         FROM <target>
         [MAXDEPTH <number>]
         [WHILE <condition>]
         [LIMIT <max-records>]
         [STRATEGY DEPTH_FIRST|BREADTH_FIRST]
```

- `<fields>` Defines the fields you want to traverse.
- `<target>` Defines the target you want to traverse. This can be a type, one or more buckets, a single Record ID, set of Record ID's, a sub-query, or a context variable such as a script `LET` variable (`TRAVERSE out() FROM $x`) or the parent record in a `LET` sub-query (`TRAVERSE out() FROM $parent.$current`). A variable can hold a record, a Record ID, a result set or a list of them; a variable that is not set traverses nothing.
- `MAXDEPTH` Defines the maximum depth of the traversal. `0` indicates that you only want to traverse the root node. Negative values are invalid. The result is every record reachable within that many hops of a root, whichever strategy is used.
- `WHILE` Defines the condition for continuing the traversal while it is true. It is evaluated on every record before it is traversed, the roots included: a root that does not satisfy it is neither returned nor traversed. `MAXDEPTH` and `WHILE` can be used together.
- `LIMIT` Defines the maximum number of results the command can return.
- `STRATEGY` Defines strategy for traversing the graph. It also decides what `$depth` and `$path` mean on a graph where a record can be reached by more than one path, see Traversal Strategies.

`TRAVERSE` has no `WHERE` clause. To filter the traversed records, wrap the `TRAVERSE` in a `SELECT`, as in the examples below.

There is a difference between `MAXDEPTH N` and `WHILE $depth <= N`: the `MAXDEPTH` will evaluate exactly N levels, while the `WHILE` will evaluate N+1 levels and will discard the N+1th, so the `MAXDEPTH` in general has better performance. Both return the same records.

Examples

In a social network-like domain, a user profile is connected to friend through links. The following examples consider common operations on a user with the record ID `#10:1234`.

- Traverse the outgoing edges of the root record (`*` is not accepted after `TRAVERSE`; use `out()`, `in()` or `both()`):

```
ArcadeDB> TRAVERSE out() FROM #10:1234
```

- Specify fields and depth up to the third level, using the `BREADTH_FIRST` strategy:

```
ArcadeDB> TRAVERSE out("Friend") FROM #10:1234 MAXDEPTH 3
            STRATEGY BREADTH_FIRST
```

- Execute the same command, this time filtering for a minimum depth to exclude the first target vertex:

```
ArcadeDB> SELECT FROM (TRAVERSE out("Friend") FROM #10:1234 MAXDEPTH 3)
            WHERE $depth >= 1
```

Bound the depth with `MAXDEPTH` or `WHILE` in the inner `TRAVERSE` rather than with `$depth` in the outer `SELECT`. It is much more efficient, because the traversal stops there instead of walking the whole graph. It is also correct with both strategies, while an outer `WHERE $depth <= N` returns every record within N hops only with `STRATEGY BREADTH_FIRST`: with the default `DEPTH_FIRST`, a record first reached through a longer path carries that longer `$depth` and is filtered out, see Traversal Strategies. Filtering out only the root, as above, is safe with both: the roots are the only records at depth `0`.

- Combine traversal with `SELECT` command to filter the result-set. Repeat the above example, filtering for users in Rome:

```
ArcadeDB> SELECT FROM (TRAVERSE out("Friend") FROM #10:1234 MAXDEPTH 3)
            WHERE city = 'Rome'
```

- Extract movies of actors that have worked, at least once, in any movie produced by J.J. Abrams:

```
ArcadeDB> SELECT FROM (TRAVERSE out("Actors"), out("Movies") FROM (SELECT FROM
            Movie WHERE producer = "J.J. Abrams") MAXDEPTH 3) WHERE
            @type = 'Movie'
```

- Display the current path in the traversal:

```
ArcadeDB> SELECT $path FROM ( TRAVERSE out() FROM V MAXDEPTH 10 )
```

Supported Variables

Fields

Defines the fields that you want to traverse. If set to `*`, then it traverses all fields. This can prove costly to performance and resource usage, so it is recommended that you optimize the command to only traverse the pertinent fields.

In addition to his, you can specify the fields at a type-level. Inheritance is supported. By specifying `Person.city` and the type `Customer` extends person, you also traverse fields in `Customer`.

Field names are case-sensitive, types not.

Target

Targets for traversal can be:

- `<type>` Defines the type that you want to traverse.
- `BUCKET:<bucket>` Defines the bucket you want to traverse. The bucket name can also be an input parameter, `BUCKET::name` or `BUCKET:?` *(since v26.12.1, issue #12480: earlier versions failed with "Bucket with name 'null' was not found").*
- `<record-id>` Individual root Record ID that you want to traverse.
- `[<record-id>,<record-id>,…]` Set of Record ID's that you want to traverse. This is useful when navigating graphs starting from the same root nodes.

Context Variables

In addition to the above, you can use the following context variables in traversals:

- `$parent` Gives the parent context, if any. You may find this useful when traversing from a sub-query.
- `$current` Gives the current record in the iteration. To get the upper-level record in nested queries, you can use `$parent.$current`.
- `$depth` Gives the number of hops from the root to the current record, along `$path`. The root is at depth `0`.
- `$path` Gives the path the traversal followed to reach the current record, as the list of its Record IDs from the root to the record itself (for instance `[#8:0, #8:1, #8:3]`). It always has `$depth + 1` elements. You can also display it through `SELECT`:

```
ArcadeDB> SELECT $path, $depth FROM (TRAVERSE out() FROM #10:1234)
```

- `$stack` Gives the same Record IDs as `$path`, in reverse order: the current record first, the root last.

With `STRATEGY BREADTH_FIRST`, `$depth` is the shortest distance from the root. With the default `DEPTH_FIRST` it can be longer, see Traversal Strategies.

Use Cases

`TRAVERSE` versus `SELECT`

When you already know traversal information, such as relationship names and depth-level, consider using `SELECT` instead of `TRAVERSE` as it is faster in some cases.

For example, this query traverses the `follow` relationship from the Twitter account `#10:1234`, getting the accounts at the second level of friendship, that is the accounts it does not follow directly but that the accounts it follows do:

```
ArcadeDB> SELECT FROM (TRAVERSE out('follow') FROM #10:1234 MAXDEPTH 2
          STRATEGY BREADTH_FIRST) WHERE $depth = 2
```

`STRATEGY BREADTH_FIRST` is what makes `$depth = 2` mean "two hops away and not closer", see Traversal Strategies. Note that the traversal starts from one account: each record is returned once per `TRAVERSE`, so starting from every account (`FROM TwitterAccounts`) makes them all roots, an account is typically returned at depth `0`, and `$depth = 2` finds next to nothing.

If you only need the accounts reachable in exactly two hops, you can express it with a `SELECT`, in a way that is also shorter and faster:

```
ArcadeDB> SELECT expand(out('follow').out('follow')) FROM #10:1234
```

The two are not the same query: the `SELECT` also returns an account the root follows directly when it is followed by another account the root follows, and returns an account once for every path that reaches it.

`TRAVERSE` with the Graph Model and API

While you can use the `TRAVERSE` command with any domain model, it provides the greatest utility with the Graph Model.

This model is based on the concepts of the Vertex (or Node) and the Edge (or Arc, Connection, Link, etc.) If you want to traverse in a direction, you have to use the type name when declaring the traversing fields. The supported directions are:

- **Vertex to outgoing edges** Using `outE()` or `outE('EdgeTypeName')`. That is, going out from a vertex and into the outgoing edges.
- **Vertex to incoming edges** Using `inE()` or `inE('EdgeTypeName')`. That is, going from a vertex and into the incoming edges.
- **Vertex to all edges** Using `bothE()` or `bothE('EdgeTypeName')`. That is, going from a vertex and into all the connected edges.
- **Edge to Vertex (end point)** Using `inV()` . That is, going out from an edge and into a vertex.
- **Edge to Vertex (starting point)** Using `outV()` . That is, going back from an edge and into a vertex.
- **Edge to Vertex (both sizes)** Using `bothV()` . That is, going from an edge and into connected vertices.
- **Vertex to Vertex (outgoing edges)** Using `out()` or `out('EdgeTypeName')`. This is the same as `outE().inV()`
- **Vertex to Vertex (incoming edges)** Using `in()` or `in('EdgeTypeName')`. This is the same as `inE().outV()`
- **Vertex to Vertex (all directions)** Using `both()` or `both('EdgeTypeName')`.

For instance, traversing outgoing edges on the record `#10:3434`:

```
ArcadeDB> TRAVERSE out() FROM #10:3434
```

In a domain for emails, to find all messages sent on January 1, 2012 from the user Luca, assuming that they are stored in the vertex type `User` and that the messages are contained in the vertex type `Message`. Sent messages are stored as `out` connections on the edge type `SentMessage`:

```
ArcadeDB> SELECT FROM (TRAVERSE outE(), inV() FROM (SELECT FROM User WHERE
          name = 'Luca') WHILE $depth &lt;= 2 AND (@type = 'User' OR
          @type = 'Message' OR (@type = 'SentMessage' AND
          sentOn = '01/01/2012'))) WHERE @type = 'Message'
```

The `WHILE` condition accepts `User` too because it is also evaluated on the root: without it, the traversal would stop before leaving the user.

14.7.1. Traversal Strategies

When ArcadeDB traverses a graph it can use one of two available approaches, either explore every branch to its leaf before backtracking (depth-first), or visiting each child before progressing down the next level (breadth-first), see:

- `DEPTH_FIRST` Strategy
- `BREADTH_FIRST` Strategy

By default, the depth-first strategy is used.

A `TRAVERSE` returns each record at most once, even when several roots or several paths reach it. Without a `LIMIT`, both strategies return the same records, and `MAXDEPTH N` (or `WHILE $depth <= N`) returns every record within N hops with either of them. What differs is the order of the results and, on a graph where a record can be reached by paths of different lengths, the `$depth` and `$path` a record is returned with:

Strategy	Order of the results	`$depth` and `$path` of a record
`DEPTH_FIRST` (default)	A branch is explored to its end before the next one	The path the walk took. It can be longer than the shortest one
`BREADTH_FIRST`	All the records at depth N before any record at depth N+1	A shortest path. `$depth` is the distance from the root

For example, if `A` reaches `X` through `A → B → X` and also through `A → C → D → E → X`, a depth-first traversal that explores the `C` branch first returns `X` with `$depth` 4 and the longer path, while a breadth-first one always returns it with `$depth` 2. On a tree, where every record has only one path from the root, the two strategies return the same `$depth` and `$path`.

When the meaning of `$depth` matters, as when you filter or group the results by it in an outer `SELECT` (`WHERE $depth = 2`, "the friends of my friends"), use `STRATEGY BREADTH_FIRST`. With the default `DEPTH_FIRST`, `SELECT FROM (TRAVERSE …) WHERE $depth <= 2` can miss records that are within 2 hops, because they were first

reached through a longer path. To only bound the traversal, `MAXDEPTH` and `WHILE` are correct with both strategies.

14.8. SQL - `MOVE VERTEX`

Moves one or more vertices into another type or bucket.

```
MOVE VERTEX <source> TO <TYPE|BUCKET>:<destination> [SET|REMOVE <field-name> = <field-value>[,]*]|[CONTENT|MERGE <JSON>] [BATCH <number>]
```

- `<source>` the vertex (RID) or vertices (Array of RIDs) to be moved.
- `<TYPE|BUCKET>` determines if a type or bucket is the target.
- `<destination>` the target type or bucket of the moved vertex or vertices.
- `<number>` size of vertices moved at once. Using `-1` moves all vertices at once.
- `CONTENT` replaces the whole content of each moved vertex. It takes a JSON object, a JSON array (one item per moved vertex, in order) or a parameter holding either, as `UPDATE ⋯ CONTENT` does. `MERGE` takes a JSON object or a parameter.

This is command is similar to the UPDATE command.

A vertex moved to a bucket takes the type that owns the bucket, as `INSERT INTO BUCKET:` does. Moving a vertex to a bucket of another type therefore also moves it to that type: it leaves the indexes of its old type and enters those of the new one.

Examples

```
MOVE VERTEX #4:1 TO Type:MyType
```

```
MOVE VERTEX [#4:16,#4:20,#4:24] TO BUCKET:MyBucket
```

```
MOVE VERTEX (SELECT FROM User) TO TYPE:NewUsers BATCH 4
```

```
MOVE VERTEX (SELECT FROM User WHERE name = 'Jay') TO TYPE:NewUsers CONTENT :payload
```

14.9. SQL - `CREATE VERTEX`

Creates a new vertex in the database.

The Vertex and Edge are the main components of a Graph database. ArcadeDB supports polymorphism on vertices.

Syntax

```
CREATE VERTEX [<type>] [BUCKET <bucket>]
  [SET <field> = <expression>[,]*]|
  [CONTENT {<JSON>}|[{<JSON>}[,]*]]
```

- `<type>` Defines the type to which the vertex belongs.
- `<bucket>` Defines the bucket in which it stores the vertex.
- `<field>` Defines the field you want to set.
- `<expression>` Defines the express to set for the field.

When using a distributed database, you can create vertices through two steps (creation and update). Doing so can break constraints defined at the type-level for vertices. To avoid these issues, disable constraints in the vertex type.

Examples

- Create a new vertex type, then create a vertex in that type:

```
ArcadeDB> CREATE VERTEX TYPE V1
ArcadeDB> CREATE VERTEX V1
```

- Create a new vertex within a particular bucket:

```
ArcadeDB> CREATE VERTEX V1 BUCKET recent
```

- Create a new vertex, defining its properties:

```
ArcadeDB> CREATE VERTEX SET brand = 'fiat'
```

- Create a new vertex of the type `V1`, defining its properties:

```
ArcadeDB> CREATE VERTEX V1 SET brand = 'fiat', name = 'wow'
```

- Create a vertex using JSON content:

```
ArcadeDB> CREATE VERTEX Employee CONTENT { "name" : "Jay", "surname" : "Miner" }
```

- Create multiple vertices using JSON content:

```
ArcadeDB> CREATE VERTEX Employee CONTENT [{"name" : "Jay", "surname" : "Miner"},
```

```
            {"name": "Frank", "surname": "Hermier"},{"name": "Emily", "surname":
"Sout"}]
```

Idempotent vertex creation (upsert)

`CREATE VERTEX` does not have an `UPSERT` clause. To create a vertex **only if it does not already exist**, or update it otherwise, use the `UPDATE ⋯ UPSERT` command against the vertex type. When the target type is a vertex type, `UPSERT` creates a real vertex (not a plain document), so it can be used as the endpoint of an edge right away.

The lookup must be backed by a `UNIQUE` index on the property used in the `WHERE` clause; this is what makes the operation atomic under concurrent writers.

```
ArcadeDB> CREATE VERTEX TYPE Person
ArcadeDB> CREATE PROPERTY Person.email STRING
ArcadeDB> CREATE INDEX ON Person (email) UNIQUE

ArcadeDB> UPDATE Person SET name = 'Luca', email = 'luca@example.com'
            UPSERT WHERE email = 'luca@example.com'
```

Re-running the same `UPDATE ⋯ UPSERT` updates the existing vertex instead of creating a duplicate. See `UPDATE` for the full syntax and the upsert limitations.

For more information, see:

- `CREATE EDGE`
- `UPDATE` (`UPSERT` clause)

14.10. SQL - `CREATE EDGE`

Creates a new edge between vertices in the database. An edge can connect from and to a vertex record, an array of vertex records, or a query result of vertex records.

Syntax

```
CREATE EDGE <type> [BUCKET <bucket>] FROM <rid>|[<rid>[,]*]|(<query>) TO <rid>|[<rid
>[,]*]|(<query>)
                   [UNIDIRECTIONAL]
                   [IF NOT EXISTS]
                   [SET <field> = <expression>[,]*]|CONTENT {<JSON>}
                   [RETRY <retry> [WAIT <pauseBetweenRetriesInMs]]
                   [BATCH <batch-size>]
```

- `<type>` defines the type name for the edge.
- `<bucket>` defines the bucket in which you want to store the edge.
- `UNIDIRECTIONAL` creates a unidirectional edge; by default edges are bidirectional.

- `IF NOT EXISTS` skips the creation of the edge if another edge already exists with the same direction (same from/to) and same edge type. *(Since v26.12.1)* The check holds under concurrent writers too: of two transactions creating the same edge at the same time, the second to commit is refused with a retryable `ConcurrentModificationException`, and its retry finds the edge the first one created (see concurrent `MERGE`). In v26.11.1 and earlier, concurrent writers could store the edge more than once.
- `JSON` provides JSON content to set as the record. Use this instead of entering data field by field.

  An `@in` / `@out` set through `SET` or `CONTENT` re-points the new edge, as `UPDATE … SET @in` does to an existing one: `CREATE EDGE F FROM #1:0 TO #1:1 SET @in = #1:2` creates an edge from `#1:0` to `#1:2`. A named record that does not exist, was deleted in the same transaction, or is not a vertex fails the statement, and no edge is stored. The same applies to an openCypher `CREATE` with an `@in` / `@out` property and to the Java `newEdge()`. *(Since v26.12.1: v26.11.1 stored an edge whose `@in` / `@out` disagreed with the vertex it was linked to, which `CHECK DATABASE FIX` then deleted, and v26.10.1 ignored the value.)*
- `RETRY` defines the number of retries to attempt in the event of conflict (optimistic approach).
- `WAIT` defines the time to delay between retries in milliseconds.
- `BATCH` defines whether it breaks the command down into smaller blocks and the size of the batches. This helps to avoid memory issues when the number of vertices is too high. By default, it is set to `100`.

Edges and Vertices form the main components of a Graph database. ArcadeDB supports polymorphism on edges.

When no edges fail to create, ArcadeDB throws a `CommandExecutionException` error. This makes it easier to integrate edge creation in transactions. In such cases, if the source or target vertices don't exist, it rolls back the transaction. An endpoint that is a document or an edge rather than a vertex, named by its RID or returned by a query, fails the statement with `Invalid vertex for edge creation: the document #5:0 is not a vertex` (a `TypeMismatchException`, answered `400` over HTTP), and no edge is stored. *(Since v26.12.1: earlier versions answered `500`, with a raw `ClassCastException` for a RID.)*

Examples

- Create a new edge type and an edge of the new type:

```
ArcadeDB> CREATE EDGE TYPE E1
ArcadeDB> CREATE EDGE E1 FROM #10:3 TO #11:4
```

- Create an edge in a specific bucket:

```
ArcadeDB> CREATE EDGE E1 BUCKET EuropeEdges FROM #10:3 TO #11:4
```

- Create an edge and define its properties:

```
ArcadeDB> CREATE EDGE FROM #10:3 TO #11:4 SET brand = 'fiat'
```

- Create an edge of the type `E1` and define its properties:

```
ArcadeDB> CREATE EDGE E1 FROM #10:3 TO #11:4 SET brand = 'fiat', name = 'wow'
```

- Create multiple edges from a single source vertex to many destination records:

```
ArcadeDB> CREATE EDGE E FROM #1:0 TO [#4:0, #7:0];
```

- Create no edges from a single source vertex (this is useful when using subqueries for the destination):

```
ArcadeDB> CREATE EDGE E FROM #1:0 TO [];
```

- Create edges of the type `Watched` between all action movies in the database and the user Luca, using sub-queries:

```
ArcadeDB> CREATE EDGE Watched FROM (SELECT FROM account WHERE name = 'Luca') TO
            (SELECT FROM movies WHERE type.name = 'action')
```

- Create an edge using JSON content:

```
ArcadeDB> CREATE EDGE E FROM #22:33 TO #22:55 CONTENT { "name": "Jay",
            "surname": "Miner" }
```

- Create an edge only if not previously created:

```
ArcadeDB> CREATE INDEX Watched_out_in ON Watched (`@out`, `@in`) UNIQUE
ArcadeDB> CREATE EDGE Watched FROM (SELECT FROM account WHERE name = 'Luca') TO
            (SELECT FROM movies WHERE type.name = 'action') IF NOT EXISTS
```

Why there is no `UPSERT` for edges

The `UPDATE … UPSERT` clause is supported on document and vertex types, but not on edge types: an edge always needs `FROM` and `TO` endpoints, which an `UPDATE WHERE` clause cannot supply. The idempotent equivalent for edges is `CREATE EDGE … IF NOT EXISTS`, ideally combined with a `UNIQUE` index on `(@out, @in)` so the uniqueness check is atomic under concurrent writers (see the example above).

A common pattern is to upsert the two vertices first, then connect them:

```
ArcadeDB> UPDATE Person SET name = 'Luca', email = 'luca@example.com'
            UPSERT WHERE email = 'luca@example.com'
ArcadeDB> UPDATE City   SET name = 'London'
            UPSERT WHERE name = 'London'
ArcadeDB> CREATE EDGE Lives
            FROM (SELECT FROM Person WHERE email = 'luca@example.com')
            TO   (SELECT FROM City   WHERE name  = 'London')
            IF NOT EXISTS
```

For an end-to-end multi-statement version (with retry on conflict), see SQL Script.

For more information, see:

- `CREATE VERTEX`
- `UPDATE` (`UPSERT` clause)

14.11. Types

14.11.1. SQL - `CREATE TYPE`

Creates a new type in the schema.

Syntax

```
CREATE <DOCUMENT|VERTEX|EDGE> TYPE <type>
[UNIDIRECTIONAL] [LIGHTWEIGHT] [UNIQUE] [ IF NOT EXISTS ]
[EXTENDS <super-type>] [BUCKET <bucket-id>[,]*] [BUCKETS <total-bucket-number>]
[PAGESIZE <page-size>]
[CUSTOM <custom-key> = <custom-value>[,]*]
```

- Use `<DOCUMENT|VERTEX|EDGE>` if you are creating respectively a document, vertex or edge type.
- `<type>` Defines the name of the type you want to create. You must use a letter, underscore or dollar for the first character, for all other characters you can use alphanumeric characters, underscores and dollar.
- **UNIDIRECTIONAL** Defines an edge types (only) to be of single direction instead of the default bi-directional edge. An edge of such a type is written on its source vertex only, which halves the write cost of the edge list, but the target vertex keeps no trace of it: walking the vertex API from the target (`vertex.getVertices(IN, …)`, Gremlin's `in()`) finds nothing, and so do the SQL functions `in()`, `inE()`, `both()`, `bothE()` and `shortestPath()` called on their own (`SELECT in('E') FROM …`), which read what the vertex stores, as the vertex API does embedded, remote and through Gremlin. A pattern is answered whichever way it is written: Cypher and SQL `MATCH` start the walk from the source side whenever the pattern allows it, and a walk that has to start from the target (the target is bound by an earlier clause, the hop is undirected, a pattern expression such as `COUNT { (t)<-[:E]-() }`, a Cypher `shortestPath()`) is answered by scanning the edges of the type once per query. That scan is held in heap and counts against `arcadedb.queryMaxHeapElementsAllowedPerOp` and `arcadedb.queryMaxHeapRAM`, so declare an edge

type unidirectional only when its edges are queried from their source: a workload that looks up the incoming side often (many small queries by target, or a loop that commits between a write and a read) pays that scan every time, and is better served by a bidirectional type.

Graph algorithms and path procedures ask about the graph rather than about what a vertex stores, so they always answer the incoming side of a unidirectional edge type, the same as with a Graph Analytical View: the Cypher `algo.*` procedures, the `path.*` procedures (`path.expand`, `path.expandConfig`, `path.spanningTree`, `path.subgraphNodes`, `path.subgraphAll`), the SQL functions `dijkstra()`, `astar()`, `bellmanFord()`, `cchShortestPath()` and `duanSSSP()`, and the functions `node.degree`, `node.degree.in`, `node.relationship.exists` and `node.relationship.types`. An `algo.*` procedure that loads the whole graph derives the incoming side from the outgoing lists of the vertices it loaded, with no scan. A walk that goes vertex by vertex (the `path.*` procedures, the SQL path-finding functions, the `node.*` functions) builds one in-heap index per unidirectional type the first time it needs it, the scan described above, bounded by `arcadedb.queryMaxHeapElementsAllowedPerOp`. `refactor.mergeNodes`, `refactor.cloneNodesWithRelationships` and adding or removing a node's labels in Cypher (which moves the vertex to another type) carry the incoming unidirectional edges of the node over too, instead of dropping them. The SQL functions `in()`, `inE()`, `both()`, `bothE()` and `shortestPath()` called on their own keep reading what the vertex stores, as before.

Since 26.10.2 the graph algorithms, path procedures and path-finding functions listed above answer both sides of a unidirectional edge, and `refactor.mergeNodes`, `refactor.cloneNodesWithRelationships` and a Cypher label change keep the incoming unidirectional edges of the node.

- **LIGHTWEIGHT** Defines an edge type (only) whose edges are stored inside the two vertices they connect, with no edge record and therefore no properties. Orthogonal to `UNIDIRECTIONAL`. See Lightweight Edges.
- **UNIQUE** Defines an edge type (only) that allows at most one edge per ordered `(out, in)` pair of vertices. On a regular edge type this creates a unique index on `(@out, @in)`; on a `LIGHTWEIGHT` type there are no records to index and the check scans the source vertex's edge list instead. See Lightweight Edges.
- **IF NOT EXISTS** Specifying this option, the type creation will just be ignored if the type already exists (instead of failing with an error)
- **`<super-type>`** Defines the super-type you want to extend with this type. It must be a document type or a type of the same kind as the new one: `CREATE VERTEX TYPE … EXTENDS` takes a vertex or a document type, `CREATE EDGE TYPE … EXTENDS` an edge or a document type, and `CREATE DOCUMENT TYPE … EXTENDS` only a document type. Any other super-type refuses the statement before the type is created. *(Since v26.12.1, issue #12029. Before, for example, a document or an edge type could extend a vertex type, and openCypher then failed with `ClassCastException` on the first such record a label scan met.)*
- **`<bucket-id>`** Defines in a comma-separated list the ID's of the buckets you want this type to use. A bucket given by its ID must exist already, and an unknown ID refuses the statement; a bucket given by its name is created when it does not exist. *(Since v26.12.1, issue #12436. Before, a bucket given by its ID failed the statement with a `NullPointerException`.)*
- **`<total-bucket-number>`** Defines the total number of buckets you want to create for this type.

When omitted, the default is taken from the `arcadedb.typeDefaultBuckets` global configuration setting (`TYPE_DEFAULT_BUCKETS`), which is `1` by default.

- `<page-size>` Defines the page size of the type.
- `<custom-key>` Name of a custom property to set on the new type. Multiple pairs can be listed, separated by commas.
- `<custom-value>` Value for the custom property. All data types are supported as values.

In the event that a bucket of the same name exists in the bucket, the new type uses this bucket by default. If you do not define a bucket in the command and a bucket of this name does not exist, ArcadeDB creates one. The new bucket has the same name as the type, but in lower-case.

A bucket belongs to one type only. The statement is refused when a bucket it would attach already belongs to another type, whether listed in `BUCKET` or reused because its name is `<type>_<n>`, and when the same bucket is listed twice. The refusal is checked before anything is created, so the schema is left as it was and no half-built type remains. *(Since v26.11.1. Before, the type was registered first and stayed in the schema, without the refused bucket, and also after a restart.)*

When working with multiple cores, it is recommended that you use multiple buckets to improve concurrency during inserts. To change the number of buckets created by default, `ALTER DATABASE` command to update the `minimumbuckets` property. You can also define the number of buckets you want to create using the `BUCKETS` option when you create the type.

Examples

- Create the document type `Account`:

```
ArcadeDB> CREATE DOCUMENT TYPE Account
```

- Create the vertex type `Car` to extend `Vehicle`:

```
ArcadeDB> CREATE VERTEX TYPE Car EXTENDS Vehicle
```

- Create the vertex type `Car`, using the bucket with name 'Car_classic' and 'Car_modern':

```
ArcadeDB> CREATE VERTEX TYPE Car BUCKET Car_classic,Car_modern
```

- Create the document type `Account` with the custom property 'description':

```
ArcadeDB> CREATE DOCUMENT TYPE Account CUSTOM description = 'All users'
```

- Create the vertex type `Car` with multiple custom properties:

```
ArcadeDB> CREATE VERTEX TYPE Car CUSTOM icon = 'car.png', owner = 'fleet-team'
```

Declaring `CUSTOM` inline is equivalent to running `ALTER TYPE … CUSTOM` right after the creation, but it happens in the same statement. When `IF NOT EXISTS` is used and the type is already there, the whole statement is skipped, so the custom properties are not applied to the existing type.

`IF NOT EXISTS` always returns exactly one row, whether the object was created or was already there. The `created` property tells the two apart, so a retry never looks like a failure to a client that checks the number of returned rows:

```
CREATE DOCUMENT TYPE Person IF NOT EXISTS
--> { "operation": "create document type", "typeName": "Person",
"created": true }

CREATE DOCUMENT TYPE Person IF NOT EXISTS
--> { "operation": "create document type", "typeName": "Person",
"created": false }
```

The same applies to `CREATE PROPERTY`, `CREATE BUCKET`, `CREATE INDEX`, `CREATE TIMESERIES TYPE`, `CREATE TRIGGER`, `CREATE GRAPH ANALYTICAL VIEW`, `CREATE MATERIALIZED VIEW` and `CREATE CONTINUOUS AGGREGATE`.

Before that version, `CREATE TYPE`, `CREATE PROPERTY`, `CREATE BUCKET` and `CREATE TIMESERIES TYPE` returned no rows at all when the object already existed.

A type inherits the bucket selection strategy defined at the database level (`round-robin` by default). For the full list of supported strategies, the decision tree for picking one, and the `WITH repartition = true` workflow for changing strategy on a populated type, see Bucket Selection Strategies in the Schema concepts section and Bucket Selection Strategy for the animated walkthrough.

For more information, see:

- `ALTER TYPE`
- `DROP TYPE`
- `CREATE BUCKET`
- Schema design 101: choosing a bucket strategy

14.11.2. SQL - `ALTER TYPE`

Change a type defined in the schema. The change is persistent.

Syntax

```
ALTER TYPE <type> <change> [, <change> ]*
```

Where each `<change>` is either `<attribute-name> <attribute-value>` or `CUSTOM <custom-key> = <custom-value>`.

- `<type>` Defines the type you want to change.
- `<attribute-name>` Defines the attribute you want to change. For a list of supported attributes, see the table below.
- `<attribute-value>` Defines the value you want to set.
- `<custom-key>` Name of the custom property to set.
- `<custom-value>` Value for the custom property. All data types are supported as values.

A custom property can be deleted by setting its value to `null`.

Several changes can be applied in one statement, separated by commas, and they are applied in the order written:

```
ALTER TYPE Employee NAME Staff, SUPERTYPE +Person
```

Before that version the comma-separated form parsed and answered `"result": "OK"`, but only the last change was actually applied.

Examples

- Add `Person` to the super types:

```
ArcadeDB> ALTER TYPE Employee SUPERTYPE +Person
```

- Remove a super-type:

```
ArcadeDB> ALTER TYPE Employee SUPERTYPE -Person
```

Removing a super-type also removes the indexes the type inherited from it. Those indexes were created when the super-type was assigned, they cover a property the type no longer has, and the super-type's `UNIQUE` constraints stop applying to the type's records. Assigning the super-type again re-creates and rebuilds them.

Assigning a super-type is refused, before anything changes, when it would close a cycle (the super-type is the type itself or already one of its sub types, at any depth), or when it is a type of another kind: a vertex type can extend a vertex or a document type, an edge type an edge or a document type, and a document type only a document type. The same applies to the Java API (`addSuperType()`) and to the JSONL import of a logical export, which links the exported super-types the same way. *(Since v26.12.1, issues #12028 and #12029. Before, a cycle failed with a `StackOverflowError` (HTTP `500`) but stayed in the schema, after which queries on the type failed and the database could not be reopened; a link to a type of another kind was accepted, and openCypher then failed with `ClassCastException`.)* A database whose schema already holds a cycle still opens: one link of the cycle is left out,

with a warning in the log naming it, and the next schema save writes the result. Which link is left out depends on the order the types are loaded in. A link between types of different kinds written by an earlier version is kept.

Assigning a super-type with a `UNIQUE` index is refused with a `DuplicatedKeyException` when a record of the type holds a key the index already holds (or two records of the type share one), and the type is left as it was. The same applies to `ALTER TYPE ⋯ BUCKET +` when the bucket added already holds records: a duplicate key refuses the statement and the bucket is not attached. *(Since v26.12.1, issue #11302. Before, the statement was accepted and the type's part of the index was left empty, so queries using that index missed the type's records.)*

Assigning a super-type builds its indexes (and those of the types above it) for the whole branch that joins it: the type and every type below it, at any depth, since their records become records of the super-type too. A `UNIQUE` key that clashes anywhere in that branch refuses the statement in the same way, with nothing changed. *(Since v26.12.1, issue #12036. Before, only the type's own records were indexed: a `UNIQUE` index was not enforced for the records of its sub types, and `SEARCH_INDEX` and `vectorNeighbors` did not return them. In a database where the super-type was assigned on an earlier version, removing it and assigning it again builds the missing entries.)*

Assigning a super-type is refused, before anything changes, when a stored record of the type, or of a type below it, violates a property the type would inherit from it (or from the types above it): a `MANDATORY` property the record lacks, a `NOTNULL` one it holds as `null`, a value of another type than the declared one, or a value outside its `MIN`, `MAX` or `REGEXP`. This is the check `CREATE PROPERTY` and `ALTER PROPERTY` apply to existing records. The message names the first violating record and the reason, for example `Cannot make type 'T' a sub type of 'S': record #1:0 violates the properties it would inherit, The property 'T.p' value 5 is less than 10. Fix or delete the non-conforming records first`. A property the type already declares itself with the same type is governed by that declaration and not checked again. The check reads the records of the type and its sub types, so on a large type the statement takes as long as that scan. *(Since v26.12.1, issue #12030. Before, the statement was accepted, and every later write to such a record failed, even one that did not touch the inherited property.)*

Removing a super-type is refused, and nothing changes, while an index of the type or of a type below it stands on a property that type would stop inheriting, the same rule `DROP PROPERTY` applies. The message names the property and the index: drop the index first, declare the property on the type, or add a super-type that provides it before removing the old one. The changes of one statement are applied in the order written, so to swap two super-types that both declare the property write the addition first (`ALTER TYPE C SUPERTYPE +Q, -P`); `SUPERTYPE -P, +Q` is refused at the removal. A removal is also refused when the property would then be inherited from another super-type with a different type (for example `STRING` instead of the `INTEGER` the index is keyed on). `DROP TYPE` applies the same rule to the sub types of

the dropped type (see `DROP TYPE`). *(Since v26.12.1, issue #12038. Before, the removal was accepted and left the index keyed on a property the type no longer declared, so values were stored with one type and indexed with another.)*

- Add the "account2" bucket to the type `Account`.

```
ArcadeDB> ALTER TYPE Account BUCKET +account2
```

In the event that the bucket does not exist, it automatically creates it.

- Remove the bucket `account_34` from the type `Account` (buckets are referenced by name):

```
ArcadeDB> ALTER TYPE Account BUCKET -account_34
```

The records stay in the bucket, but they no longer belong to any type. They can still be read and deleted, for example with `SELECT FROM BUCKET:account_34`, `DELETE FROM BUCKET:account_34` or `TRUNCATE BUCKET account_34`, and they are returned without `@type`. Updating one is refused with an error that names the bucket: attach the bucket to a type with `ALTER TYPE … BUCKET +` to update it again. *(Since v26.12.1, issue #11335. Before, reading such a record as JSON, updating it and deleting it each failed with a `NullPointerException`.)*

- Modify the bucket selection strategy to `partitioned` selecting the property id as partition key:

```
ArcadeDB> ALTER TYPE Account BucketSelectionStrategy `partitioned('id')`
```

- Set the custom value with key 'description':

```
ArcadeDB> ALTER TYPE Account CUSTOM description = 'All users'
```

- Set the custom nested value with key 'meta':

```
ArcadeDB> ALTER TYPE Account CUSTOM meta = { abool: true, alist: [1,2,3,4,5]};
```

- Change the type name from 'Account' to 'Client':

```
ArcadeDB> ALTER TYPE Account NAME Client;
```

Renaming a type carries what is bound to it by name: its buckets and indexes (those it declares and the sub-indexes it holds for the indexes of its super types; renaming a super type leaves the sub-indexes of its sub types as they are), the files a TIMESERIES type keeps its samples in (the mutable bucket and sealed file of each shard, and the tag dictionary; the rename is refused, before anything is changed, when a file of that name is already in the database directory; when that name is

held by a bucket, such as `X_shard_0` for a rename to `X`, the refusal is an "already exists" conflict, HTTP `409`), the materialized views and continuous aggregates that read it (their stored source type and the `FROM` of their defining query), the triggers defined on it, and the graph analytical views that list it (their definition is updated and the view rebuilt). The rename is refused, before anything is changed, when the type is the backing type of a materialized view or a continuous aggregate (it is named after the view), or when a view's query names the type in a form that cannot be re-pointed with certainty. *(Since v26.11.1. Before, a view or an aggregate kept naming the old type: its refresh failed, `DROP TYPE` on the renamed source was let through, and a trigger came back unbound after a restart. A renamed TIMESERIES type came back empty after a restart, its samples left in the files named after the old type.) (Since v26.12.1 this holds for `LSM_VECTOR` indexes too, issue #12035. Before, their per-bucket indexes kept the old type name, so `SELECT FROM schema:indexes` failed with `Type with name '…' was not found` until a restart, and a type created later under the old name was listed as their owner.)*

A property declared `EMBEDDED`, `LINK`, `LIST` or `MAP OF` the renamed type follows the rename too, and new embedded documents of the renamed type can be written, whether the property declares the type or not. *(Since v26.12.1, issue #11305. Before, the property kept the old name and every write of it failed with "Type with name ... was not found", and no embedded document of the renamed type could be written at all.)* An embedded document written BEFORE the rename still names the old type and is not read back until the type is renamed back (issue #11336): rename a type that holds embedded documents only once the data can be re-imported.

An index with the default name follows the rename: `Account[id]` becomes `Client[id]`, and `DROP INDEX` and `REBUILD INDEX` find it by the new name at once (the old name no longer resolves). An index created with a name of its own (`CREATE INDEX AccountIds ON Account (id) UNIQUE`) keeps that name across the rename and across a restart. A rename is refused, and nothing changes, when an index of the type would take a name another index already has. *(Since v26.11.1. Before, the index was listed under the new name but resolved only by the old one, and a name of its own was replaced by the default one.)*

- Add the former type name 'Account' as alias for 'Client':

```
ArcadeDB> ALTER TYPE Client ALIASES Account;
```

Using `ALIASES` overwrites any previous `ALIASES` of this type.

A statement that names the type by an alias answers what it answers with the type's name, an index used or not: `SELECT`, `UPDATE` and `DELETE`, `MATCH` (`type:`) and `INSTANCEOF`, on the type and on its sub-types. *(Since v26.12.1, issue #12968. Before, a statement through an alias that the planner answered with an index returned no rows, and an `UPDATE` or `DELETE` reported `count: 0` and changed nothing.)*

For more information, see:

- `CREATE TYPE`
- `DROP TYPE`.

Supported Attributes

Attribute	Type	Description
`NAME`	Identifier	Changes the type name.
`ALIASES`	Identifier(s)	Add one or more (comma-separated) alternate name(s) for the type name.
`SUPERTYPE`	Identifier	Defines a super-type for the type. Use `NULL` to remove a super-type assignment. It supports multiple inheritances. To set or add a new type, you can use the syntax `+<type>`, to remove it use `-<type>`.
`BUCKET`	Identifier	Name of the bucket: `+` to add a bucket and `-` to remove it from the type. If the bucket doesn't exist, it creates a physical bucket. Adding buckets to a type is also useful in storing records in distributed servers.
`BUCKETSELECTIONSTRATEGY`	Identifier	Change the selection of the bucket when new records are created. Using `partitioned()` is recommended when your have a unique id

14.11.3. SQL - `DROP TYPE`

Removes a type from the schema. To drop a type (safely), first, all its instances need to be removed.

Syntax

```
DROP TYPE <type> [IF EXISTS] [UNSAFE]
```

- `<type>` Defines the type you want to remove.
- `IF EXISTS` Prevent errors if the type does not exist when attempting to drop it.
- `UNSAFE` Defines whether the command drops non-empty edge and vertex types. Note, this can disrupt data consistency. Be sure to create a backup before running it.

Bear in mind, that the schema must remain coherent. For instance, avoid removing types that are super-types to others. The buckets of the type are dropped with it, together with the paired `_ext` buckets that hold the values of its EXTERNAL properties.

An edge type whose edges are all lightweight is refused without `UNSAFE`, as any edge type that still has edges is. *(Since v26.11.1, issues #10923 and #11142. Before, the check looked for edge records only, so the drop went through and left entries in the*

vertices that no command could read or delete.)

The paired `ext` bucket of a bucket detached from the type earlier (with `ALTER TYPE … BUCKET -`) is kept, since the records of the detached bucket still point into it. *(Since v26.12.1, issue #12995. Before, `DROP TYPE` left every paired `_ext` bucket behind: its file stayed on disk with the EXTERNAL values, it stayed in the schema, `CHECK DATABASE` counted its records, and a type created later under the same name took it over with the old values.)*

`DROP TYPE` is refused, even with `UNSAFE`, while a materialized view, a continuous aggregate or a graph analytical view reads the type. Drop the view first. *(Since v26.11.1 for graph analytical views. Before, the drop went through and the view failed to build at the next open.)*

`DROP TYPE` is refused, even with `UNSAFE`, while a property of another type is declared `EMBEDDED`, `LINK`, `LIST` or `MAP OF` the type (or of one of its aliases). The message names the property: drop it first with `DROP PROPERTY <Type>.<property>`. A property of the dropped type itself, one declared of its own type included, goes with it. *(Since v26.12.1, issue #11305. Before, the drop went through and every write of the property, and every read of a record holding one, failed to resolve the type.)*

Dropping a type links its sub types directly to its super types. When that would link a sub type directly under the backing type of a materialized view or a continuous aggregate, or under a type below one, `DROP TYPE` is refused before anything is changed, with a message naming the link to take out first, for example `ALTER TYPE` `` `Deep` SUPERTYPE -`Mid` `` to drop `Mid` out of `Deep -> Mid -> <view>`. Once that link is removed, the drop goes through. Such a chain exists only in a database written before 26.11.1, which refuses a type extending a backing type (issue #11204). *(Since v26.12.1, issue #11272. Before, the drop linked the sub type under the backing type anyway.)* The same refusal applies when the new direct link would join types of different kinds, which only a database written before 26.12.1 can hold (for example a document type under a vertex type under another vertex type: dropping the middle type is refused, naming the link to take out first). A hierarchy that follows the rule can be dropped from the middle as before. *(Since v26.12.1, issue #12029.)*

`DROP TYPE` is refused, even with `UNSAFE`, while an index of one of its sub types (at any depth) stands on a property that sub type would no longer inherit after the drop, the same rule `DROP PROPERTY` applies. For example, with `C EXTENDS P`, `P.k` declared only on `P` and an index `C[k]`, `DROP TYPE P` answers `Cannot drop the type 'P' because the property 'k' that type 'C' would no longer inherit is used by index 'C[k]'` and changes nothing: drop the index first, or declare the property on the sub type. A property the sub type keeps does not count: one it declares itself, one it inherits through another super type, or one declared by a super type of the dropped type. The dropped type's own indexes are dropped with it, as before. *(Since v26.12.1, issue #12318. Before, the drop went through and left the index keyed on a property no type declared, so values were stored with one type and indexed with*

another.)

A database written by a version affected by issue #8169 can still contain a type that was dropped but came back after a restart with no buckets: inserting into it fails with `Cannot retrieve a bucket for type ⋯ because there are no buckets associated`. The database logs a warning at open naming every type that has no buckets and no subtypes, and `DROP TYPE <name>` removes such a type permanently.

`DROP TYPE`, `ALTER TYPE ⋯ BUCKET -` and `ALTER TYPE ⋯ SUPERTYPE -` ask what can refuse them before they change anything, and take the locks of the index files they delete once, up front. A file that cannot be deleted (it is not a regular file, or its directory cannot be written), an index that names a type that does not exist, or a lock another thread holds refuses the statement with the schema as it was (`Nothing was changed`; a lock refusal is retryable). It is a pre-check, not a guarantee: a disk error on one of the deletes after it passed is not undone, and leaves the statement part way. *(Since v26.11.1, issue #10421. Before, a refusal part way left the sub types unlinked and the indexes before it dropped, and saved that.)*

`IF EXISTS` always returns exactly one row, whether the object was dropped or wasn't there to begin with. The `dropped` property tells the two apart:

```
DROP TYPE Person IF EXISTS
--> { "operation": "drop type", "typeName": "Person", "dropped": true }

DROP TYPE Person IF EXISTS
--> { "operation": "drop type", "typeName": "Person", "dropped": false
}
```

The same applies to `DROP BUCKET` and `DROP PROPERTY`.

Before that version, these three commands returned no rows at all when the object didn't exist.

Examples

- Remove the type `Account`:

```
ArcadeDB> DROP TYPE Account
```

For more information, see:

- `CREATE TYPE`
- `ALTER TYPE`

14.11.4. SQL - TRUNCATE TYPE

Deletes records of all buckets defined as part of the type.

By default, every type has an associated bucket with the same name. This command operates at a lower level than DELETE. This commands ignores sub-types, (That is, their records remain in their buckets). If you want to also remove all records from the type hierarchy, you need to use the POLYMORPHIC keyword.

Truncation is not permitted on a vertex or edge type that holds records, but you can force its execution using the UNSAFE keyword. Forcing truncation is strongly discouraged, as it can leave the graph in an inconsistent state. With POLYMORPHIC, the refusal covers the whole hierarchy below the type, and the error names the type that actually holds the records.

Add UNSAFE where the truncation is intended:

```
TRUNCATE TYPE Person UNSAFE
```

TRUNCATE BUCKET refuses in the same way, so the two commands agree on the same data.

Syntax

```
TRUNCATE TYPE <type> [ POLYMORPHIC ] [ UNSAFE ]
```

- <type> Defines the type you want to truncate.
- POLYMORPHIC Defines whether the command also truncates the type hierarchy.
- UNSAFE Defines whether the command forces the truncation on vertex or edge types.

Transactions

TRUNCATE TYPE behaves differently depending on whether a transaction is already active when it runs, and the result set reports which of the two applied through the transactional field:

- **Inside a transaction** (transactional: true) the truncate is part of that transaction: the records are deleted with the type's indexes maintained record by record, and a ROLLBACK puts every record back. This is the behaviour to rely on when the truncate is one half of a reload, e.g. BEGIN; TRUNCATE TYPE Staging UNSAFE; INSERT …; COMMIT; - if the insert fails, the previous contents survive. The arcadedb.truncateBatchSize setting does not apply.
- **With no transaction active** (transactional: false) the command owns a transaction of its own and takes a faster path: it drops the type's indexes, deletes the records in batches of arcadedb.truncateBatchSize (default 1000), each committed separately, and recreates the (empty) indexes at the end. This is considerably cheaper on a large type, but the command commits as it goes, so it cannot be undone.

Over HTTP the request's auto-commit transaction is what makes the command transactional; pass "autoCommit": false in the request payload to select the faster, non-undoable path for a bulk clear.

On a TIMESERIES type (since 26.11.1) the command removes samples, and it never takes part in the transaction it runs in, whatever the paragraphs above say of other types: the samples are gone, durably, when the statement returns, and a `ROLLBACK` does not bring them back (the result carries `transactional: false`, with the `count` of the samples removed). `DELETE FROM` a TIMESERIES type behaves the same way. See Deleting Samples.

A lightweight edge type with a subtype is an exception to "ignores sub-types by default": since a lightweight edge has no bucket of its own, the command refuses without `POLYMORPHIC` instead of silently truncating only part of the hierarchy.

Examples

- Remove all records of the type `Profile`:

```
ArcadeDB> TRUNCATE TYPE Profile
```

For more information, see:

- `DELETE`
- `TRUNCATE BUCKET`
- `CREATE TYPE`

14.12. Properties

14.12.1. SQL - `CREATE PROPERTY`

Creates a new property in the schema. It requires that the type for the property already exist on the database.

Syntax

```
CREATE PROPERTY
<type>.<property> [IF NOT EXISTS] <data-type> [OF <embd-type>]
[(<property-constraint>[,]*)]
[CUSTOM <custom-key> = <custom-value>[,]*]
```

- `<type>` Defines the type for the new property.
- `<property>` Defines the logical name for the property.
- `<data-type>` Defines the property data type. For supported types, see the table below.
- `<embd-type>` Defines the property's embedded type (only for embedding property types `EMBEDDED`, `LIST`, `MAP`, as well as `LINK`).
- `<custom-key>` Name of a custom property to set on the new property. Multiple pairs can be listed, separated by commas.

- `<custom-value>` Value for the custom property. All data types are supported as values.
- `<property-constraint>` See `ALTER PROPERTY <attribute-name> * <attribute-value>`
  - `mandatory <true|false>` If true, the property must be present. Default is **false**
  - `notnull <true|false>` If true, the property, must be present and cannot be null. Default is **false**
  - `readonly <true|false>` If true, the property cannot be changed after the creation of the record. Default is **false**
  - `hidden <true|false>` **Deprecated, accepted but ignored.** See the note below. Default is **false**
  - `min <number|string>` Defines the minimum value for this property. For number types it is the minimum number as a value. For strings it represents the minimum number of characters. For dates is the minimum date (uses the database date format). For collections (lists, sets, maps) this attribute determines the minimally required number of elements.
  - `max <number|string>` Defines the maximum value for this property. For number types it is the maximum number as a value. For strings it represents the maximum number of characters. For dates is the maximum date (uses the database date format). For collections (lists, sets, maps) this attribute determines the maximally allowed number of elements.
  - `regexp <string>` Defines the mask to validate the input as a Regular Expression. A pattern is matched against the text form of the value; a `LINK` is matched as the RID of the record it points to (e.g. `#12:0`). *(Since v26.10.2)* It is refused when it is declared if no value could ever pass it: on a `BINARY`, `LIST`, `MAP`, `EMBEDDED` or `ARRAY_OF_*` property, on a `BOOLEAN` when it matches neither `true` nor `false`, when it is not a valid regular expression, or when no value of the type can be written as a text it matches. A pattern on a number, a temporal type or a link is allowed when some value can match it (e.g. `'[0-9]{5}'` on a `LONG`, `'#12:.*'` on a `LINK`) and refused otherwise (e.g. `''` on a `LONG` or a `DATE`, `''[0-9]{5}''` on a `BYTE`, or `''[0-9]{25}''` on a `LONG`, which has at most 19 digits). The same check runs in `CREATE PROPERTY`, `ALTER PROPERTY ... REGEXP` and the Java `Property.setRegexp()`. A pattern the check cannot settle within its search budget (back references, look-arounds, or a pattern such as `''[0-9],[0-9]+'` on a `LONG`) is accepted with a WARNING in the server log naming `ALTER PROPERTY ⋯ REGEXP null`. The check judges a `DATE` or `DATETIME` by the `arcadedb.dateImplementation` in effect when the property is declared. A database that already holds a refused declaration still opens with a warning, and a write of such a property is refused with a message naming `ALTER PROPERTY ⋯ REGEXP null`, which removes it.
  - `default <any>` Defines the value to use when the property is not present. A property set to `NULL` on purpose is present: `INSERT INTO T SET id = 1, df = null` stores `NULL`, and only a statement that leaves `df` out stores the default, as in SQL. `UPDATE ⋯ APPLY DEFAULTS` fills the properties that are null as well as the missing ones. *(Since v26.11.1: earlier versions stored the default for an explicit `NULL`.)* The value is an **SQL expression**, evaluated once per new record, not a literal: write `default 'active'` for the string `active`, since an unquoted `active` is a field reference. The expression is compiled and validated when the property is declared, so a malformed one is rejected right there rather than silently producing garbage on every record. A default written with literals only (a string, a number, a boolean, `null`, or arithmetic over them) is also converted to the property's type at that moment, and refused when the conversion fails, e.g. `INTEGER (default 'dflt')` or `DATE (default 'notadate')`; the property keeps the default it had. A default that calls a function, such as `sysdate()`, is still evaluated

per record. *(Since v26.12.1: earlier versions accepted such a default and then refused every new record of the type.)* A database that already holds one still opens with a warning in the server log, and `ALTER PROPERTY ⋯ DEFAULT` repairs it. A default that evaluates to `null` fills in nothing, which makes it equivalent to declaring no default at all - see the note below.

- `external <true|false>` If true, the property's value is stored in a paired **external bucket** instead of inline in the primary record. The main record carries only a pointer (`[bucketId][position]`) to the external record. This keeps the primary bucket pages dense for graph traversal and pushes heavy payloads (vector embeddings, long strings, embedded JSON, full-text payloads) to a separate file that traversal-only queries do not touch. Default is **false**. See External property storage in the concepts section for details.
- `compression '<none|lz4|auto>'` Only meaningful together with `external true`. Controls compression of the value stored in the paired bucket: `none` (default) writes raw bytes, `lz4` always LZ4-compresses, `auto` tries LZ4 and keeps the compressed form only if it saves more than 10% (decided per record, so a single property can mix compressed and uncompressed records).
- `IF NOT EXISTS` If specified, create the property only if not exists. The statement returns one row either way, carrying a `created` flag: see the note on `IF NOT EXISTS` results. If a property with the same name already exists in the type, then no error is returned

The `hidden` attribute was **removed** and there is **no plan to reimplement it**. It is still accepted by the parser and stored in the schema so that existing scripts and databases keep working, but it no longer has any effect on queries: a property flagged as `hidden` is returned by `SELECT * FROM` exactly like any other property. Do not use it to conceal data: to keep a property out of a result set, list the properties you want explicitly (`SELECT name, email FROM ⋯`) instead of using `*`; to restrict who can read a property, use the security layer.

When you create a property, ArcadeDB reads the records already stored in the type (including its subtypes) and checks them against the declaration: the declared type (a value that cannot be read as that type, such as `'abc'` for an `INTEGER`, is refused), `mandatory`, `notnull`, `min`, `max` and `regexp`. If a record does not satisfy the declaration, the statement fails, names the first offending record, and the property is not created. `ALTER PROPERTY ⋯ MANDATORY`, `NOTNULL`, `MIN`, `MAX` and `REGEXP`, and the openCypher `NOT NULL` and `NODE KEY` constraints, make the same check. *(Since v26.12.1)* The refusal is a `ValidationException` whose kind names the constraint (`MANDATORY`, `NOT_NULL`, `MIN`, ...), answered over HTTP with `400` and the kind in `exceptionArgs`. Before, a `MANDATORY` or `NOTNULL` refusal was a `CommandExecutionException` answered with `500`. On a type with many records this scan takes time proportional to the number of records.

A `STRING` property is refused over records that already hold the property as a non-string value (an integer, a long, a float, a date, a boolean, ...), the same way `INTEGER` is refused over a stored `'abc'`. The error names the record and suggests converting the values first, for example:

```
UPDATE T SET n = n.asString()
```

```
CREATE PROPERTY T.n STRING
```

This behavior changed in 26.11.1 (issue #10204). Before, `CREATE PROPERTY T.n STRING` was accepted without converting the stored values, and an index on `n` then compared them as text while a scan compared them natively, so range conditions and `ORDER BY` could disagree depending on whether the index was used. A script that declares a `STRING` property over existing non-string values now fails until the values are converted.

(Since v26.11.1) `CREATE PROPERTY` and `ALTER PROPERTY` read an attribute value the same way. `null` removes the attribute (`regexp null`, `min null`, `max null`, `compression null`), while a quoted `'null'` is the text. A flag (`mandatory`, `notnull`, `readonly`, `hidden`, `external`) takes `true` or `false`. A bare word such as `ALTER PROPERTY Doc.name REGEXP abc` is a field reference, not a value, and is refused: quote it (`'abc'`). Earlier versions stored the text `null` as the pattern of `CREATE PROPERTY … (regexp null)`, which left the property unable to hold any other value, and silently removed the attribute on `ALTER PROPERTY` with a bare word.

A `LIST OF <type>` or `MAP OF <type>` property only accepts elements of that type. A single value written to a `LIST OF <type>` property is stored as a one-element list, converted to the element type like the elements of a list are: `longs = 5` stores `[5]` as a `LONG`. A number outside the range of the element type is refused instead of being clamped. For a document type, a plain map becomes an embedded document of that type, and a value that is not a document of that type (a number, a string, a map typed as another type) is refused.

When constraints are set, such as `mandatory`, `read-only`, etc., it's not possible to use the Gremlin syntax to add vertices and edges without incurring in a Validation exception. The issue is that Gremlin API sets the properties after the creation of the vertex and edge.

Properties of the type `LINK` may have the value `NULL`, furthermore accessing a `LINK` property which is not set results also in a `NULL` value.

A default value of `NULL` is the same as no default value at all

A default exists to supply a value when one is missing, so a default that evaluates to `NULL` supplies nothing and the property is left unset: `has()` returns false and the key is absent from the stored record, exactly as if no default had been declared. This holds however the `NULL` arises - `default null`, or an expression that evaluates to it such as `ifnull(NULL, NULL)`.

It follows that a `NULL` default cannot satisfy `mandatory`: nothing is filled in, so the record is rejected with `is mandatory, but not found`, which points at the missing value rather than at the default. Conversely it never trips `notnull`, which only applies to a property that is present.

Examples

- Create the property `name` of the string type in the type `User`:

```
ArcadeDB> CREATE PROPERTY User.name STRING
```

- Create a property called `tags` in the type `Profile` only allowing list members of type string:

```
ArcadeDB> CREATE PROPERTY Profile.tags LIST OF STRING
```

- Create the property `friends`, as an embedded map:

```
ArcadeDB> CREATE PROPERTY Profile.friends MAP
```

- Create the property `address`, as an embedded document:

```
ArcadeDB> CREATE PROPERTY Profile.address EMBEDDED
```

- Create the property `createdOn` of type date with additional constraints:

```
ArcadeDB> CREATE PROPERTY Transaction.createdOn DATE (mandatory true, notnull true,
readonly true, min "2010-01-01")
```

- Create the property `salary` only if it does not exist:

```
ArcadeDB> CREATE PROPERTY Employee.salary IF NOT EXISTS double
```

- Create the property `hiredAt` with the time of creation being the default value:

```
ArcadeDB> CREATE PROPERTY Employee.hiredAt DATETIME (readonly, default sysdate('YYYY-
MM-DD HH:MM:SS'))
```

- Create the property `embedding` for vector data, stored in a paired external bucket so traversal-only queries do not pay for it:

```
ArcadeDB> CREATE PROPERTY Person.embedding ARRAY_OF_FLOATS (EXTERNAL true)
```

- Create the property `body` stored externally with adaptive LZ4 compression (compresses text, falls back to raw when compression would not pay off):

```
ArcadeDB> CREATE PROPERTY Doc.body STRING (EXTERNAL true, COMPRESSION 'auto')
```

- Create the property `num` with the custom property 'oflines':

```
ArcadeDB> CREATE PROPERTY Doc.num INTEGER (readonly true) CUSTOM oflines = 1
```

- Create the property `width` with multiple custom properties:

```
ArcadeDB> CREATE PROPERTY Doc.width FLOAT CUSTOM unit = 'meters', scale = 2
```

Declaring `CUSTOM` inline is equivalent to running `ALTER PROPERTY … CUSTOM` right after the creation, but it happens in the same statement. When `IF NOT EXISTS` is used and the property is already there, the whole statement is skipped, so the custom properties are not applied to the existing property.

For more information, see:

- `ALTER PROPERTY`
- `DROP PROPERTY`
- External property storage

Supported Types

ArcadeDB supports the following data types for standard properties:

`BOOLEAN`	`BYTE`	`SHORT`	`INTEGER`	`LONG`
`STRING`	`LINK`	`BINARY`	`DATE`	`DATETIME`
`FLOAT`	`DOUBLE`	`DECIMAL`	`ZONED_DATETIME`	`DURATION`
`DATETIME_SECOND`	`DATETIME_MICROS`	`DATETIME_NANOS`	`LOCAL_TIME`	`OFFSET_TIME`

`LOCAL_TIME`, `OFFSET_TIME`, `ZONED_DATETIME` and `DURATION` are available since v26.10.2 (see Managing Dates).

It supports the following data types for container properties:

`LIST`	`MAP`	`EMBEDDED`

14.12.2. SQL - `ALTER PROPERTY`

Change a property defined in the schema. The change is persistent.

Syntax

```
ALTER PROPERTY <type-name>.<property-name> <attribute-name> <attribute-value> [CUSTOM
<custom-key> = <custom-value>]
```

```
ALTER PROPERTY <type-name>.<property-name> NAME <new-property-name>
```

- `<type-name>` Defines the type where the property is defined
- `<property-name>` Defines the property in the `type-name` you want to change
- `<attribute-name>` Defines the attribute you want to change. For a list of supported attributes, see the table below
- `<attribute-value>` Defines the value you want to set
  - `name <new-property-name>` Renames the property. This is a schema-metadata-only rename: it does not touch existing records. Records keep answering under the old name until they are next written, at which point they pick up the new name. Not allowed on an indexed property, on a column declared directly on a TIME SERIES type, or if `<new-property-name>` already exists on the type.
  - `mandatory <true|false>` If true, the property must be present. Default is false. Setting it to true scans the existing records and is refused, naming the first offending record, if any of them has no value for the property.
  - `notnull <true|false>` If true, the property, if present, cannot be null. Default is false. Setting it to true is refused, naming the first offending record, if an existing record holds a null for the property.
  - `readonly <true|false>` If true, the property cannot be changed after the creation of the record. Default is false
  - `hidden <true|false>` **Deprecated, accepted but ignored.** See the note below.
  - `min <number|string>` Defines the minimum value for this property. For number types it is the minimum number as a value. For strings, it represents the minimum number of characters. For dates is the minimum date (uses the database date format). For collections (lists, sets, maps) this attribute determines the minimally required number of elements.
  - `max <number|string>` Defines the maximum value for this property. For number types it is the maximum number as a value. For strings, it represents the maximum number of characters. For dates is the maximum date (uses the database date format). For collections (lists, sets, maps) this attribute determines the maximally allowed number of elements.
  - `regexp <string>` Defines the mask to validate the input as a Regular Expression. Refused when no value of the property's type could pass it, including a number or temporal type (`'[A-Z]+'` on a `LONG` or a `DATE`), as for `CREATE PROPERTY`. A pattern the check cannot settle is accepted with a WARNING.
  - `default <any>` Defines the value to use when the property is not present, as an **SQL expression** evaluated once per new record. Quote a literal (`default 'active'`); an unquoted `active` is a field reference and is rejected. A literal default that cannot be converted to the property's type (`default 'zz'` on an `INTEGER`) is refused and the previous default is kept, as for `CREATE PROPERTY`. A default that evaluates to `null` is equivalent to no default at all - see `CREATE PROPERTY`.
  - `external <true|false>` Toggles external storage of the property's value (see External property storage). Switching the flag does not migrate existing records eagerly; they are

converted lazily on the next write, or all at once with `REBUILD TYPE`.

- `compression '<none|lz4|auto>'` Sets the compression mode for an EXTERNAL property's payload (`none`, `lz4`, or `auto`). Has no effect on inline properties.

- `<custom-key>` Name of the custom property to set
- `<custom-value>` Value for the custom property. All data types are supported as values.

The `hidden` attribute was **removed** and there is **no plan to reimplement it**. It is still accepted and persisted in the schema for backward compatibility, but it has no effect on queries. See `CREATE PROPERTY` for details.

A custom property can be deleted by setting its value to `null`.

Examples

- Set the property 'subscribedOn' as mandatory:

```
ArcadeDB> ALTER PROPERTY User.subscribedOn MANDATORY true
```

- Set the property 'createdOn' as read-only

```
ArcadeDB> ALTER PROPERTY User.createdOn READONLY true
```

- Set the custom value with key 'description':

```
ArcadeDB> ALTER PROPERTY Account.signedOn CUSTOM description = 'First Sign In'
```

- Move an existing property to external storage and enable adaptive LZ4 compression:

```
ArcadeDB> ALTER PROPERTY Doc.body EXTERNAL true
ArcadeDB> ALTER PROPERTY Doc.body COMPRESSION 'auto'
```

- Move a property back to inline storage:

```
ArcadeDB> ALTER PROPERTY Doc.body EXTERNAL false
```

- Rename the property 'surname' to 'lastName':

```
ArcadeDB> ALTER PROPERTY User.surname NAME lastName
```

For more information, see:

- `CREATE PROPERTY`

- `DROP PROPERTY`
- `ALTER TYPE`
- External property storage

14.12.3. SQL - `DROP PROPERTY`

Removes a property from the schema. Does not remove the property values in the records, it just changes the schema information. Records continue to have the property values, if any. Anything declared on the property, such as a `DEFAULT` value or a constraint, stops applying at once: new records are created without it.

Syntax

```
DROP PROPERTY <type>.<property> [IF EXISTS] [FORCE]
```

- `<type>` Defines the type where the property exists.
- `<property>` Defines the property you want to remove.
- `IF EXISTS` Prevent errors if the property does not exist when attempting to drop it. See here for the shape of the result.
- `FORCE` In case one or more indexes are defined on the property, the command will throw an exception that names them. This includes the indexes a subtype defined on the inherited property, unless the subtype declares its own property with the same name. Use `FORCE` to drop all those indexes together with the property. The statement is all or nothing: if the property cannot be dropped anyway (for example, because a supertype has an index on a property with the same name), it fails before any index is dropped

Examples

- Remove the `name` property from the type `User`:

```
ArcadeDB> DROP PROPERTY User.name
```

For more information, see:

- `ALTER PROPERTY`
- `CREATE PROPERTY`

14.13. SQL Indexes

14.13.1. SQL - `CREATE INDEX`

Creates a new index. Indexes can be:

- **Unique** Where they don't allow duplicates (LSM Tree).

- **Not Unique** Where they allow duplicates (LSM Tree).
- **Unique Hash** O(1) equality lookups, no duplicates (Hash Index).
- **Not Unique Hash** O(1) equality lookups, allows duplicates (Hash Index).
- **Full Text** Where they index any single word of text.
- **Geospatial** Where they index WKT geometry strings for spatial queries.
- **HNSW** (Hierarchical Navigable Small World) vector index.
- **LSMVectorIndex** LSM Tree-based vector index for persistent similarity search.

> There are several index algorithms available to determine how ArcadeDB indexes your database. For more information on these, see Indexes.

Syntax

```
CREATE INDEX [<manual-index-name>]
[ IF NOT EXISTS ]
[ ON <type> (<property> [BY KEY|VALUE|ITEM] [COLLATE CI][,]*) ]
<index-type> [<key-type>]
[ NULL_STRATEGY SKIP|ERROR|INDEX]
```

- `<manual-index-name>` Only for manual indexes, defines the logical name for the index. For automatic indexes, the index name is assigned automatically by ArcadeDB at creation as `<type>[<property>[,]*]`. For example, the index created on type Friend, properties "firstName" and "lastName", it will be named "Friend[firstName,lastName]"
- `IF NOT EXISTS` Specifying this option, the index creation will just be ignored if the index already exists (instead of failing with an error). An index counts as already existing when it covers the same properties with the same index type, either on the type named in the statement or on one of its **super types**: an index declared on a super type indexes the records of all its sub types, so there is nothing left to create. An index of the same name on an unrelated type, on a **sub** type, on different properties, or of a different index type is still reported as an error, because none of those provides what the statement asked for. It is safe to send the same statement from several connections at once: exactly one of them builds the index and returns `created: true`, and the others wait for it and return `created: false` (see the note on `IF NOT EXISTS` results).

  ```
  CREATE INDEX `Element_uuid` IF NOT EXISTS ON `Element` (uuid) UNIQUE;
  -- ignored: the index on the super type already covers UnidentifiedElement
  CREATE INDEX `Element_uuid` IF NOT EXISTS ON `UnidentifiedElement` (uuid) UNIQUE;
  ```

- `<type>` Defines the type to create an automatic index for. The type must already exist.
- `<property>` Defines the property you want to automatically index. The property must already exist.
  - `BY KEY` index by key names for map properties.
  - `BY VALUE` index by values for map properties.

    - `BY ITEM` index by values for list elements.
    - `COLLATE CI` enables case-insensitive collation for this property. Values are stored in lowercase internally, so lookups match regardless of case. Can be specified independently for each property in a composite index. The only accepted values are `CI` and `DEFAULT`, in any letter case; anything else is reported as an error rather than ignored, so a misspelled collation cannot silently produce a case-sensitive index. See concepts/indexes.pdf for details.
- `<index-type>` Defines the index type you want to use:
    - `UNIQUE` does not allow duplicate keys (LSM Tree),
    - `NOTUNIQUE` allows duplicate keys (LSM Tree),
    - `UNIQUE_HASH` does not allow duplicate keys (Hash Index — O(1) equality lookups, no range queries),
    - `NOTUNIQUE_HASH` allows duplicate keys (Hash Index — O(1) equality lookups, no range queries),
    - `FULL_TEXT` based on any single word of text. Uses Apache Lucene for tokenization and supports configurable analyzers, multi-property indexing, advanced query syntax, and BM25 relevance scoring. See how-to/data-modeling/full-text-index.pdf for full documentation.
    - `GEOSPATIAL` indexes WKT geometry strings stored in `STRING` properties, enabling spatial queries via the `geo.*` SQL function namespace. See how-to/data-modeling/geospatial.pdf for full documentation.
    - `HNSW` vector index (Hierarchical Navigable Small World).
    - `LSMVECTORINDEX` LSM Tree-based vector index.
    - `LSM_SPARSE_VECTOR` posting-list inverted index for sparse-vector retrieval (BM25-style, SPLADE, BGE-M3, OpenSearch sparse). See Sparse Vector Search for full documentation.
- `<key-type>` Defines the key type. With automatic indexes, the key type is automatically selected when the database reads the target schema property. For manual indexes, when not specified, it selects the key at run-time during the first insertion by reading the type of the type. In creating composite indexes, it uses a comma-separated list of types.

To create an automatic index bound to the schema property, use the `ON` clause. In order to create an index, the schema must already exist in your database.

In the event that the `ON` and `<key-type>` clauses both exist, the database validates the specified property types. If the property types don't equal those specified in the key type list, it throws an exception.

The `NULL_STRATEGY` determines how null values are indexed:

- `SKIP` (default) excludes null values from the index,
- `ERROR` throws an error in case a null value is found during indexing,
- `INDEX` adds null values to the index.

With the default `SKIP`, an ascending `ORDER BY` on an indexed property has to scan

the type for records with a null value, because nulls sort first and are not in the index. The scan is skipped when the property is declared `NOTNULL` or the `WHERE` clause excludes nulls (`IS NOT NULL`, `=`, `<`, `>`). For optional properties read in order with a `LIMIT`, create the index with `NULL_STRATEGY INDEX`.

A unique index does not regard derived types or embedded documents of the indexed type.

While `CREATE INDEX` is still building the index, queries do not use it: `SELECT`, `database.select()` and Cypher scan the type instead, so their results are complete, and `lookupByKey()` answers as if the index did not exist yet. Queries use the index as soon as the command completes. Since 26.10.2.

A transaction that began before the index was created, in another session or as an outer transaction of the caller, and wrote to the indexed type, does not hold up `CREATE INDEX` and is not left out of the index: the build indexes the committed state, and that transaction queues the entries of the new index for what it wrote when it commits, as if the index had existed when it saved the records. A `UNIQUE` index refuses that commit with a `DuplicatedKeyException` if another record took the key in the meantime. A transaction that wrote to the same bucket both before and after the index was created is refused at its commit with a retryable `ConcurrentModificationException` instead, because it cannot tell the entries it queued from the ones it owes: the retry finds the index there from its first write. An outer transaction that owes entries cannot be committed while a transaction nested inside it is still open (`TransactionException`): commit or roll back the innermost one first. Since 26.11.1; before, such a record was missing from every index-backed query and a `UNIQUE` index accepted a duplicate of its key.

> You can use list key types when creating manual composite indexes, but bear in mind that such indexes are not yet fully supported.

Examples

- Create an automatic index bound to the new property `id` in the type `User`:

```
ArcadeDB> CREATE PROPERTY User.id BINARY
ArcadeDB> CREATE INDEX ON User (id) UNIQUE
```

- Create a series of automatic indexes for the `thumbs` map property in the type `Movie`. `BY KEY` indexes the map's keys, `BY VALUE` its values. They are usually `NOTUNIQUE`, since the same key (or value) normally appears across many records:

```
ArcadeDB> CREATE INDEX ON Movie (thumbs BY KEY) NOTUNIQUE
ArcadeDB> CREATE INDEX ON Movie (thumbs BY VALUE) NOTUNIQUE
```

Once created, these indexes are automatically used when filtering a map by key or by value:

```
ArcadeDB> SELECT FROM Movie WHERE thumbs CONTAINSKEY 'poster'
ArcadeDB> SELECT FROM Movie WHERE thumbs CONTAINSVALUE 'poster.png'
```

- Create a index for the `actors` list property in the type `Movie`. `BY ITEM` indexes each element of the list, so a filter on a single value uses the index:

```
ArcadeDB> CREATE INDEX ON Movie (actors BY ITEM) NOTUNIQUE
ArcadeDB> SELECT FROM Movie WHERE actors = 'Keanu Reeves'
```

The key type of a `BY ITEM` index is the declared item type of the list (`LIST OF DOUBLE` is keyed as DOUBLE), so `WHERE prices CONTAINS 7` finds a stored `7.0` with or without the index. A list without a declared item type is keyed as STRING. An index created before this rule keeps its STRING keys until it is rebuilt (`REBUILD INDEX`).

- A composite index can expand more than one collection: several `BY KEY` / `BY VALUE` properties, of different maps or of the same one, and `BY ITEM` properties of one list next to them. A record is indexed under every combination of the elements, so a conjunction of membership tests uses the index:

```
ArcadeDB> CREATE INDEX ON Doc (m1 BY KEY, m2 BY VALUE) NOTUNIQUE
ArcadeDB> SELECT FROM Doc WHERE m1 CONTAINSKEY 'k1' AND m2 CONTAINSVALUE 'wanted'
```

The index holds one entry per key of `m1` times value of `m2`, so it grows with the product of the two sizes. A record whose collection has no element has no entry, which is why a filter that leaves such a column out (`WHERE m1 CONTAINSKEY 'k1'` alone) is answered without this index. *(Since v26.12.1: before, only the last modified property was expanded and the others were indexed as whole collections, so the index answered such a query with no rows. Rebuild an existing index of this shape with* `REBUILD INDEX`*.)*

- Create a series of properties and on them create a composite index:

```
ArcadeDB> CREATE PROPERTY Book.author STRING
ArcadeDB> CREATE PROPERTY Book.title STRING
ArcadeDB> CREATE PROPERTY Book.publicationYears LIST
ArcadeDB> CREATE INDEX ON Book (author, title, publicationYears) UNIQUE
```

- Create an index on an edge's date range:

```
ArcadeDB> CREATE VERTEX TYPE File
ArcadeDB> CREATE EDGE TYPE Has
ArcadeDB> CREATE PROPERTY Has.started DATETIME
ArcadeDB> CREATE PROPERTY Has.ended DATETIME
```

```
ArcadeDB> CREATE INDEX ON Has (started, ended) NOTUNIQUE
```

You can create indexes on edge types only if they contain the begin and end date range of validity. This is use case is very common with historical graphs, such as the example above.

- Using the above index, retrieve all the edges that existed in the year 2014:

```
ArcadeDB> SELECT FROM Has WHERE started >= '2014-01-01 00:00:00.000' AND
            ended < '2015-01-01 00:00:00.000'
```

- Using the above index, retrieve all edges that existed in 2014 and write them to the parent file:

```
ArcadeDB> SELECT outV() FROM Has WHERE started >= '2014-01-01 00:00:00.000'
            AND ended < '2015-01-01 00:00:00.000'
```

- Using the above index, retrieve all the 2014 edges and connect them to children files:

```
ArcadeDB> SELECT inV() FROM Has WHERE started >= '2014-01-01 00:00:00.000'
            AND ended < '2015-01-01 00:00:00.000'
```

- Create an index that includes null values.

By default, indexes ignore null values. Queries against null values that use an index returns no entries. To return an error in case of null values, append `NULL_STRATEGY ERROR` when you create the index.

```
ArcadeDB> CREATE INDEX ON Employee (address) NOTUNIQUE NULL_STRATEGY ERROR
```

- Create a case-insensitive index on a string property:

```
ArcadeDB> CREATE PROPERTY Product.Name STRING
ArcadeDB> CREATE INDEX ON Product (Name COLLATE CI) NOTUNIQUE
```

Queries like `SELECT FROM Product WHERE Name = 'Hello World'` will now match `"Hello World"`, `"HELLO WORLD"`, `"hello world"`, etc. A plain equality or `IN` is case sensitive (since 26.10.2) and a range (`>`, `>=`, `<`, `<=`, `BETWEEN`) on the plain property does not use the index; use `Name.toLowerCase() = 'john'` or `Name.toLowerCase() >= 'c'` for a case-insensitive lookup or range.

- Create a case-insensitive unique index to prevent duplicate entries ignoring case:

```
ArcadeDB> CREATE PROPERTY User.username STRING
```

```
ArcadeDB> CREATE INDEX ON User (username COLLATE CI) UNIQUE
```

This prevents inserting both `"Admin"` and `"admin"`, since they are treated as the same key.

- Create a composite index with case-insensitive collation on some properties only:

```
ArcadeDB> CREATE PROPERTY Product.Name STRING
ArcadeDB> CREATE PROPERTY Product.Code STRING
ArcadeDB> CREATE INDEX ON Product (Name COLLATE CI, Code) UNIQUE
```

In this example, `Name` uses case-insensitive collation while `Code` uses the default (case-sensitive) collation.

When a `COLLATE CI` index exists on a property, the query optimizer also recognizes queries that use `.toLowerCase()` (e.g., `WHERE Name.toLowerCase() = 'hello world'`) and will use the case-insensitive index for these queries.

- Create a unique hash index for O(1) primary key lookups:

```
ArcadeDB> CREATE PROPERTY User.email STRING
ArcadeDB> CREATE INDEX ON User (email) UNIQUE_HASH
```

- Create a non-unique hash index for fast equality lookups:

```
ArcadeDB> CREATE PROPERTY Order.status STRING
ArcadeDB> CREATE INDEX ON Order (status) NOTUNIQUE_HASH
```

Hash indexes use extendable hashing and provide O(1) equality lookups — significantly faster than LSM Tree for point queries (`WHERE key = value`). However, they do not support range scans (`<`, `>`, `BETWEEN`) or ordered iteration (`ORDER BY`). Use `UNIQUE`/`NOTUNIQUE` (LSM Tree) when you need range queries or ordering.

- Build an index over a large existing type with the sorted build:

```
ArcadeDB> CREATE INDEX ON LineItem (shipdate) NOTUNIQUE METADATA {buildMode: 'SORTED'}
```

(Since v26.10.2.) `buildMode` is `DEFAULT` (records are added one at a time) or `SORTED` (a bounded external sort writes the ordered entries straight into the index). `SORTED` is meant for bulk builds on large types, works on `UNIQUE` and `NOTUNIQUE` indexes only, and cannot run on a replicated database or inside a transaction that has pending changes. *(Since v26.11.1)* an empty transaction no longer stops it, so `SORTED` also works from Studio and over HTTP, where every command runs inside an auto-commit transaction.

(Since v26.11.1.) With `DEFAULT`, a `UNIQUE` index over a type that holds at least `index.buildSortedUniqueMinRecords` records (10,000 by default, subtypes included) is built with the sorted build. Adding records one at a time checks every key against the index being built, and that check reads every page written so far, so on a large type the build time grew with the square of the record count. A build inside a transaction with pending writes to the buckets it indexes and a build on a replicated database still add records one at a time, but over a type that large they no longer check every key against the index being built: they check the keys once, over the finished index (in the order of the index and across every bucket), so the build time grows with the record count and not with its square. A duplicate key is then refused when that last pass reaches it, with the same error, and the index is not created. Set `index.buildSortedUniqueMinRecords` to 0 to turn the switch off, and that deferred check with it.

(Since v26.11.1.) With `DEFAULT`, a `NOTUNIQUE` index over a type that holds at least `index.buildSortedNonUniqueMinRecords` records (1,000,000 by default, subtypes included) is built with the sorted build too. Nothing is quadratic here, but adding records one at a time pays the transaction bookkeeping of every key, which a bulk build over existing records does not need: when the keys fit the memory budget of the sorted build (a quarter of the free heap, unless the Java API sets one), it takes about two thirds of the time. The sorted build sizes its runs by the memory its entries take, measured from the keys, so a column of short or repeated keys is sorted in one pass over millions of records and only a table larger than the budget is spilled to disk and merged. Set `index.buildSortedNonUniqueMinRecords` to 0 to turn the switch off.

- Create a full-text index and search it using `SEARCH_INDEX()`:

```
ArcadeDB> CREATE INDEX ON Employee (address) FULL_TEXT
```

```
SELECT * FROM Employee
WHERE SEARCH_INDEX('Employee[address]', 'New York')
```

`LIKE` and `ILIKE` do not use full-text indexes. See how-to/data-modeling/full-text-index.pdf for analyzer configuration, Lucene query syntax, relevance scoring, and More Like This.

- Create a basic LSMVectorIndex for similarity search on vector embeddings:

```
ArcadeDB> CREATE VERTEX TYPE Doc
ArcadeDB> CREATE PROPERTY Doc.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {dimensions: 100,
similarity: 'COSINE'}
```

By default, `CREATE INDEX ... LSM_VECTOR` builds the HNSW graph immediately, so the index is ready to query as soon as the command completes. This is the recommended behavior for most use cases. If you prefer to defer graph construction to the first search (lazy mode), set `buildGraphNow: false` in the metadata.

- Create an LSMVectorIndex with deferred graph building (useful when bulk-loading data after index creation):

```
ArcadeDB> CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {dimensions: 100,
similarity: 'COSINE', buildGraphNow: false}
```

The graph will be built lazily on the first `vector.neighbors()` query or when `REBUILD INDEX` is executed.

- Create an LSMVectorIndex with DOT_PRODUCT similarity metric:

```
ArcadeDB> CREATE VERTEX TYPE Image
ArcadeDB> CREATE PROPERTY Image.featureVector ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON Image (featureVector) LSM_VECTOR METADATA {dimensions: 512,
similarity: 'DOT_PRODUCT'}
```

- Create an LSMVectorIndex with EUCLIDEAN distance metric and custom performance parameters:

```
ArcadeDB> CREATE VERTEX TYPE Product
ArcadeDB> CREATE PROPERTY Product.attributes ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON Product (attributes) LSM_VECTOR METADATA {dimensions: 256,
similarity: 'EUCLIDEAN', maxConnections: 32, beamWidth: 200}
```

- Create an LSMVectorIndex with minimal metadata:

```
ArcadeDB> CREATE VERTEX TYPE CompactDoc
ArcadeDB> CREATE PROPERTY CompactDoc.vec ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX IF NOT EXISTS ON CompactDoc (vec) LSM_VECTOR METADATA
{dimensions: 4, similarity: 'COSINE'}
```

- Create an LSMVectorIndex with tuned performance parameters for large-scale indexing:

```
ArcadeDB> CREATE VERTEX TYPE VectorVertex
ArcadeDB> CREATE PROPERTY VectorVertex.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX IF NOT EXISTS ON VectorVertex (embedding) LSM_VECTOR METADATA
{dimensions: 1024, similarity: 'COSINE', maxConnections: 16, beamWidth: 100}
```

- Create an LSMVectorIndex with a custom efSearch for consistent high recall:

```
ArcadeDB> CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {dimensions: 768,
similarity: 'COSINE', quantization: 'INT8', efSearch: 200}
```

When `efSearch` is set in the index metadata, all queries on this index use the specified beam width by default. Individual queries can still override it by passing efSearch as the 4th argument to `vector.neighbors()`.

- Create an LSMVectorIndex on embeddings with metadata about source:

```
ArcadeDB> CREATE VERTEX TYPE Doc
ArcadeDB> CREATE PROPERTY Doc.content STRING
ArcadeDB> CREATE PROPERTY Doc.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE PROPERTY Doc.source STRING
ArcadeDB> CREATE INDEX ON Doc (embedding) LSM_VECTOR METADATA {dimensions: 384,
similarity: 'COSINE', maxConnections: 16, beamWidth: 100}
```

Then query for similar documents:

```
ArcadeDB> SELECT expand(vector.neighbors('Doc[embedding]', [0.1, 0.2, 0.3], 10))
```

- Create multiple LSMVectorIndexes on different properties in the same type (multi-modal embeddings):

```
ArcadeDB> CREATE VERTEX TYPE MultiModalRecord
ArcadeDB> CREATE PROPERTY MultiModalRecord.imageEmbedding ARRAY_OF_FLOATS
ArcadeDB> CREATE PROPERTY MultiModalRecord.textEmbedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON MultiModalRecord (imageEmbedding) LSM_VECTOR METADATA
{dimensions: 512, similarity: 'COSINE', maxConnections: 16, beamWidth: 100}
ArcadeDB> CREATE INDEX ON MultiModalRecord (textEmbedding) LSM_VECTOR METADATA
{dimensions: 768, similarity: 'COSINE', maxConnections: 16, beamWidth: 100}
```

Query for similar records using the image embedding:

```
ArcadeDB> SELECT expand(vector.neighbors('MultiModalRecord[imageEmbedding]', [0.1,
0.2, 0.3], 5))
```

- Create an LSMVectorIndex with tuned parameters for high-dimensional data:

```
ArcadeDB> CREATE VERTEX TYPE Word
ArcadeDB> CREATE PROPERTY Word.vector ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON Word (vector) LSM_VECTOR METADATA {dimensions: 100,
similarity: 'COSINE', maxConnections: 16, beamWidth: 100}
```

- Create an LSMVectorIndex with INT8 quantization for 4x memory savings:

```
ArcadeDB> CREATE VERTEX TYPE LargeCorpus
ArcadeDB> CREATE PROPERTY LargeCorpus.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON LargeCorpus (embedding) LSM_VECTOR METADATA {dimensions:
1536, similarity: 'COSINE', quantization: 'INT8'}
```

The index will automatically quantize vectors on insert and dequantize on retrieval, reducing memory usage by 75% with minimal accuracy loss (<2%).

- Create an LSMVectorIndex with BINARY quantization for extreme compression (32x memory savings):

```
ArcadeDB> CREATE VERTEX TYPE MassiveDataset
ArcadeDB> CREATE PROPERTY MassiveDataset.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON MassiveDataset (embedding) LSM_VECTOR METADATA {dimensions:
1536, similarity: 'COSINE', quantization: 'BINARY'}
```

Binary quantization compresses each dimension to 1 bit, achieving 97% memory reduction. Suitable for approximate search with reranking.

- Create an LSMVectorIndex with PRODUCT quantization for zero-disk-I/O graph construction on large datasets:

```
ArcadeDB> CREATE VERTEX TYPE BigDataset
ArcadeDB> CREATE PROPERTY BigDataset.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON BigDataset (embedding) LSM_VECTOR METADATA {dimensions:
1024, similarity: 'COSINE', quantization: 'PRODUCT'}
```

Product quantization builds the graph using compact PQ codes that stay in memory, eliminating disk I/O during construction. Most effective on large datasets (100K+ vectors).

Since 26.11.1 a graph rebuild of a `PRODUCT` index keeps its PQ codebook while it still fits the data and encodes only the vectors added since; it is trained again when the index has grown past `vectorIndex.pqRetrainGrowthFactor` or the new vectors have drifted past `vectorIndex.pqRetrainErrorRatio`. `REBUILD INDEX` always trains a new codebook. See PQ codebook reuse.

Quantization types: `NONE` (default, no compression), `INT8` (recommended, 4x compression, <2% accuracy loss), `BINARY` (32x compression, approximate search), `PRODUCT` (16-64x compression, zero-disk-I/O graph build). Quantization is transparent - vectors are stored compressed but queries work the same way.

- Create an LSMVectorIndex with inline vector storage for large-scale deployments (100K+ vectors):

```
ArcadeDB> CREATE VERTEX TYPE LargeScale
ArcadeDB> CREATE PROPERTY LargeScale.embedding ARRAY_OF_FLOATS
ArcadeDB> CREATE INDEX ON LargeScale (embedding) LSM_VECTOR METADATA {
  dimensions: 1536,
  similarity: 'COSINE',
  quantization: 'INT8',
  storeVectorsInGraph: true,
  mutationsBeforeRebuild: 1000
}
```

The `storeVectorsInGraph` option stores vectors inline within the JVector graph file, eliminating expensive document lookups during search operations. This dramatically reduces RAM pressure for large indexes (100K+ vectors) by avoiding document page cache thrashing.

When to use `storeVectorsInGraph`:

- **Enable** (`true`) for: Large indexes (100K+ vectors), high search throughput, read-heavy workloads, RAM-constrained environments
- **Disable** (`false`, default) for: Small indexes (<10K vectors), frequent vector updates, write-heavy workloads

Trade-offs:

- **Benefit**: 70-80% RAM reduction for large indexes by co-locating vectors with graph topology
- **Cost**: Vectors duplicated on disk (stored in both documents and graph file)
- **Best with**: Combine with `quantization=INT8` or `BINARY` for maximum efficiency (reduced disk + reduced RAM)

Example - optimal large-scale configuration:

```
ArcadeDB> CREATE INDEX ON LargeScale (embedding) LSM_VECTOR METADATA {
  dimensions: 1536,
  similarity: 'COSINE',
  quantization: 'INT8',              -- 4x memory reduction (recommended)
  storeVectorsInGraph: true,         -- Fast co-located access
  efSearch: 200,                     -- Consistent high recall
  maxConnections: 32,                -- Vamana per-layer degree (default); = hnswlib
M=16
  beamWidth: 100,
  locationCacheSize: 200000,         -- Cache hot vectors
  mutationsBeforeRebuild: 1000,      -- Reduce rebuild frequency
  inactivityRebuildTimeoutMs: 30000  -- Flush buffered vectors after 30s of inactivity
}
```

- Create a geospatial index on a WKT property and query it:

```
CREATE VERTEX TYPE Location
```

```
CREATE PROPERTY Location.coords STRING
CREATE INDEX ON Location (coords) GEOSPATIAL

INSERT INTO Location SET name = 'Eiffel Tower', coords = 'POINT(2.2945 48.8584)'

SELECT name FROM Location
WHERE geo.within(coords, geo.geomFromText(
  'POLYGON((2.28 48.84, 2.42 48.84, 2.42 48.90, 2.28 48.90, 2.28 48.84))'
)) = true
```

See how-to/data-modeling/geospatial.pdf for full documentation on geometry formats, predicate functions, and precision configuration.

- Create a sparse vector index for SPLADE / BM25 / BGE-M3 hybrid retrieval.

```
CREATE DOCUMENT TYPE Doc
CREATE PROPERTY Doc.tokens  ARRAY_OF_INTEGERS
CREATE PROPERTY Doc.weights ARRAY_OF_FLOATS

CREATE INDEX ON Doc (tokens, weights) LSM_SPARSE_VECTOR
  METADATA { dimensions: 105000, modifier: 'IDF', weightQuantization: 'FP32' }
```

The two parallel array properties carry the non-zero dimension ids and matching weights for each record. `dimensions` declares the vocabulary cap and rejects out-of-range ids at write time (set to `0` to disable the cap). `modifier: 'IDF'` enables Robertson-Sparck-Jones IDF weighting at query time; omit it (or set to `'NONE'`) for plain dot-product scoring.

`weightQuantization` controls how posting weights are stored on disk, bringing the sparse index to parity with the dense `LSM_VECTOR` index's quantization knob:

Value	Storage / trade-off
`INT8`	1 byte per weight (the default). Smallest index; the stored weights are approximate, the returned scores are exact (see `rescoreOversample` below).
`FP16`	2 bytes per weight. Half the size of `FP32` with near-exact stored weights; the returned scores are exact (see `rescoreOversample` below).
`FP32`	4 bytes per weight. Exact stored weights; largest index.

Omit `weightQuantization` to keep the `INT8` default. The setting is fixed at index-creation time; to change it, drop and recreate (or `REBUILD INDEX`) the index.

`FP16` stores weights as IEEE 754 half-precision, subnormals included, so magnitudes down to `6e-8` keep their value rather than flushing to zero. A weight whose magnitude exceeds the format's largest finite value (`65504`) is saturated to it rather than stored as infinity, and `NaN` is rejected at write time.

`rescoreOversample` controls **exact rescoring**. With `INT8` or `FP16` weights, a search fetches `k * rescoreOversample` candidates from the index, recomputes each one's score from the record's own

full-precision `weights` (with the IDF weighting too, when `modifier: 'IDF'` is set), and returns the best `k` by that exact score. The quantized weights only choose the candidates, so the results and their scores do not depend on the quantization, or on how the data was committed and compacted, which otherwise shifts `INT8` weights a little at every segment merge. This applies to `groupBy` searches as well.

Value	Behavior
omitted	`2` for `INT8` and `FP16`, off for `FP32` (whose stored weights are already exact). On the BigANN sparse track (SPLADE, top-10) this takes `INT8` recall from 0.995 to 1.0, for 10-20% more query time.
`0`	Off: results are ranked by the quantized scores.
`1`	Exact scores for the returned results, without widening the candidate set.
`2` .. `100`	Candidates fetched per requested result.

A sparse-vector index can be created on a type that already holds records: the build indexes the existing rows.

Query with `vector.sparseNeighbors`:

```
SELECT expand(`vector.sparseNeighbors`('Doc[tokens,weights]', :qIdx, :qVal, 50))
```

For server-side hybrid retrieval (dense + sparse + full-text fused into one ranked top-K), see Hybrid Search with `vector.fuse`.

After a large bulk load, run `COMPACT INDEX` to merge the index segments and get lower, more stable query latency without waiting for the background compaction.

For more information, see:

- `COMPACT INDEX`
- `DROP INDEX`
- `Indexes`

14.13.2. SQL - `REBUILD INDEX`

Rebuilds automatic indexes.

Syntax

```
REBUILD INDEX <index-name>
```

- `<index-name>` It is the index name that you want to rebuild. Use `*` to rebuild all automatic indexes. Quote the index name if it contains special characters like square brackets.

During the rebuild, any idempotent queries made against the index, skip the index and perform sequential scans. This means that queries run slower during this operation. Non-idempotent commands, such as `INSERT`, `UPDATE`, and `DELETE` are blocked waiting until the indexes are rebuilt.

Since 26.11.1 `LSM_TREE`, `FULL_TEXT`, `GEOSPATIAL` and `HASH` indexes are rebuilt in place, one bucket at a time, without blocking the database: queries on every type keep running (they still use the old index until each bucket is replaced) and writes wait only while their own bucket is rebuilt. The index keeps its name. If the server stops mid-rebuild, every bucket reopens with its old or its new index, never a partial one. `CHECK DATABASE … FIX` repairs these index kinds the same way. Vector and sparse-vector indexes, HA databases, and a rebuild that changes what a full-text index stores (its similarity was altered) still use the blocking rebuild described above.

Since 26.11.1 an index missing the part of one of its type's buckets (for example after a rebuild interrupted by an older version) is reported with `valid = false`, queries do not use it and `COMPACT INDEX` refuses it. The server logs which buckets are missing at startup; `REBUILD INDEX <index-name>` fills them without blocking the database. Index files left behind by an interrupted rebuild are deleted at the next startup, with one warning each. Since 26.12.1 no query planner uses such an index (SQL, Cypher, the Java `select()` API, the MongoDB wire protocol): the query falls back to a scan and answers every row, where some of them used to return short results from it. `database.lookupByKey()` on it raises an `IllegalArgumentException` naming the index and asking for `REBUILD INDEX`, and Cypher `SHOW INDEXES` reports it with state `FAILED` (issue #11303).

An index declared on a super type also covers the buckets of its sub types. `REBUILD INDEX *` (and `CHECK DATABASE … FIX`) rebuilds the part on a sub type's bucket under the index the super type declares. Since 26.11.1; before, the rebuild could detach the super type's own buckets from the index.

During normal operations an index rebuild is not necessary. Rebuild an index only if it breaks.

If the rebuild fails (for example because a stored record has a null key and the index null strategy is `ERROR`), the statement reports the error and the index is put back as it was, with its unique constraint. If the original null strategy cannot be applied to the stored data, the index is restored with `SKIP` and the original strategy is set in memory only, so after a restart it uses `SKIP` again.

To build a large `LSM_TREE` index (unique or non-unique) over data that is already loaded (single-server, no active transaction), the Java API offers a much faster offline **sorted bulk build** that constructs the index directly in compacted form. See Sorted bulk build.

Examples

- Rebuild an index on the `email` property on the type `Profile`:

```
ArcadeDB> REBUILD INDEX `Profile[email]`
```

- Rebuild all indexes:

```
ArcadeDB> REBUILD INDEX *
```

For more information, see:

- `CREATE INDEX`
- `COMPACT INDEX`
- `DROP INDEX`
- Indexes

14.13.3. SQL - `COMPACT INDEX`

Forces an immediate compaction of automatic indexes, merging their on-disk segments into a compacted form.

ArcadeDB compacts indexes automatically in the background. `COMPACT INDEX` lets you trigger it on demand, which is useful right after a bulk load to settle an index and get lower, more stable query latency without waiting for the background schedule. Unlike `REBUILD INDEX`, it does not re-index the data: it only merges the existing segments, so it is much faster.

It works for standard LSM-Tree indexes as well as vector and sparse-vector indexes, and runs synchronously (it returns when the compaction has finished). On a sparse-vector index it also seals the in-memory buffer of recent writes, so afterwards the whole index sits in a single segment. Because it is a regular SQL command, it can be executed from any client, including the HTTP `/command` endpoint.

Syntax

```
COMPACT INDEX <index-name>
```

- `<index-name>` It is the index name that you want to compact. Use `*` to compact all automatic indexes. Quote the index name if it contains special characters like square brackets.

Examples

- Compact the sparse-vector index on the `tokens` and `weights` properties of the type `Doc`:

```
ArcadeDB> COMPACT INDEX `Doc[tokens,weights]`
```

- Compact all indexes:

```
ArcadeDB> COMPACT INDEX *
```

For more information, see:

- `CREATE INDEX`
- `REBUILD INDEX`
- Indexes

14.13.4. SQL - `DROP INDEX`

Removes an index from a property defined in the schema.

If the index does not exist, this call just returns with no errors.

Syntax

```
DROP INDEX <index-name> [ IF EXISTS ]
```

- `<index-name>` Defines the name of the index. Use `*` to drop all the indexes of the database.

An index on a type with several buckets is one file per bucket. The statement checks that every one of them can be deleted, and takes their locks, before it drops the first: if one cannot be deleted (a file that is not a regular file, a directory that cannot be written) it is refused with `Nothing was changed` and the index stays whole. The same holds for `DROP PROPERTY … FORCE`, `TRUNCATE TYPE`, `REBUILD INDEX` and the other statements that drop indexes, which drop all of theirs or none. Since 26.11.1; before, the files before the refusing one were already dropped and the index answered 0 rows for the records of their buckets, even after a restart.

`DROP INDEX *` is one statement too: it checks the files of every index before it drops the first, so when one cannot be deleted no index is dropped. It returns one row per index dropped (the per-bucket indexes included), with the name in `indexName` (the same name is also in `bucketName`, kept for compatibility). Since 26.11.1; before, it dropped the indexes one at a time, so a refusal left the indexes before it dropped, and the rows carried the name in `bucketName` only.

An index on a type is made of one bucket index per bucket of the type (for example `S_0_4559193709411875`, the part of `S[k]` on one bucket). `DROP INDEX` naming one of those bucket indexes, from SQL or Cypher, is refused with a `SchemaException` (`Index 'S_0_4559193709411875' is the bucket index of index 'S[k]' and cannot be dropped alone: … Drop 'S[k]' instead`), since the type index would miss every record of that bucket. Drop the type index itself, which drops all its bucket indexes. *(Since v26.12.1, issue #11303. Before, the bucket index was dropped and the type index kept answering without the records of that bucket; an index left that way is repaired with* `REBUILD INDEX`*.)*

A write running while the index it reads is dropped fails with a retryable `ConcurrentModificationException` (HTTP `503`), and the retry succeeds without the index *(since v26.12.1, see Writes during an index change)*.

Examples

- Remove the index on the `Id` property of the `Users` type:

```
ArcadeDB> DROP INDEX `Users[Id]`
```

For more information, see:

- `CREATE INDEX`
- Indexes

14.14. Buckets

14.14.1. SQL - `CREATE BUCKET`

Creates a new bucket in the database. Once created, you can use the bucket to save records by specifying its name during saves. If you want to add the new bucket to a type, follow its creation with the `ALTER TYPE` command.

Syntax

```
CREATE BUCKET <bucket> [IF NOT EXISTS] [ID <bucket-id>]
```

- `<bucket>` Defines the name of the bucket you want to create. You must use a letter for the first character, for all other characters, you can use alphanumeric characters, underscores and dashes.
- **IF NOT EXISTS** Specifying this option, the bucket creation is just ignored if the bucket already exists, instead of failing with an error. The statement returns one row either way, carrying a `created` flag: see the note on `IF NOT EXISTS` results.
- `<bucket-id>` Defines the numeric ID you want to use for the bucket.

Examples

- Create the bucket `account`:

```
ArcadeDB> CREATE BUCKET account
```

> For more information see:

- `DROP BUCKET`

14.14.2. SQL - DROP BUCKET

Removes the bucket and all of its content. This operation is permanent and cannot be rolled back.

Syntax

```
DROP BUCKET <bucket-name>|<bucket-id> [IF EXISTS]
```

- `<bucket-name>` Defines the name of the bucket you want to remove.
- `<bucket-id>` Defines the ID of the bucket you want to remove.
- `IF EXISTS` Prevent errors if the bucket does not exist when attempting to drop it. See here for the shape of the result.

A bucket that belongs to a type cannot be dropped: the statement is refused (HTTP `400`) and the bucket keeps its records. Detach it first with `ALTER TYPE … BUCKET -`, or drop the whole type with `DROP TYPE`. *(Since v26.12.1, issue #12436. Before, the same refusal was answered with `500`.)*

Examples

- Remove the bucket `Account`:

```
ArcadeDB> DROP BUCKET Account
```

A bucket cannot be renamed or altered (`ALTER BUCKET` was removed in v26.12.1). To move the records of a bucket to another one, create the new bucket, copy the records with `MOVE VERTEX` or `INSERT … SELECT`, then drop the old bucket. A copied record gets a new RID: rewrite the links that point to the old RIDs before `DROP BUCKET`, which deletes the source records.

For more information, see:

- `CREATE BUCKET`
- `DROP TYPE`

14.14.3. SQL - TRUNCATE BUCKET

Deletes all records of a bucket. This command operates at a lower level than the standard `DELETE` command.

Truncation is not permitted on vertex or edge types, but you can force its execution using the `UNSAFE` keyword. Forcing truncation is strongly discouraged, as it can leave the graph in an inconsistent state.

Syntax

```
TRUNCATE BUCKET <bucket>
```

- `<bucket>` Defines the bucket to delete.
- `UNSAFE` Defines whether the command forces the truncation on vertex or edge types.

Transactions

As with `TRUNCATE TYPE`, the command is part of the caller's transaction when one is active - a `ROLLBACK` puts every record back - and owns a transaction of its own, committing one batch of `arcadedb.truncateBatchSize` records at a time, when none is. The result set reports which applied through the `transactional` field.

Examples

- Remove all records in the bucket `profile`:

```
ArcadeDB> TRUNCATE BUCKET profile
```

For more information, see:

- `DELETE`
- `TRUNCATE TYPE`

14.15. Triggers (DDL)

14.15.1. SQL - `CREATE TRIGGER`

Creates a new database trigger that automatically executes SQL statements, JavaScript code, or Java classes in response to record events.

Triggers enable you to implement business logic, data validation, audit trails, and automated workflows directly in your database.

Syntax

```
CREATE TRIGGER [IF NOT EXISTS] <trigger-name>
  (BEFORE|AFTER) (CREATE|READ|UPDATE|DELETE)
  ON [TYPE] <type-name>
  EXECUTE (SQL|JAVASCRIPT|JAVA) '<code-or-class-name>'
```

- `IF NOT EXISTS` Optional: skip creation if trigger already exists (instead of failing with an error)
- `<trigger-name>` Unique name for the trigger
- `BEFORE|AFTER` When to execute the trigger:
  - `BEFORE` - Execute before the event occurs

    - `AFTER` - Execute after the event occurs
- `CREATE|READ|UPDATE|DELETE` Which database event to respond to:
    - `CREATE` - When a new record is created
    - `READ` - When a record is read from the database
    - `UPDATE` - When a record is modified
    - `DELETE` - When a record is deleted
- `<type-name>` The document, vertex, or edge type to monitor. The trigger also fires for the records of its subtypes. It can be one of the type's aliases: the trigger is bound to the type itself, so it is listed with the type, follows a rename, survives the removal of the alias and is dropped with the type. *(Since v26.11.1)* In 26.10.2 and earlier a trigger stored the alias, and outlived a `DROP TYPE` to bind itself to a type created later under that name (issue #10750)
- `SQL|JAVASCRIPT|JAVA` The language/type of the trigger action:
    - `SQL` - Execute SQL statements
    - `JAVASCRIPT` - Execute JavaScript code using GraalVM
    - `JAVA` - Execute a Java class implementing the JavaTrigger interface
- `<code-or-class-name>` The SQL statement, JavaScript code, or fully qualified Java class name to execute

Context Variables

Triggers have access to the following context variables:

- **SQL Triggers**:
    - `$record` or `record` - The current record being operated on
- **JavaScript Triggers**:
    - `record` or `$record` - The current record being operated on
    - `oldRecord` or `$oldRecord` - The original record (for UPDATE operations only)
    - `database` - The database instance
- **Java Triggers**:
    - Parameters passed to the `execute()` method

Return Values

- **SQL Triggers**: If the SELECT statement returns no results, the trigger fails and the operation is aborted
- **JavaScript Triggers**: Return `false` to abort the operation (BEFORE triggers only), or throw an exception
- **Java Triggers**: Return `false` to abort the operation (BEFORE triggers only), or throw an exception

Examples

- Create an audit log that tracks all user creations:

```
CREATE DOCUMENT TYPE AuditLog;

CREATE TRIGGER user_audit AFTER CREATE ON TYPE User
EXECUTE SQL 'INSERT INTO AuditLog SET action = "user_created", userName =
$record.name, timestamp = sysdate()';
```

- Validate email format before creating users using JavaScript:

```
CREATE TRIGGER validate_email BEFORE CREATE ON TYPE User
EXECUTE JAVASCRIPT 'if (!record.email || !record.email.includes("@")) {\nthrow new
Error("Invalid email address");\n}';
```

- Automatically set timestamps using JavaScript:

```
CREATE TRIGGER user_defaults BEFORE CREATE ON TYPE User
EXECUTE JAVASCRIPT 'record.createdAt = new Date();\nif (!record.username &&
record.email) {\nrecord.username = record.email.split("@")[0];\n}';
```

- Ensure product prices are positive using SQL validation:

```
CREATE TRIGGER validate_price BEFORE CREATE ON TYPE Product
EXECUTE SQL 'SELECT FROM Product WHERE @this = $record AND price > 0';
```

- Update related records when parent changes:

```
CREATE TRIGGER customer_email_update AFTER UPDATE ON TYPE Customer
EXECUTE SQL 'UPDATE Order SET customerEmail = $record.email WHERE customerId =
$record.@rid';
```

- Use a Java class for maximum performance (class must be in the classpath):

```
CREATE TRIGGER validate_email BEFORE CREATE ON TYPE User
EXECUTE JAVA 'com.example.triggers.EmailValidationTrigger';
```

Java Trigger Example

To create a Java trigger, implement the `com.arcadedb.schema.trigger.JavaTrigger` interface:

```
package com.example.triggers;

import com.arcadedb.database.Database;
```

```
import com.arcadedb.database.Document;
import com.arcadedb.database.Record;
import com.arcadedb.schema.trigger.JavaTrigger;

public class EmailValidationTrigger implements JavaTrigger {

  @Override
  public boolean execute(Database database, Record record, Record oldRecord) throws
Exception {
    if (record instanceof Document) {
      Document doc = (Document) record;
      String email = doc.getString("email");

      if (email == null || !email.contains("@")) {
        throw new IllegalArgumentException("Invalid email address");
      }
    }
    return true;  // Continue with the operation
  }
}
```

For more information and advanced examples, see Triggers.

14.15.2. Listing triggers

Query the read-only `schema:triggers` virtual type to see the triggers a database defines. It works the same way locally and against a remote server.

```
SELECT FROM schema:triggers
```

Each row carries `name`, `typeName`, `timing` (`BEFORE`/`AFTER`), `event` (`CREATE`/`READ`/`UPDATE`/`DELETE`), `actionType` (`SQL`/`JAVASCRIPT`/`JAVA`) and `actionCode`. Filter and project it like any other type:

```
SELECT name, timing, event FROM schema:triggers WHERE typeName = 'User'
```

Triggers defined on a type the current user cannot read are omitted rather than failing the query.

Available. See System Types for the other `schema:*` virtual types.

14.15.3. SQL - DROP TRIGGER

Removes a trigger from the database.

Syntax

```
DROP TRIGGER [IF EXISTS] <trigger-name>
```

- `IF EXISTS` Optional: do not fail if the trigger does not exist
- `<trigger-name>` The name of the trigger to remove

Examples

- Drop a trigger:

```
DROP TRIGGER user_audit;
```

- Drop a trigger only if it exists:

```
DROP TRIGGER IF EXISTS old_trigger;
```

For more information about triggers, see Triggers.

14.16. SQL Materialized Views

14.16.1. SQL - `CREATE MATERIALIZED VIEW`

Creates a new materialized view in the schema. A materialized view stores the result of a SQL `SELECT` query in a backing document type for fast reads. The defining query is executed immediately on creation to perform the initial data load.

Syntax

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] <name>
  AS <select-query>
  [REFRESH MANUAL | REFRESH INCREMENTAL | REFRESH EVERY <n> SECOND | MINUTE | HOUR]
  [BUCKETS <n>]
```

- `<name>` Defines the name of the materialized view. It must not conflict with any existing type or materialized view.
- `IF NOT EXISTS` Silently skips creation if a view with the same name already exists, instead of raising an error. See here for the shape of the result.
- `<select-query>` A SQL `SELECT` statement that defines the data for the view. All source types referenced, including those in subqueries, must exist at creation time.
- `REFRESH MANUAL` (default) The view is only refreshed when triggered explicitly.
- `REFRESH INCREMENTAL` The view is automatically refreshed after each committed transaction that modifies a source type. Every type the query names is a source type, at any depth (top-level `FROM`, subqueries in `FROM`, `WHERE`, projection or `LET`), together with all its subtypes. Refused with an error when the query reads a source that cannot be tracked: a `TRAVERSE`, a `MATCH`, a function call, a bucket, an index, `schema:` metadata, record ids or parameters in `FROM`. See INCREMENTAL.
- `REFRESH EVERY <n> SECOND | MINUTE | HOUR` The view is refreshed on a schedule at the given

interval.

- `BUCKETS <n>` Sets the number of buckets for the backing document type.

Examples

- Create a simple manual-refresh view:

```
CREATE MATERIALIZED VIEW ActiveUsers
  AS SELECT name, email FROM User WHERE active = true
```

- Create a view that updates automatically after each relevant commit:

```
CREATE MATERIALIZED VIEW RecentOrders
  AS SELECT customer, amount FROM Order WHERE status = 'PENDING'
  REFRESH INCREMENTAL
```

- Create a view that refreshes on a schedule:

```
CREATE MATERIALIZED VIEW HourlySummary
  AS SELECT product, sum(amount) AS total FROM Sale GROUP BY product
  REFRESH EVERY 1 HOUR
```

- Skip creation if the view already exists:

```
CREATE MATERIALIZED VIEW IF NOT EXISTS ActiveUsers
  AS SELECT name FROM User WHERE active = true
```

- Create a view with a custom bucket count for large result sets:

```
CREATE MATERIALIZED VIEW BigView
  AS SELECT * FROM HugeType
  BUCKETS 8
```

For more information, see:

- Materialized Views
- `DROP MATERIALIZED VIEW`
- `REFRESH MATERIALIZED VIEW`
- `ALTER MATERIALIZED VIEW`

14.16.2. SQL - `ALTER MATERIALIZED VIEW`

Changes the refresh mode of an existing materialized view.

Syntax

```
ALTER MATERIALIZED VIEW <name>
  REFRESH MANUAL | REFRESH INCREMENTAL | REFRESH EVERY <n> SECOND | MINUTE | HOUR
```

- `<name>` The name of the materialized view to alter.
- `REFRESH MANUAL` Switches the view to manual refresh mode.
- `REFRESH INCREMENTAL` Switches the view to incremental refresh mode (automatic after each relevant commit). Refused with an error when the view's query reads a source that cannot be tracked, as on `CREATE MATERIALIZED VIEW`.
- `REFRESH EVERY <n> SECOND | MINUTE | HOUR` Switches the view to periodic refresh mode at the specified interval.

For more information, see:

- Materialized Views
- `CREATE MATERIALIZED VIEW`
- `REFRESH MATERIALIZED VIEW`

14.16.3. SQL - `DROP MATERIALIZED VIEW`

Removes a materialized view and its backing document type, including all stored data.

Syntax

```
DROP MATERIALIZED VIEW [IF EXISTS] <name>
```

- `<name>` The name of the materialized view to remove.
- `IF EXISTS` Suppresses errors when the view does not exist.

Examples

- Drop a materialized view:

```
DROP MATERIALIZED VIEW ActiveUsers
```

- Drop a view only if it exists:

```
DROP MATERIALIZED VIEW IF EXISTS NonExistentView
```

You cannot drop the backing document type of a materialized view directly using `DROP TYPE`, nor any type its query reads (listed in `sourceTypes` of `schema:materializedViews`). Drop the materialized view first.

The backing type is written by the refresh of the view only: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE TYPE`, `TRUNCATE BUCKET` and the removal of one of its buckets are refused on it, naming `REFRESH MATERIALIZED VIEW` and `DROP MATERIALIZED VIEW`. *(Since v26.11.1, issue #11020.)* No type can extend it either: `CREATE ⋯ TYPE ⋯ EXTENDS <view>` and `ALTER TYPE ⋯ SUPERTYPE +<view>` are refused, since the view would answer the subtype's rows as its own. *(Since v26.11.1, issue #11204.)*

For more information, see:

- Materialized Views
- `CREATE MATERIALIZED VIEW`

14.16.4. SQL - `REFRESH MATERIALIZED VIEW`

Triggers an immediate full refresh of a materialized view's data. The operation truncates all existing data in the backing type and reloads it by re-executing the defining query. The refresh runs inside its own transaction.

This command is useful for `MANUAL` refresh mode views and for recovering a `STALE` view after a failed automatic refresh.

Syntax

```
REFRESH MATERIALIZED VIEW <name>
```

- `<name>` The name of the materialized view to refresh.

Examples

- Refresh a view immediately:

```
REFRESH MATERIALIZED VIEW HourlySummary
```

- Refresh after a bulk import to bring a manual view up to date:

```
REFRESH MATERIALIZED VIEW SalesByProduct
```

For more information, see:

- Materialized Views
- `CREATE MATERIALIZED VIEW`
- `ALTER MATERIALIZED VIEW`

14.17. Database Admin

14.17.1. SQL - `ALIGN DATABASE`

Reserved for a distributed alignment of the database's replicas.

(Since v26.12.1.) No database implements it, and the statement is refused with an error that says so (HTTP 400, it used to answer 500 with a bare `UnsupportedOperationException`). With Raft replication a replica is realigned by the cluster itself, by replaying the leader's log or by installing a snapshot of the leader, and a database with no replicas has nothing to align. To inspect the copy of a node, run CHECK DATABASE on that node.

Syntax

```
ALIGN DATABASE
```

The command returns which page have been aligned on each server.

Examples

- Align the current database.

```
ArcadeDB> ALIGN DATABASE
```

14.17.2. SQL - `ALTER DATABASE`

Change a database setting. You can find the available settings in Settings appendix. The update is persistent.

Syntax

```
ALTER DATABASE <setting-name> <setting-value>
```

- `<setting-name>` Check the available settings in Settings appendix. Since the setting name contains `.` characters, surround the setting name with `` ` ``.
- `<setting-value>` The new value to set

Only **database** settings can be set this way. A setting that belongs to the whole server or to the JVM is refused, naming its scope: use `SET SERVER SETTING` for a server setting, or a `-D` JVM property for a JVM one.

The value reported back is the one that took effect, which is not always the one you sent: a setting with a valid range is corrected to fit. For example `arcadedb.maxPageRAM` is reduced when the page cache asked for is larger than the

heap can accommodate.

On an HA cluster the changed setting is replicated to every node, which applies exactly that key. A node running an older build cannot apply it, so during a rolling upgrade the statement is refused with HTTP `409` and the setting is put back. See Database settings for path-valued settings and for what to do when the statement times out. *(Since v26.10.2: the setting was written on the node that ran the statement only.)*

Examples

- Set the date time format to support milliseconds (the default is 'yyyy-MM-dd HH:mm:ss').

```
ArcadeDB> ALTER DATABASE `arcadedb.dateTimeFormat` 'yyyy-MM-dd HH:mm:ss.SSS'
```

- Set the default page size for buckets to 262,144 bytes. This is useful when importing database with records bigger than the default page.

```
ArcadeDB> ALTER DATABASE `arcadedb.bucketDefaultPageSize` 262144
```

14.17.3. SQL - `BACKUP DATABASE`

Executes a backup of the current database. The resulting file is a compressed archive using ZIP as algorithm. The archive contains the database directory without the transaction logs. The backup is executed taking a snapshot of the database at the time the command is executed. Any pending transaction will not be in the backup archive. *(Since v26.12.1)* a backup taken inside a transaction leaves that transaction as it was, and its writes are committed with it: it used to be rolled back before the backup. ArcadeDB allows to execute a non-stop backup of a database while it is used without blocking writes or affecting performance. A close requested while a backup is running waits for the backup to complete, and the database keeps serving in the meantime: the archive is always complete, and the close goes ahead right after.

One database runs one whole-database maintenance operation at a time: backup, restore, import, export and `drop database`. A second one on the same database is refused immediately with an `operation in progress` error naming the one already running, rather than proceeding alongside it - simply retry once the first has finished. Two exceptions: several exports of one database may run together (they read only), and a `drop database` conflicts with everything, including another drop.

Previously a `drop database` issued during a backup, restore or import could delete the database directory while that operation was still using it.

Syntax

```
BACKUP DATABASE [ <backup-file-url> ]
```

- `<backup-file-url>` Optional, defines the name of the backup archive. If not specified, the backup file will be `backups/<db-name>/<db-name>-backup-<timestamp>.zip`, where the timestamp is expressed from the year to the millisecond. Example of backup file name `backups/TheMatrix/TheMatrix-backup-20210921-172750767.zip`.

`<backup-file-url>` must be a bare file name (e.g. `mybackup.zip`), never a path or a `file://` URL that carries one. `BACKUP DATABASE` always writes into the server's backup directory so the archive's location cannot be steered elsewhere through SQL; a value with a directory component - `/`, `\`, or `..` in it - is rejected with `Backup file cannot contain path change because the directory is specified`. Use the standalone `Backup` class or the embedded API directly (not through SQL) if a full path is genuinely needed.

The backup directory is the one every other backup command uses - the running auto-backup plugin's configuration, then `config/backup.json` even when the schedule is off, then `arcadedb.server.backupDirectory` - so an archive written here is visible to `list backups` and reachable by `delete backup` and `restore backup`. See Where an On-Demand Backup Goes.

Previously the statement read `arcadedb.server.backupDirectory` on its own, so with a `config/backup.json` naming a different directory the archive landed out of reach of those commands and was never retention-pruned. One consequence of the change: `BACKUP DATABASE` now fails if that configured directory is invalid (absolute, or escaping the server root), which is the same refusal the other backup commands already gave.

Since 26.10.2 the statement applies the retention policy after a successful backup, as `trigger backup` and the auto-backup scheduler do. Retention works off the archives in `backups/<db-name>` rather than off the databases registered with the scheduler, so a database that is not named in `config/backup.json` is pruned by the `defaults` retention (`maxFiles`, or the tiered policy) instead of growing its directory without bound. The archive just written is always kept.

Only archives named `<db-name>-backup-<timestamp>.zip` are subject to retention. An archive you name yourself with `BACKUP DATABASE 'before-the-upgrade.zip'` is never deleted by it. With the auto-backup scheduler disabled nothing is pruned, which is the behavior of an on-demand backup without a scheduler.

Examples

- Execute the backup of the current database with the default filename.

```
ArcadeDB> BACKUP DATABASE
```

14.17.4. SQL - `CHECK DATABASE`

Executes an integrity check and in case of a repair of the database. This command analyzes the

following things:

- buckets: all the pages and records are scanned and checked if can be loaded (no physical corruption)
- vertices: all the vertices are loaded and all the connected edges are checked. In case some edges point to records that have been deleted they can be fixed automatically if the `FIX` option is enabled.
- edges: scan all the edges and check the incoming and outgoing links are consistent in the relative vertices. If not, the edges can be automatically removed if the `FIX` option is enabled.
- TimeSeries types: every shard's mutable bucket, sealed store and shared tag dictionary is validated, including the CRC32 of every sealed block. See TimeSeries integrity below.
- indexes: the structure of each index is validated. A corrupted index is dropped and rebuilt if the `FIX` option is enabled, keeping its name and its configuration - a full-text index's analyzers and BM25 parameters, a geospatial index's precision, a vector index's dimensions. *Since v26.11.1*, an `LSM_TREE`, `FULL_TEXT`, `GEOSPATIAL` or `HASH` index is instead rebuilt in place without blocking the database (see `REBUILD INDEX`), and an index missing the part of some of its type's buckets has only those parts created and filled.
- existence constraints: every record of a type that declares a `MANDATORY` or `NOTNULL` property is checked against it. See Existence constraints below.
- cached record counters: the counter `count(*)` answers from is compared against the records the bucket scan actually found. See Cached record counters below.

> *(Since v26.12.1.)* Inside a transaction (`BEGIN … COMMIT` in a script, `database.transaction(…)`, an HTTP session), `CHECK DATABASE`, with or without `FIX` and `COMPRESS`, runs outside that transaction and leaves it as it was: it checks the committed database, and the writes made in the transaction before and after it are committed with it. It used to roll the transaction back, losing those writes, and the next statement failed with `Transaction not begun`.

- index entries against records: *since v26.12.1*, the entries of every `LSM_TREE` index are compared with the records of its buckets. An entry whose record does not exist (`danglingIndexEntries`), an entry whose record does not hold its key (`staleIndexEntries`) and a key a record holds with no entry (`missingIndexEntries`) are reported, the index is listed under `corruptedIndexes`, `warnings` names the entries and records involved, and `FIX` rebuilds it. Such entries make an indexed query return a row whose value differs from the one asked for (a dangling entry names whatever record reuses its RID later) or miss a row. A healthy index costs one scan of its buckets, shared by all their indexes, and one walk of its entries.
- a compacted sub-index the index could not load: *since v26.12.1*, an `LSM_TREE` index whose compacted file was missing when the database opened (the server log says `Invalid sub-index … File with id N was not found`) is listed under `corruptedIndexes`, and `warnings` says that the entries of that file are not served. Until then such an index answered from its newest entries alone, and a `UNIQUE` one did not refuse a duplicate of a key only the lost file held, while the check reported it healthy. `FIX` rebuilds it, and so does `REBUILD INDEX`. The index is not compacted meanwhile, because a compaction would write the partial index out as the whole one. On a cluster node, a `UNIQUE` index in this state also quarantines the database, which is then re-

installed from the cluster, and the node hands leadership off rather than lead with it.

`CHECK DATABASE` on a cluster

Since v26.12.1, `CHECK DATABASE` without `FIX` (and without `COMPRESS`) runs on the node it is sent to, and reports that node's files. Before, a replica forwarded it to the leader like any write, so it reported the **leader's** copy: a replica whose record counter or index had drifted was reported clean, which is the question the check exists to answer. `FIX` and `COMPRESS` are still forwarded to the leader, so their repairs replicate to every node.

`HASH`, `FULL_TEXT`, vector, geospatial and columnar indexes get the **structural** check only: it validates the index's own pages, not that every record of the type appears in the index. If you suspect one of them is incomplete - for example its build was interrupted by a crash - run `REBUILD INDEX` on it: rebuilding re-reads every record and also preserves the index configuration.

When a vertex's edge list is unreadable (for example a lost or corrupted chunk of the adjacency linked list), the `FIX` mode does not delete anything: every edge record stores its own endpoint identities, so the adjacency list is rebuilt from the surviving edge records. After the rebuild, a full-scope `FIX` (no `TYPE`/`BUCKET` filter) also reclaims the orphaned edge-list segments the old, unreachable chains left on disk.

The reverse shape is an **orphan edge record**: an edge record whose `@out` and `@in` are live vertices, but that no vertex's edge list references, so no traversal reaches it. It is reported as `ORPHAN RECORD` and counted under `unreachableEdgeRecords`. *Since v26.11.1* a plain `FIX` relinks it: the edge goes back into the outgoing list of its `@out` vertex and, for a bidirectional edge type, into the incoming list of its `@in` vertex. The relinked records are counted under `relinkedOrphanEdgeRecords`, and the adjacency entries written under `reconnectedEdges`. `FIX DELETE ORPHANS` deletes them instead, and takes precedence when given. Without `FIX` the warning suggests both: `CHECK DATABASE FIX` to relink the edge, or `CHECK DATABASE FIX DELETE ORPHANS` to remove it. Before v26.11.1 a plain `FIX` left orphan edge records untouched, and `DELETE ORPHANS` was the only repair. One run relinks a bounded number of orphans to cap its memory; what it leaves is counted under `orphanEdgeRecordsLeftToRelink`, and the next `FIX` relinks it. A `GraphBatch` flush that failed inside a live process deletes the records it wrote, and logs the RID of any it could not delete: delete those before a `FIX`, which would otherwise relink edges the caller was told were not created.

The usual cause is a GraphBatch whose process died (`kill -9`, an OOM kill, a power loss) before `close()`: the edge records its flushes committed are on disk, but the vertices' head-chunk pointers are only written at `close()`. `CHECK DATABASE FIX` makes those committed edges traversable without reloading them.

A lightweight edge has no record to be rebuilt from. *Since v26.11.1* the rebuild carries the lightweight entries of the part of the dropped list it can still read over into the new one (`salvagedLightEdges`), and rebuilds the lightweight edges of a bidirectional type stored in the unreadable part from their copy in the other endpoint's list (`recoveredLightEdges`), as many times as that list holds them. That costs one more walk of the edge lists of every vertex, only in `FIX` mode and only when a list is rebuilt. A lightweight edge of a unidirectional type is stored in its source's outgoing list only: one stored in the unreadable part has no other copy and is lost, and the run counts the list under `unsalvageableEdgeLists` with a warning naming the vertex and the direction.

Before v26.11.1 a rebuild dropped every lightweight edge of the list.

The other shape is a **dangling reference**: an adjacency entry that points at a record which no longer exists, typically left behind by a vertex deleted by an old release, before edge removal became strict. There is nothing to rebuild in that case - the record the entry names is gone - so `FIX` drops the entry instead. *Since v26.12.1* this covers lightweight edge entries too: one whose other vertex is gone, as left by `DROP TYPE … UNSAFE` on a vertex type, is reported in `warnings` and counted under `invalidLinks` and `topMissingReferences`, and `FIX` drops it (counted under `prunedDanglingEntries`). Before, the check saw only record-backed edges, so `FIX` removed those and left the lightweight entries, which SQL `outE()`/`inE()` and `count(*)` kept returning.

An application that reaches such a record before the repair has run is told so plainly, and always in the same way. Removing an entry from the edge list of a vertex whose own record is missing, adding one to it, and deleting that vertex outright all raise `VertexNotFoundException`, a `RecordNotFoundException` (answered as HTTP `404`) that names the vertex and points at `CHECK DATABASE FIX`. Creating an edge whose **endpoint** is such a record raises it too, naming the endpoint - which is not the vertex the statement named. It is deliberately **not** a `NeedRetryException`: the record is gone rather than temporarily unreadable, so a retry can only fail identically, and a job walking possibly-stale references can skip that entry instead of spending its retry budget and rolling back the whole transaction. An unreadable **chunk** of a vertex that does exist keeps raising the retryable `ConcurrentModificationException` described above - the two are told apart by which record the failure names.

A `DELETE VERTEX` refused this way is refused before it touches anything: the vertex's record is checked first, so the failure does not cost a walk over an adjacency list that is about to be rolled back.

the same answer now covers the genuinely concurrent case. A vertex that **another** transaction deletes while this one is removing its edges used to be reported as a retryable `ConcurrentModificationException`; it is a `VertexNotFoundException` like every other "the vertex is not there", because the record is equally gone either way and a retry could only rediscover that on the next attempt. An application that wraps its vertex deletes in its own `catch (ConcurrentModificationException)` retry loop no longer catches that case: catch `RecordNotFoundException` alongside the conflict and treat it as already deleted, which is the conclusion the loop would have reached after spending its retries.

That up-front check reads the vertex's record, so deleting a vertex needs the `readRecord` permission on its type as well as `deleteRecord` (see groups). This is not a new requirement in practice, since every way of obtaining the vertex in order to delete it already reads it, but a group granting `deleteRecord` alone now fails at the first statement of the delete rather than part of the way through it.

the `RECORD` scope cannot repair a dangling reference. The RID in the message is the record that is missing, so there is no edge list to rebuild there, and the entry to drop lives on the vertex at the **other** end. No index maps a vertex back to the lists that reference it, so finding that vertex takes a scan - which is why the message names the database-wide form.

While the check runs, its progress (current step and percentage) is visible live in the console, in Studio, and through the HTTP progress endpoint. Run FIX in a maintenance window: like every repair, it assumes no concurrent writers.

(Since v26.12.1.) CHECK DATABASE with FIX or COMPRESS requires the updateSchema database permission, like REBUILD INDEX and COMPACT INDEX (see groups), because a repair deletes and rewrites records whatever the user's per-type grants. A user without it, such as one of the built-in readonly and crud groups, gets a security error (HTTP 403) and the database is left untouched. CHECK DATABASE without FIX and COMPRESS only reports and stays allowed. Before, any user who could open the database could run a repair.

Syntax

```
CHECK DATABASE [ TYPE <type-name>[,]* ] [ BUCKET <bucket-name>[,]* ] [ RECORD <rid>
[,]* ]
              [ FIX ] [ DELETE ORPHANS ] [ DELETE INVALID RECORDS ] [ RECLAIM
UNREFERENCED FILES ]
              [ DEEP ] [ COMPRESS ]
```

The optional clauses must appear in the order shown above.

- <type-name> Optional, if specified limit the check (and the fix) only to the specific types
- <bucket-name> Optional, if specified limit the check (and the fix) only to the specific buckets
- <rid> Optional, limits the check to the listed records instead of scanning whole types. Cannot be combined with TYPE or BUCKET. Useful when a single vertex's edge list is broken and a full scan is not worth its cost. The listed records must exist: repairing a reference that points at a **deleted** record is a database-wide FIX, not this scope
- FIX Optional, if used auto fix the issue found with the check
- DELETE ORPHANS Optional, requires FIX. Removes orphan edge records: edge records whose endpoints are valid vertices but which no vertex's edge list references. A clause of its own because the same shape is also produced by a vertex that merely lost its head-chunk pointer, whose edges are still recoverable. Without it a FIX relinks those records into their vertices' edge lists instead *(since v26.11.1; before, it left them untouched)*, and without FIX the finding is only reported, under unreachableEdgeRecords
- DELETE INVALID RECORDS Optional, requires FIX,. Removes the records that do not satisfy their own type's existence constraints. A clause of its own because ALTER PROPERTY ⋯ MANDATORY TRUE on a type that already holds records makes every one of them a finding while they hold real data - see Existence constraints. Without it the finding is still reported, under constraintViolatingRecords
- RECLAIM UNREFERENCED FILES Optional, requires FIX. Deletes files this node holds that no schema component was ever built for. Without it the finding is still reported, under unreferencedFiles
- DEEP Optional, adds the checks that decode the stored data rather than reconciling what describes it. Today only the TimeSeries pass has a deeper tier - see TimeSeries integrity. Independent of FIX: nothing this tier finds is repairable

- `COMPRESS` Optional, if used compresses the database with the check, packing every page to reclaim the space a deleted or shrunk record left behind. Independent of `FIX`: while it runs, it also removes any record-table slot whose declared size cannot fit the page, so compressing a database repairs that class of physical corruption even without `FIX`

The command returns the integrity check report in one record with the fields:

Field	Description
`avgPageUsed`	average space used in a page. This is a percentage value, where 100% means all the pages are full. If a record is too big, a page is split into multiple pages and multiple reads are necessary. Sometimes it helps to create buckets with larger pages to accommodate large records, see the `bucketDefaultPageSize` setting.
`warnings`	set of warning messages.
`totalSurrogateRecords`	number of records that have been moved between pages. This happens when a record is updated and doesn't fit the original page, so a surrogate (placeholder) is set to point to the new page/position or it's a piece of a record stored on multiple pages. Internal, can be ignored.
`totalPlaceholderRecords`	see above. Internal, can be ignored.
`pageSize`	internal, can be ignored.
`totalDeletedRecords`	number of records deleted in the database (allocated minus active). When a record is deleted, it's RID is not recycled!
`totalAllocatedRecords`	total number of records created in the database. A record can be of type document, vertex or edge.
`totalAllocatedDocuments`	see above, but only documents records.
`totalAllocatedVertices`	see above, but only vertex records.
`totalAllocatedEdges`	see above, but only edge records.
`totalActiveRecords`	total number of records that are "active" (not deleted). A record can be of type document, vertex or edge.
`totalActiveDocuments`	see above, but only vertex document.
`totalActiveVertices`	see above, but only vertex records.
`totalActiveEdges`	see above, but only edge records.
`totalMaxOffset`	internal, can be ignored.
`deletedRecordsAfterFix`	holds the list of RIDs deleted after a fix.

Field	Description
`corruptedRecords`	records that are present but cannot be read or are inconsistent. This should be zero; if not, it means there is a bug in the graph operations that doesn't keep operations 100% transactional. Please report this. A record that simply no longer exists is **not** listed here: an edge whose vertex was deleted is reported under `topMissingReferences` instead, and only the edge itself is counted as corrupted. The distinction matters because `FIX` rebuilds every index on a bucket that holds a corrupted record, which is wasted work when the record is not there at all.
`totalPages`	total number of pages in the database.
`rebuiltIndexes`	list of indexes that have been rebuilt automatically if corrupted records were found.
`corruptedIndexes`	names of the indexes the check found damaged on their own, independently of record corruption. The findings are in `warnings`. A `FIX` rebuilds each of them, and they are listed in `rebuiltIndexes` as well. See Vector indexes for the `LSM_VECTOR` finding.
`danglingIndexEntries`	*since v26.12.1*, entries of an `LSM_TREE` index whose record does not exist.
`staleIndexEntries`	*since v26.12.1*, entries of an `LSM_TREE` index whose record does not hold the entry's key (or is in another bucket).
`missingIndexEntries`	*since v26.12.1*, keys records hold that have no entry in their `LSM_TREE` index.
`topMissingReferences`	records an edge points at that could not be loaded, each with the number of edges that reference it - the usual shape after vertices were deleted while their edges survived. One line per target rather than per edge, so a single missing vertex referenced by millions of edges stays readable.
`distinctMissingReferences`	how many distinct targets `topMissingReferences` summarises.
`operation`	can be ignored, set by the console.
`invalidLinks`	similar to corrupted records; should be zero.
`totalErrors`	counter of errors encountered while checking the database. It should be zero.
`autoFix`	counter of autofix operations executed by the check database under the hood when corrupted records have been encountered. Some operations, like broken edges, can be fixed and restored automatically.
`unreadableStripeDirectories`	*since v26.10.2*, the RIDs of the edge-list stripe directories (the adjacency index of a high-degree vertex) this release cannot read, because their format version is unknown or they are truncated. Present, possibly empty, on every run; the reason for each is in `warnings`. They are not listed under `corruptedRecords`. A `FIX` rebuilds the owning vertex's edge list from the surviving edge records, and a full-scope `FIX` then reclaims the old directory. Before v26.10.2 a truncated directory read as an empty edge list.

Field	Description
`salvagedLightEdges`	*since v26.11.1*, lightweight edges a `FIX` carried over from the readable part of an edge list it rebuilt. Lightweight edges have no record, so the rebuild from the edge records cannot bring them back.
`recoveredLightEdges`	*since v26.11.1*, lightweight edges of a bidirectional type that sat in the unreadable part of a rebuilt edge list, rebuilt from their copy in the other endpoint's list.
`unsalvageableEdgeLists`	*since v26.11.1*, rebuilt edge lists that could not be read in full. A lightweight edge of a unidirectional type stored in the unreadable part had no other copy and is lost; the warning names the vertex and the direction.
`unreachableEdgeRecords`	orphan edge records: edge records whose endpoints are live vertices but that no vertex's edge list references. Each one is also reported as `ORPHAN RECORD` in `warnings`.
`relinkedOrphanEdgeRecords`	*since v26.11.1*, how many of those records a plain `FIX` relinked into their vertices' edge lists. The adjacency entries it wrote are counted in `reconnectedEdges` as well. Zero under `FIX DELETE ORPHANS`, which deletes them instead.
`orphanEdgeRecordsLeftToRelink`	*since v26.11.1*, how many orphan edge records a `FIX` run left for the next one. A run relinks at most what a tenth of the heap can plan (never fewer than 100,000), so a very large interrupted load is repaired by running `CHECK DATABASE FIX` again until it is zero.
`orphanedEdgeSegments`	number of unreachable edge-list segments found in the dedicated edge-list buckets (garbage left behind by repaired adjacency lists). Only present on a full-scope `FIX`.
`orphanedEdgeSegmentsReclaimed`	number of orphaned edge-list segments actually deleted by the `FIX`. Only present on a full-scope `FIX`.
`totalTimeSeriesTypes`	number of TimeSeries types the check walked. Zero on a database with none, so a clean report can be told apart from a pass that never ran.
`totalTimeSeriesShards`	number of TimeSeries shards walked, across all types.
`totalTimeSeriesSamples`	number of samples those shards hold, mutable and sealed together.
`totalTimeSeriesSealedBlocks`	number of compressed columnar blocks in the sealed stores.
`corruptedTimeSeries`	names of the TimeSeries types with at least one finding. The findings themselves are in `warnings`.
`repairedTimeSeries`	number of TimeSeries repairs a `FIX` applied. Deliberately not folded into `autoFix`, which counts record actions only.
`timeSeriesRepairs`	one line per repair applied, naming the type, the shard and what was rewritten.

Field	Description
`constraintViolatingRecords`	Records that load correctly but do not satisfy an existence constraint (`MANDATORY`, `NOTNULL`) of their own type. Deliberately not listed under `corruptedRecords`: such a record reads, deserialises and indexes correctly, and flagging it as corrupt would have `FIX` rebuild every index on its bucket for nothing.
`totalConstraintViolations`	how many such records the check found. Exact even when the list above is capped.
`deletedConstraintViolatingRecords`	which of them a `FIX DELETE INVALID RECORDS` actually removed. Empty on every other run, and capped like every other RID list - read the exact count from `totalDeletedConstraintViolatingRecords`, never from this list's size.
`totalDeletedConstraintViolatingRecords`	how many were removed. A record that another write completed while the check was running is **not** removed and not counted here: the repair re-checks each record immediately before deleting it.

Vector indexes

The check reads the pages of every `LSM_VECTOR` index and reports one that has two records on the same vector id. The index appears under `corruptedIndexes`, and `warnings` carries how many records are affected and the first few RIDs, each with the record that took its id.

Such an index never keeps its persisted graph. A graph build counts both records, while every load of the same pages keeps only the one written last, so the graph holds more nodes than the load has live vectors and is rejected as stale on the next load. Each search that finds it rejected waits for a full rebuild of the graph, and the rebuild produces the same graph again. The record that lost its vector id is not returned by a vector search until its vector is indexed again. A graph build now logs those records by RID and keeps one record per id, which ends the loop but does not restore them.

While an index has records left out this way it does not compact, because a compaction would erase the shared ids from the pages. The index keeps growing until it is rebuilt. `getStats()` exposes `compactionBlockedBySharedIds` and `recordsLeftOutBySharedIds` so this can be monitored. The check reads every page of each vector index and holds that index's read lock while it does, so writes to the index wait for the scan.

`CHECK DATABASE` alone only reports. `FIX` rebuilds the index from the records, which gives every record that holds an embedding a vector id of its own. The rebuild reads every vector of the type, so on a large index it takes as long as a `REBUILD INDEX`: run it off-peak. In a cluster the statement runs on the leader, wherever it is issued, and the rebuilt index reaches the followers as replicated data like any other `FIX` of an index.

Existence constraints

The check also asks a schema question: does every record satisfy the `MANDATORY` and `NOTNULL` constraints its own type declares? It is the one question nothing else in the database could answer, and a record that fails it is not merely untidy - every later write to it fails, because the update path validates the whole document, so the record reads as broken to the application long before anyone

thinks to look at the schema.

A record can end up in that state three ways, and only the first is a fault:

- an openCypher `CREATE`/`MERGE` pattern commits the record before the `SET` that completes it has run - which is what makes the standard `MERGE (n:L {id: $id}) SET n.prop = $v` upsert work the way it does in Neo4j - and the process dies, or the connection is killed, before the end of the statement;
- a `RESTORE DOCUMENT`/`VERTEX`/`EDGE` run by an earlier release, which did not apply the schema contract;
- a schema change that no existing record was measured against: `ALTER PROPERTY … MANDATORY TRUE` on a type that already holds records, which does not validate them, or renaming a `MANDATORY` property, which relabels the schema and leaves every record that predates it carrying the old name. Every one of them becomes a finding while holding data that is exactly as real as it was the day before the DDL ran.

Because the third is legitimate and the state it leaves is identical to the first, the **removal is a clause of its own**: a `FIX` that deleted by default would answer a DDL by destroying the type's contents. Reporting is always on.

There are two repairs, and which one is right depends on whether the records hold data:

- complete the record with a **single** `UPDATE` supplying every missing property. It has to be a single statement - the write path validates the whole document, so supplying two missing mandatory properties in two statements fails on the first;
- remove the records with `CHECK DATABASE FIX DELETE INVALID RECORDS`.

The two are safe to use together, including at the same time: the repair re-reads and re-checks every record inside the transaction that deletes it, so a record an `UPDATE` completed while the check was running is reported and left in place rather than deleted. A removal a `beforeDelete` trigger refuses is reported too, and not counted as removed. The removal is committed in batches sized by `arcadedb.checkDatabaseRepairBatchPages`, like every other `CHECK DATABASE … FIX` repair, so a bulk removal stays inside the replicated-entry limit under HA.

The check costs nothing on a database whose types declare no such constraint: the types to check are resolved from the schema before a record is read, so a database with none asks no question at all and its report simply says zero. *Since v26.10.2* a run that only reports asks the question while it decodes the content of each record, which every run does anyway, so a constrained type is not read a second time and the run has no `Checking existence constraints` step. `FIX DELETE INVALID RECORDS` keeps that step: its delete passes need a scan of their own.

Cached record counters

`SELECT count(*)` does not scan: every bucket keeps a record counter that each commit folds its delta into, and the counter is what the query answers from. That counter is persisted in the database's `statistics.json` and restored when the database is opened, so a counter that goes wrong stays wrong across restarts and nothing recomputes it until something invalidates it.

The bucket pass compares the two. It already reads every slot of every bucket, so the authoritative

number is in hand, and a disagreement is reported as an error naming the bucket, both numbers and the direction:

```
cached record count of bucket 'Order_0' is 10869 but the bucket holds 10906 records:
count(*) on this bucket under-reports by 37 - the counter is recomputed
```

The check recomputes it - the counter is invalidated, and the next `count(*)` rescans and republishes it. The report carries `staleRecordCounters` and `staleRecordCountersFixed`. Both count **buckets**, not records: one per bucket whose counter disagreed, whatever the size of the disagreement. They sit next to record tallies such as `totalActiveRecords` in the same JSON, so the distinction is worth reading twice.

This is the symptom to look for when `count(*)` disagrees with a forced full scan such as `SELECT count(*) FROM <type> WHERE @rid IS NOT NULL`, in **either** direction. On a replicated database the drift is per node and per bucket, so the replicas disagree with each other while holding identical data, which looks exactly like HA state divergence until a full scan rules it out.

Since v26.12.1 a read-only `CHECK DATABASE` recomputes a counter it proves wrong, as `FIX` does; before, only `FIX` did and a read-only run reported it with the advice to run `CHECK DATABASE FIX`. The counter is a cache of the node alone: no replication carries it and a restart reloads it as it was, so on a cluster replica, where `FIX` runs on the leader, a wrong counter had no repair at all. Recomputing it changes no record, so it is done whatever the mode.

a bucket whose counter has never been computed is not reported - there is nothing to be wrong yet - and neither is one the current transaction has pending changes on, nor one this same run has just repaired, since in those cases the two sides of the comparison describe different moments. The same applies to a read-only check that another transaction committed into while it was scanning, so running `CHECK DATABASE` on a live node under write load does not raise a false alarm.

Counters written by an earlier release

Earlier releases could fold a replicated transaction's record-count delta twice when a Raft entry was replayed on a node that had been restarted **cleanly**, which includes every rolling upgrade and every Kubernetes pod rotation. Nothing in `statistics.json` says which counters that happened to, so none of them is trusted: the file carries a format version, counts written without one are discarded when the database is opened, and each affected bucket recomputes on its next `count(*)`. That is one extra bucket scan, once, and the marker is written back on the next clean close. A `WARNING` in the server log says how many buckets it applied to, so the first `count(*)` that changes after an upgrade is not a mystery.

TimeSeries integrity

A TimeSeries type has neither record buckets nor indexes, so it is walked by a pass of its own that reads the three on-disk formats it owns: the mutable bucket (`.tstb`), the shared tag dictionary (`.tstd`) and the sealed store (`.ts.sealed`). The default tier reconciles every header and directory

against the data it describes and verifies the CRC32 of every sealed block - which means it reads the whole sealed file, the same cost class as the record scan the check already runs over every bucket. A block is otherwise CRC-checked only on the first read that touches it, so a block nothing queries is a block nothing verifies.

`DEEP` adds the checks that have to decompress. A CRC proves the bytes are the bytes that were written; it proves nothing about whether they mean what the block claims. Three things every read path answers queries from, without ever looking at a value, are verified only under `DEEP`:

- the block's timestamps are sorted — the range iterator binary-searches them, so an unsorted block silently returns a subset of the rows that match
- the block's declared per-column min/max/sum are what its values actually add up to — aggregation push-down answers `MIN`/`MAX`/`SUM`/`AVG` straight from them without decompressing
- the block's declared distinct tag values cover the values it holds — block pruning skips a whole block whose declaration does not list the value being filtered on

Each of those, when wrong, produces a wrong answer rather than an error, so nothing else in the engine reports it.

`FIX` repairs only what is **derived** and never a sample: the mutable bucket's page-0 counters, the sealed header's block count and global timestamp bounds, and the tail of an interrupted sealed append. Those matter beyond tidiness - a wrong global bound is read straight out of the header when the file is opened, so a range query pruned against it silently misses data the file holds, and a tail no reader can see also hides every block appended after it.

A sealed block that fails its CRC, or that `DEEP` finds inconsistent, is reported and left untouched. It is the only copy of the samples in it, so discarding it is a decision for an operator, who under HA can also rebuild a sealed store by recompacting from the replicated mutable pages.

Examples

- Execute the integrity check of the entire database without fixing any issue found.

```
ArcadeDB> CHECK DATABASE
```

- Execute the integrity check of the types 'Account' and 'Bill' without fixing any issue found.

```
ArcadeDB> CHECK DATABASE TYPE Account, Bill
```

- Execute the integrity check only on the bucket 'Account_Europe' without fixing any issue found.

```
ArcadeDB> CHECK DATABASE BUCKET Account_Europe
```

- Execute the integrity check of the entire database and auto fix any issues found.

```
ArcadeDB> CHECK DATABASE FIX
```

- Execute the deeper integrity check, which decodes the stored TimeSeries data instead of only reconciling what describes it.

```
ArcadeDB> CHECK DATABASE DEEP
```

- Execute the deeper check on one TimeSeries type, repairing the derived counters it finds wrong.

```
ArcadeDB> CHECK DATABASE TYPE SensorReading FIX DEEP
```

14.17.5. SQL - `EXPORT DATABASE`

Exports a database in the `exports` directory under the root directory where ArcadeDB is running.

Syntax

```
EXPORT DATABASE [<url>] [WITH ( <setting-name> = <setting-value> [,] )* ]
```

- `<url>` Optional, defines the location of the file to export. Use:
  - `file://` as prefix for files located on the same file system where ArcadeDB is running. For security reasons, it is not possible to provide an absolute or relative path to the file
  - By default the file name is set to `exports/<db-name>-export-<timestamp>.<format>.tgz`.
- `WITH` Optional settings as comma-separated key/value pairs:
  - `format` The format of the export. Default is `jsonl`.
    - `jsonl` exports in JSONL format - a newline-delimited JSON variant
    - `graphml` exports in the popular GraphML format. GraphML is supported by all the major Graph DBMS. GraphML has no type for collections and many scalar types: ArcadeDB writes such values as text with an `arcadedb:type` attribute that its own GraphML import reads back (see the note below). Using `graphson` instead of `graphml` is recommended
    - `graphson` exports in the GraphSON format supported by all the major Graph DBMS

      `graphml` and `graphson` are provided by the optional `arcadedb-gremlin` module. Without it on the classpath the export is refused, naming the format and the module.
  - `overwrite` Set to `true` to overwrite the export file if it already exists. Default is `false`. The value must be `true` or `false`; any other value is refused. *(Since v26.12.1: `overwrite = false` overwrote an existing file just like `overwrite = true`, because only the name of the setting was read, issue #11523.)*
  - `includeTypes` Comma-separated list of the types to export. Every other type is skipped. Cannot be combined with `excludeTypes`.

- `excludeTypes` Comma-separated list of the types not to export.
- `includeRecords` Comma-separated list of the RIDs of the records to export (for example `'#1:0,#1:1'`). Every other record is skipped. A LIGHTWEIGHT edge has no record id of its own, so it is selected by the id of its source vertex: listing a vertex exports its outgoing lightweight edges as well.

  The three filters are supported by the `jsonl` format only. `graphml` and `graphson` always export the whole graph and refuse them. *(Since v26.12.1: they were accepted and ignored, and the export wrote every record.)*

- `verboseLevel` How much the export logs, from `0` to `3`. Default is `2`. The value must be an integer in that range. *(Since v26.12.1: the setting was accepted and ignored.)*
- `compressionLevel` Deflate level of the export file, from `0` (stored, no compression: the fastest and the biggest file) to `9` (the smallest file and the slowest). Default is `6`, the level every export was written at before the setting existed. Level `1` writes a file about 10% bigger in a fraction of the time. *(Since v26.12.1)*
- `compressionThreads` Number of threads that compress the export file. `-1` (the default) sizes the pool automatically at half the available processors, capped at 8; `0` compresses on the thread that writes the records, as every export did before; any other value up to `256` is a pool of that size. With a pool the records are written while earlier parts of the file are compressed, which on a large database makes the export several times faster. The file is the same whichever value is used: one standard gzip stream, about the same size at the same level, read by `IMPORT DATABASE` and by any gzip tool. Each thread holds about 4 MB of buffers while the export runs (32 MB at 8 threads), so a very high value costs heap for no gain once the disk is the bottleneck. *(Since v26.12.1: the compression ran on the writing thread at the default level and took about 90% of the time of an export, issue #11539.)*

  The two settings apply to every format (`jsonl`, `graphml` and `graphson`), and are also available from the command line (`-compressionLevel`, `-compressionThreads`) and from the Java `Exporter` API (`setCompressionLevel`, `setCompressionThreads`). They are named like the `BACKUP DATABASE` ones, but the defaults differ: a backup favours speed (level `1`), an export is a portable dump and keeps the size it always had.

  Since v26.12.1 a setting that is not in this list is refused with `Unknown export setting '<name>'`, naming the closest setting when the name looks like a typo of it (`Did you mean 'excludeTypes'?`). Before, an unknown setting was ignored, so a misspelled filter exported the whole database and answered `OK`. The target file is set by the `<url>` of the statement only: `file` and `f` are not settings of `EXPORT DATABASE`, and neither is the command-line flag `o`.

An export claims its target file for its duration, so two exports that resolve to the same file no longer write into it at the same time - the second one fails immediately with `Another export to '<file>' is already in progress` instead of producing a corrupt archive. This applies with `overwrite = true` as well, which previously skipped the check altogether.

The claim is held in a sidecar file next to the target, so an export file name ending in `.exporting` is rejected as a reserved name. A running export now also publishes

its progress, so it is visible on the progress endpoint, in the console and in Studio.

The `jsonl` export includes TIMESERIES types: their definition (columns, shards, precision, retention, compaction interval and downsampling tiers) and their samples, so a JSONL export is a complete backup of a TIMESERIES type. Earlier versions wrote a file that could not be imported at all when the database contained one.

The same release also restores type-level `CUSTOM` metadata, type aliases and the bucket-selection strategy on import, and stops writing each vertex's edge identifiers into the export - no importer ever read them, and on a densely connected graph they were the bulk of the file. Edges are unaffected and are rebuilt as before.

Importing a `jsonl` export also restores the parts of the schema that live outside the types: triggers, materialized views, continuous aggregates, `DEFINE FUNCTION` libraries and module extensions. The export has always contained them; earlier versions simply skipped them on import and still reported success.

They are restored after the records, so a restored trigger does not fire again for data the source database had already processed. If one of them cannot be recreated - a trigger whose type no longer exists, for example - it is counted as a warning and the rest of the restore still completes.

On an HA cluster these schema members are restored as one schema change, which holds the database write lock while it runs and replicates to every node. *(Since v26.10.2: they reached only the leader's schema, so the followers, and a new leader after a failover, had the imported types and records but no triggers, views, aggregates or functions.)*

Installing a `JAVASCRIPT` or `JAVA` trigger, or a `js` function library, from an imported file requires the same `updateSecurity` permission that creating one by hand does, since either is code that runs inside the server.

Since 26.11.1 a `jsonl` export and import round trip keeps more of the database as it was (issue #10201):

- **Links to records that are not in the export.** A link whose target is not part of the export (the record was deleted before the export, or excluded by `includeTypes`, `excludeTypes` or `includeRecords`) is set to `null` on import. Inside a `LIST OF` or `MAP OF` property, which cannot hold `null`, the element or entry is dropped. The import counts them as `danglingLinks` in its result and logs a warning. If the property is `MANDATORY` or `NOT NULL` the record fails like any other record error. Before, such a link kept its old RID, which after the import named another record.
- **Values with no declaration keep their type.** Undeclared properties, elements of untyped `LIST` and `MAP` properties, and properties of embedded documents are restored with the type they had: a `DECIMAL` keeps its scale, `LONG`, `SHORT`, `BYTE` and

FLOAT stay themselves, dates and datetimes keep their full precision, BINARY values and primitive arrays stay arrays, and undeclared links stay links and are remapped to the imported records. The export (format version 2, exporterVersion 2 in the info line) writes, beside a record's p properties, a sparse type tree under the key y (tokens such as LONG, DATETIME, LINK, BINARY, ARRAY_OF_FLOATS, EMBEDDED; an array of per-element types for a list; an object of per-entry types for a map). Before, these values came back as whatever JSON parsed them into (an INTEGER or DOUBLE, a string, a list). A map with a t key is no longer mistaken for an embedded document.

- **Bucket count.** Each type is recreated with the number of buckets it had in the export (BUCKETS n, including 0 for an abstract type). Before, it got the default bucket count.
- **Declared lists and maps (issue #10391).** A LIST OF or MAP OF property is restored for every element type. Before, a declared LIST OF DATE, MAP OF DATE, LIST OF BINARY, MAP OF BINARY, LIST OF DATETIME_MICROS / DATETIME_NANOS, LIST OF ZONED_DATETIME, MAP OF DOUBLE or MAP OF FLOAT (holding a NaN or an infinity) aborted the import of the whole database, and a MAP OF DECIMAL lost the scale of its values (3.70 came back 3.7). A declared ZONED_DATETIME property, and a float[] or double[] holding a NaN or an infinity, are restored as well, and so is a MAP OF STRING that has a t key. A LIST OF or MAP OF whose elements the declaration reads back exactly (BOOLEAN, the integer types, FLOAT, DOUBLE, DECIMAL, STRING, DATETIME, LINK and the like) is written as it was, with no type tree, so the export of a schema-full database does not grow; any other element type is written with the tree. A file exported by 26.10.x imports where its data allows: its MAP OF DECIMAL keeps the scale its text carried, and its LIST OF DATE, MAP OF DATE, LIST OF BINARY and MAP OF BINARY are read back from the declaration. A NaN or an infinity inside a map or a float[] / double[] was written as null by that export, and stays null.
- **Links to records exported later, edge constraints and extreme dates (issue #10483).** The export writes every vertex before every document, so a LINK OF T, LIST OF T or MAP OF T (T a record type) can point at a record the import has not reached yet. The record is saved with the check of the link's target type suspended, and the link is resolved and the whole record validated again once every record is in: a link to a record of the wrong type still fails the import. Before, the check ran on the identity the target had in the source database, which in the new one names a record of another type, and the import aborted. A MAP OF LINK with a t key has its links remapped like any other. An edge is created with its properties, so an edge type with a MANDATORY or READONLY property imports. A LIST OF DATE or MAP OF DATE holding a date beyond about 292 million years from 1970 (LocalDate.MAX, LocalDate.MIN) is exported as its ISO text; before, the export skipped the record and failed.

Files exported by older versions have no y key and import exactly as before.

Since 26.11.1 the graphson and graphml exports write every property value (issue #10202):

- **GraphSON.** A `BYTE`, `SHORT`, `DECIMAL`, date or datetime property no longer makes the export of the whole database fail. Such values are written with TinkerPop's extended `gx:` types (`gx:Byte`, `gx:Int16`, `gx:BigDecimal`, `gx:LocalDate`, `gx:LocalDateTime`, ...), and `BINARY` values (`arcadedb:Binary`, base64), primitive arrays (`arcadedb:FloatArray`, ...) and Cypher durations with ArcadeDB's own `arcadedb:` types. `IMPORT DATABASE` reads them back with their types. A reader outside ArcadeDB needs TinkerPop's `GraphSONXModuleV3`, as for any GraphSON file with `gx:` types, and cannot read the `arcadedb:` types. Links are written as plain RID text.
- **GraphML.** The export is still standard GraphML in TinkerPop's layout (`labelV` / `labelE` keys), readable by TinkerPop and other GraphML tools. A key is declared with a GraphML type (`boolean`, `int`, `long`, `float`, `double`, `string`) only when all its values share it. A value GraphML has no type for (`BYTE`, `SHORT`, `DECIMAL`, dates, `BINARY` as base64, arrays, lists, and maps as JSON text) is written as text with an `arcadedb:type` attribute (namespace `https://arcadedb.com/xmlns/graphml`), which ArcadeDB's GraphML import reads back to the original type; other tools ignore the attribute and read the text. `BINARY` and `float[]` values are no longer written as Java identity strings (`[B@⋯`), and a key whose values have different types no longer aborts the import with a `NumberFormatException`. Links are written as plain RID text and imported as strings.

Examples

- Export the current database under the `exports/` directory with the default format and filename:

```
EXPORT DATABASE
```

- Export the current database to a specific file:

```
EXPORT DATABASE file://database.jsonl.tgz
```

- Export the current database in GraphSON format, overwriting any existing file:

```
EXPORT DATABASE file://Movies.graphson.tgz WITH format = 'graphson', overwrite = true
```

- Export as fast as possible, trading file size for time (fastest level, 8 compression threads):

```
EXPORT DATABASE file://database.jsonl.tgz WITH compressionLevel = 1,
compressionThreads = 8
```

- Export overwriting a previously exported file without specifying a URL:

```
EXPORT DATABASE WITH overwrite = true
```

A `jsonl` export of a TimeSeries type records each column's compression codec, because it cannot be worked out again from the column's data type alone. Importing such an export into a remote database preserves that codec exactly, even if the default codec for that data type changed between the two versions. Earlier versions could silently give the restored column the newer default instead.

14.17.6. SQL - IMPORT DATABASE

Executes an import of the database into the current one. Usually an import database is executed on an empty database, but it is possible to execute on any database. In case of conflict (unique index key already existent, etc.), the conflicting records will not be imported. The importer automatically recognize the file between the following formats:

- **OrientDB** database export
- **Neo4J** database export
- **GraphML** database export. GraphML has no type for collections and many scalar types; values exported by ArcadeDB carry an `arcadedb:type` attribute that restores their type (see the note on graph formats). Using GraphSON instead of GraphML is recommended
- **GraphSON** database export
- **JSON** documents or responses
- **JSON Lines** documents or responses

Syntax

```
IMPORT DATABASE <url> [WITH ( <setting-name> = <setting-value> [,] )* ]
```

- `<url>` Defines the location of the file to import. Use:
  - `file://` as prefix for files located on the same file system where ArcadeDB is running. The path is resolved against the directory where ArcadeDB is running, so a file written by `EXPORT DATABASE` (which writes to `exports/`) is read back with `file://exports/<file-name>`.
  - `https://` and `http://` as prefix for remote files.

Examples

- Import into the current database an export made with `EXPORT DATABASE file://mydb.jsonl.tgz`

```
IMPORT DATABASE file://exports/mydb.jsonl.tgz
```

- Import the public OpenBeer database available as demo database for OrientDB and exported in TGZ file

```
IMPORT DATABASE https://github.com/ArcadeData/arcadedb-
datasets/raw/main/orientdb/OpenBeer.gz
```

- Import the Movie database used in Neo4j's examples:

```
IMPORT DATABASE https://github.com/ArcadeData/arcadedb-
datasets/raw/main/neo4j/movies.graphson.tgz
```

- Import a JSON response into document type `mytype`

```
IMPORT DATABASE http://echo.jsontest.com/key/value/one/two WITH documentType =
'mytype'
```

- Test data source

```
IMPORT DATABASE http://echo.jsontest.com/key/value/one/two WITH probeOnly = true
```

- Import a graph from CSV, with vertices in a `vertices.csv` which contains a column `Id`, and edges in a `edges.csv` which contains columns `From` and `To`, both placed in the ArcadeDB base folder:

```
IMPORT DATABASE file://empty.csv WITH vertices="file://vertices.csv",
verticesFileType=csv, typeIdProperty=Id, typeIdType=Long, edges="file://edges.csv",
edgesFileType=csv, edgeFromField="From", edgeToField="To"
```

Result

The statement returns a single row with a `result` column: `OK` when the import completed, or `FAIL` when the source or one of the settings could not be used, in which case a `reason` column explains why. Any other failure raises an error.

See also Importer for a description of the settings.

14.18. Inspection

14.18.1. SQL - `EXPLAIN`

EXPLAIN SQL command returns information about query execution planning of a specific statement, without executing the statement itself.

Syntax

```
EXPLAIN <command>
```

- `<command>` Defines the command that you want to profile, eg. a SELECT statement

Examples

- Profile a query that executes on a type filtering based on an attribute:

```
ArcadeDB {db=foo}> explain select from v where name = 'a'

Profiled command '[{

executionPlan:{...},

executionPlanAsString:

+ FETCH FROM TYPE v
  + FETCH FROM BUCKET 9 ASC
  + FETCH FROM BUCKET 10 ASC
  + FETCH FROM BUCKET 11 ASC
  + FETCH FROM BUCKET 12 ASC
  + FETCH FROM BUCKET 13 ASC
  + FETCH FROM BUCKET 14 ASC
  + FETCH FROM BUCKET 15 ASC
  + FETCH FROM BUCKET 16 ASC
  + FETCH NEW RECORDS FROM CURRENT TRANSACTION SCOPE (if any)
+ FILTER ITEMS WHERE
  name = 'a'

}]' in 0,022000 sec(s):
```

For more information, see:

- SQL Commands
- PROFILE

14.18.2. SQL - `PROFILE`

The `PROFILE` SQL command returns information about query execution planning and statistics for a specific statement. The statement is actually executed to provide the execution stats. Because it runs the statement, `PROFILE` is classified as the statement it wraps: `PROFILE SELECT …` is a read, while `PROFILE INSERT …`, `PROFILE UPDATE …` or `PROFILE CREATE TYPE …` is a write (or a schema change) and must be sent where a write is accepted, through `command()` or `POST /api/v1/command`; on the read-only path (`query()`, `/api/v1/query`) it is refused like the bare statement. `EXPLAIN` only plans, so it stays read-only whatever it wraps. *(Since v26.10.2)*

The result is the execution plan of the query (like for EXPLAIN) with additional information about execution time spent on each step, in microseconds.

Syntax

```
PROFILE <command>
```

- `<command>` Defines the command that you want to profile, eg. a SELECT statement

Examples

```
PROFILE SELECT sum(Amount), OrderDate
FROM Orders
WHERE OrderDate > date("2012-12-09", "yyyy-MM-dd")
GROUP BY OrderDate
```

result:

```
+ FETCH FROM INDEX Orders.OrderDate (1.445µs)
  OrderDate > date("2012-12-09", "yyyy-MM-dd")
+ EXTRACT VALUE FROM INDEX ENTRY
+ FILTER ITEMS BY TYPE
  Orders
+ CALCULATE PROJECTIONS (5.065µs)
  Amount AS _$$$OALIAS$$_1, OrderDate
+ CALCULATE AGGREGATE PROJECTIONS (3.182µs)
  sum(_$$$OALIAS$$_1) AS _$$$OALIAS$$_0, OrderDate
  GROUP BY OrderDate
+ CALCULATE PROJECTIONS (1.116µs)
  _$$$OALIAS$$_0 AS `sum(Amount)`, OrderDate
```

You can see the `(1.445µs)` at the end of the first line, it means that fetching from index `Orders.OrderDate` took 1.445 microseconds (1.4 milliseconds)

For more information, see:

- EXPLAIN

14.18.3. SQL - CONSOLE

Writes a string message of given log-level to the log. Log-levels are:

- `output` (none)
- `log` (`INFO`)
- `error` (`SEVERE`)
- `warn` (`WARNING`)
- `debug` (`FINE`)

This command is useful for SQL scripts.

To flush to the log, suffix the message with a newline `\n`.

Syntax

```
CONSOLE.logLevel <expression>
```

The command returns which page have been aligned on each server.

Examples

- Write a message of level `INFO` to the log.

```
CONSOLE.log map('Hello','World')
```

14.19. Projections

A projection is a value that is returned by a query statement (SELECT, MATCH).

Eg. the following query

```
SELECT name AS firstName, age * 12 AS ageInMonths, out("Friend")
FROM Person
WHERE surname = 'Smith'
```

has three projections:

- `name as firstName`
- `age * 12 as ageInMonths`
- `out("Friend")`

Syntax

A **projection** has the following syntax:

`<expression> [<nestedProjection>] [ AS <alias> ]`

- `<expression>` is an expression (see SQL Syntax) that represents the way to calculate the value of the single projection
- `<alias>` is the Identifier (see SQL Syntax) representing the field name used to return the value in the result set

A projection block has the following syntax:

`[DISTINCT] <projection> [, <projection> ]*`

- `DISTINCT`: removes duplicates from the result-set

14.19.1. Query result

By default, a query returns a different result-set based on the projections it has:

- **no projections**: The result set is made of records as they arrive from the target, with the original `@rid` and `@type` attributes (if any)
- `*` **alone**: same behavior as without `*`.
- `*` **plus other projections**: records of the original target, merged with the other projection values, with `@rid` and `@type` of the original record.
- `expand(<projection>)`: The result set is made of the records returned by the projection, expanded (if the projection result is a link or a collection of links) and unwinded (if the projection result is a collection or an array, one record per element). Any other value - a number, a string, a map - gives one record holding that value. A `BINARY` property is one such value: it is an opaque blob, so it is never split into one record per byte.
- **one or more projections**: temporary records (with temporary `@rid` and no `@type`). Projections that represent links are returned as simple @rid values, unless differently specified in the fetch plan.

Projection values can be overwritten in the final result, the overwrite happens from left to right.

eg.

```
SELECT 1 AS a, 2 AS a
```

will return `[{"a":2}]`

eg.

Having the record `{"@type":"Foo", "name":"bar", "@rid":"#12:0"}`

```
SELECT *, "hey" AS name FROM Foo
```

will return `[{"@type":"Foo", "@rid":"#12:0", "name":"hey"}]`

`*` is accepted only as the first projection item: `SELECT "hey" AS name, * FROM Foo` is a syntax error.

When saving back a record with a valid rid, you will overwrite the existing record! So pay attention when using `*` together with other projections.

The result of the query can be further unwound using the `UNWIND` operator.

`expand()` cannot be used together with `GROUP BY`.

Aliases

The alias is the field name that a projection will have in the result-set.

An alias can be implicit, if declared with the `AS` keyword, eg.

```
SELECT name + " " + surname AS full_name FROM Person
```

Result: `[{"full_name":"John Smith"}]`

An alias can be implicit, when no `AS` is defined, eg.

```
SELECT name FROM Person
```

Result: `[{"name":"John"}]`

An implicit alias is calculated based on how the projection is written. By default, ArcadeDB uses the plain string representation of the projection as alias.

```
SELECT 1+2 AS sum
```

Result: `[{"sum": 3}]`

```
SELECT parent.name+" "+parent.surname AS full_name FROM Node
```

Result: `[{"full_name": "John Smith"}]`

The string representation of a projection is the exact representation of the projection string, without spaces before and after dots and brackets, no spaces before commands, a single space before and after operators.

eg.

```
SELECT 1+2
```

Result: `[{"1 + 2": 3}]`

Note the space before and after the plus symbol.

```
SELECT parent.name+" "+parent.surname FROM Node
```

Result: `[{"parent.name + \" \" + parent.nurname": "John Smith"}]`

```
SELECT items[4] from Node
```

Result: `[{"items[4]": "John Smith"}]`

14.19.2. Nested projections

Syntax:

`":{" ( * | (["!"] <identifier> ["*"] (<comma> ["!"] <identifier> ["*"])* ) ) "}"`

Inside the braces, `*` projects every property, `<identifier>` includes one property and `!<identifier>` excludes it. A trailing `*` turns the identifier into a prefix: `:{s*}` includes every property whose name starts with `s`, each under its own name, and `:{!s*}` excludes them (see the prefix wildcard examples below).

A projection can refer to a link or to a collection of links, eg. a LIST or MAP. In some cases you can be interested in the expanded object instead of the RID.

Here is an example to remove the `@rid` and `@type` properties from the record (a nested projection cannot follow `*`, so apply it to `@this`):

```
SELECT @this:{!@rid,!@type} FROM doc
```

Let's clarify this with another example. This is our dataset:

@rid	name	surname	parent
#12:0	foo	fooz	
#12:1	bar	barz	#12:0
#12:2	baz	bazz	#12:1

Given this query:

```
SELECT name, parent FROM TheType WHERE name = 'baz'
```

The result is

```
{
   "name": "baz",
   "parent": #12:1
}
```

Now suppose you want to expand the link and retrieve some properties of the linked object. You can do it explicitly do it with other projections:

```
SELECT name, parent.name FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent.name": "bar"
}
```

but this will force you to list them one by one, and it's not always possible, especially when you don't know all their names.

Another alternative is to use nested projections, eg.

```
SELECT name, parent:{name} FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent": {
      "name": "bar"
   }
}
```

or with multiple attributes

```
SELECT name, parent:{name, surname} FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent": {
      "name": "bar"
      "surname": "barz"
   }
}
```

or using a wildcard

```
SELECT name, parent:{*} FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent": {
      "name": "bar"
```

```
      "surname": "barz"
      "parent": #12:0
   }
}
```

You can also use the `!` exclude syntax to define which attributes you want to *exclude* from the nested projection:

```
SELECT name, parent:{!surname} FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent": {
      "name": "bar"
      "parent": #12:0
   }
}
```

You can also use a wildcard on the right of property names, to specify the inclusion of attributes that start with a prefix, eg.

```
SELECT name, parent:{surna*} FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent": {
      "surname": "barz"
   }
}
```

or their exclusion

```
SELECT name, parent:{!surna*} FROM TheType WHERE name = 'baz'
```

```
{
   "name": "baz",
   "parent": {
      "name": "bar",
      "parent": #12:0
   }
}
```

A prefix wildcard works on every value a nested projection applies to: a record, a map, and a plain

result row (a row of a `LET` subquery, for example). It matches real property names only, not `@rid` or `@type`. It combines with excludes in the same braces and with a nested expansion, which then runs on each matching property:

```
SELECT @this:{s*} FROM TheType                    -- every property starting with 's'
SELECT @this:{n*, !nested} FROM TheType           -- 'n...' properties except 'nested'
SELECT @this:{ne*:{a}} FROM TheType               -- each 'ne...' property, projected to
its 'a'
```

(Since v26.10.2.) A prefix wildcard cannot take an alias and must follow a plain property name: `{se* AS k}` fails with "Alias 'k' is not allowed on the wildcard projection item 'se*': the wildcard can match several properties, each projected under its own name", and `{a.b*}` with "The wildcard projection item 'a.b*' is not valid: a wildcard must follow a plain property name prefix" (see Nested projections).

Nested projection syntax allows for multiple level depth expressions, eg. you can go three levels deep as follows:

```
SELECT name, parent:{name, surname, parent:{name, surname}} FROM TheType WHERE name =
'baz'
```

```
{
   "name": "baz",
   "parent": {
      "name": "bar"
      "surname": "barz"
      "parent": {
         "name": "foo"
         "surname": "fooz"
      }
   }
}
```

You can also use expressions and aliases in nested projections:

```
SELECT name, parent.parent:{name, surname} as grandparent FROM TheType WHERE name =
'baz'
```

```
{
   "name": "baz",
   "grandparent": {
      "name": "foo"
      "surname": "fooz"
   }
```

```
}
```

Finally, you can rename fields with `AS`:

```
SELECT name, parent.parent:{name AS givenname} as grandparent FROM TheType WHERE name
= 'baz'
```

```
{
   "name": "baz",
   "grandparent": {
      "givenname": "fooz"
   }
}
```

CASE Expression

The `CASE` expression provides conditional logic similar to if-then-else statements in programming languages. It supports two forms: **searched CASE** and **simple CASE**.

Searched CASE Syntax

Evaluates conditions sequentially and returns the result of the first matching condition:

```
CASE
  WHEN <condition1> THEN <result1>
  WHEN <condition2> THEN <result2>
  [ELSE <default_result>]
END
```

Simple CASE Syntax

Compares an expression against multiple values:

```
CASE <expression>
  WHEN <value1> THEN <result1>
  WHEN <value2> THEN <result2>
  [ELSE <default_result>]
END
```

Behavior

- The `CASE` expression evaluates conditions in order and returns the result of the first matching `WHEN` clause
- If no conditions match and an `ELSE` clause is provided, the `ELSE` result is returned
- If no conditions match and there is no `ELSE` clause, `null` is returned

- Multiple `WHEN` clauses can be specified
- Can be used in `SELECT` projections, `WHERE` clauses, `MATCH` statements, `LET` clauses, and expressions
- Supports nested `CASE` expressions for complex conditional logic
- `NULL` values in conditions follow SQL three-valued logic (NULL comparisons return NULL, which is treated as false)

Examples

Searched CASE with conditions in SELECT:

```
SELECT
  name,
  CASE
    WHEN age < 18 THEN 'Minor'
    WHEN age >= 18 AND age < 65 THEN 'Adult'
    ELSE 'Senior'
  END as ageGroup
FROM Person
```

Simple CASE with value matching:

```
SELECT
  name,
  CASE status
    WHEN 'active' THEN 1
    WHEN 'inactive' THEN 0
    ELSE -1
  END as statusCode
FROM Account
```

CASE without ELSE (returns null if no match):

```
SELECT
  name,
  CASE
    WHEN score > 90 THEN 'Excellent'
    WHEN score > 75 THEN 'Good'
  END as grade
FROM Student
```

CASE in WHERE clause for filtering:

```
SELECT name, age
FROM Person
WHERE CASE
  WHEN age IS NULL THEN false
```

```
  WHEN age < 18 THEN 'minor'
  ELSE 'adult'
END = 'adult'
```

CASE in MATCH WHERE clause (enum-to-string conversion with pattern matching):

```
MATCH {type: Product, as: prod,
  where: ((CASE
    WHEN color = 1 THEN 'red'
    WHEN color = 2 THEN 'blue'
    WHEN color = 3 THEN 'green'
  END) ILIKE '%ed%')}
RETURN prod.name
```

This is particularly useful for converting enum values (stored as integers) to human-readable strings that can be filtered with wildcards.

CASE in MATCH RETURN clause:

```
MATCH {type: Person, as: p, where: (age IS NOT NULL)}
RETURN p.name,
  CASE
    WHEN p.age < 18 THEN 'minor'
    WHEN p.age < 65 THEN 'adult'
    ELSE 'senior'
  END as category
ORDER BY p.name
```

Nested CASE expressions:

```
SELECT
  name,
  CASE
    WHEN balance > 0 THEN
      CASE
        WHEN balance > 10000 THEN 'Premium'
        ELSE 'Standard'
      END
    ELSE 'Overdrawn'
  END as accountStatus
FROM Account
```

CASE in LET clause:

```
SELECT $category
FROM Product
```

```
LET $category = CASE
  WHEN price < 10 THEN 'Budget'
  WHEN price < 100 THEN 'Standard'
  ELSE 'Premium'
END
```

CASE with NULL handling:

```
SELECT
  name,
  CASE
    WHEN age IS NULL THEN 'Unknown'
    WHEN age < 18 THEN 'Minor'
    ELSE 'Adult'
  END as status
FROM Person
```

14.20. Pagination

ArcadeDB supports pagination natively. Pagination doesn't consume server side resources because no cursors are used. Only Record ID's are used as pointers to the physical position in the bucket.

There are 2 ways to achieve pagination:

Use the SKIP-LIMIT

The first and simpler way to do pagination is to use the `SKIP`/`LIMIT` approach. This is the slower way because ArcadeDB repeats the query and just skips the first X records from the result. Syntax:

```
SELECT FROM <target> [WHERE ...] SKIP <records-to-skip> LIMIT <max-records>
```

Where: - **records-to-skip** is the number of records to skip before starting to collect them as the result set - **max-records** is the maximum number of records returned by the query

Since v26.12.1, when the query reads an index in key order (an `ORDER BY` on the indexed property, with or without a range on it in the `WHERE`) and nothing between the index and the `SKIP` needs the record (no other `WHERE` condition, `DISTINCT`, `GROUP BY`, `expand()`, `UNWIND` or `LET`), the skipped rows are counted from the index entries without loading their records, and the projections are not computed for them. `SELECT id FROM P WHERE id >= 0 ORDER BY id SKIP 50000 LIMIT 10` therefore no longer loads 50,000 records to throw them away. The rows returned are the same, records of a type with a read trigger or read event listener are still loaded, and a user who may not read a skipped record still gets the same security error. The index entries are still walked one by one, so a `WHERE` on the last key read (`WHERE id > <last-id> ORDER BY id LIMIT <max-records>`) stays faster for deep pages.

Use the RID-LIMIT

This method is faster than the `SKIP-LIMIT` because ArcadeDB will begin the scan from the starting RID. ArcadeDB can seek the first record in about O(1) time. The downside is that it's more complex to use.

The trick here is to execute the query multiple times setting the `LIMIT` as the page size and using the greater than `>` operator against `@rid`. The **lower-rid** is the starting point to search, for example `#10:300`.

Syntax:

```
SELECT FROM <target> WHERE @rid > <lower-rid> ... [LIMIT <max-records>]
```

Where: - **lower-rid** is the exclusive lower bound of the range as RID - **max-records** is the maximum number of records returned by the query

In this way, ArcadeDB will start to scan the bucket from the given position **lower-rid** + 1. After the first call, the **lower-rid** will be the rid of the last record returned by the previous call. To scan the cluster from the beginning, use `#-1:-1` as **lower-rid** .

14.21. Filtering

The `WHERE` condition is shared among many SQL commands. A condition results in a boolean; and a compound condition, consisting of multiple conditions that are combined with logical operators, is resolved from left to right.

Syntax

`[<item>] <operator> <item>`

Items

And `item` can be:

What	Description	Example
field	Document field	`WHERE price > 1000000`
field<indexes>	Document field part. To know more about field part look at the full syntax: Document-API-Property	`WHERE tags[name='Hi'] OR tags[0...3] IN ('Hello') AND employees IS NOT NULL`
record attribute	Record attribute name with `@` as prefix	`WHERE @type = 'Profile'`
Functions	Any Function between the defined ones	`WHERE distance(x, y, 52.20, 0.14) <= 30`
`$variable`	Context variable prefixed with `$`	`WHERE $depth <= 3`

Record attributes

Name	Description	Example
`@this`	returns the record itself	`SELECT @this.toJSON() FROM Account`
`@rid`	returns the RID in the form `bucket:position`. It is null for embedded records. *NOTE:* using `@rid` in a "Where" condition slows down queries. Much better to use the RID as target. Example: change `SELECT FROM Profile WHERE @rid = #10:44` to `SELECT FROM #10:44`	`@rid = #11:0`
`@size`	returns the record size in bytes	`@size > 1024`
`@type`	returns the record type between: 'document', 'column', 'flat', 'bytes'	`@type = 'flat'`
`@out`	on an edge, returns the vertex the edge leaves, as a link: `@out.name` reads a property of that vertex. *Since v26.11.1* plain `out` reads the same vertex on an edge with no property of that name (before, it read as null). `WHERE @out = #rid` reads only that vertex's edges instead of scanning the edge type, and *since v26.12.1* so does `WHERE out = #rid` when no record or type of the database uses a property named `out`. outV() gives the same vertex	`@out = #12:0`
`@in`	on an edge, returns the vertex the edge points to, as a link. Plain `in` behaves like plain `out`. inV() gives the same vertex	`@in.name = 'Rome'`

14.21.1. Operators

Conditional Operators

Apply to	Operator	Description	Example
any	`=` or `==`	Equals. A list on the right side is compared as a whole, so `name = ['a','b']` matches nothing, with or without an index; use `IN` to match any of several values *(since v26.10.2; before, an index expanded the list like `IN`)*. Two lists, or two maps, are equal when they hold the same elements in the same order (the same keys for a map) and the numbers in them compare as two numbers do, so `tags = [1, 2]` also matches a list written with 64-bit integers by the Java API or a client *(since v26.11.1; before, the elements were compared by Java class and `[1, 2]` differed from the same list of `long` values)*. Every other element is compared as `=` compares two scalars, so two values `=` reports equal are equal as elements too: a `java.util.Date`, an `Instant`, a `Calendar` or a `java.sql.Timestamp` is equal to the stored `LocalDateTime` of the same instant inside a list or a map, as it is on its own *(since v26.11.1; before, the list was a different value and `<>` answered its row)*. `<>`, `IN` and `CONTAINS` follow the same rule. A list indexed `BY ITEM` holds	`name = 'Luke'`

Apply to	Operator	Description	Example
any	`<=>`	Null-safe-equals, is also true if left and right operands are `NULL`	`name <=> word`
string	`LIKE`	Similar to equals, but allows the wildcards '%' that means "any characters", none included (`'' LIKE '%'` is true, *since v26.10.2*), and '?' that means "any single character". Use `\%` and `\?` to match the characters themselves. `LIKE` is case sensitive. A pattern with a literal start (`'Luk%'`) uses an ordered index on the property, except a case insensitive one; a pattern starting with a wildcard does not.	`name LIKE 'Luk%'`
string	`ILIKE`	Similar to `LIKE`, but `ILIKE` is case insensitive.	`name ILIKE 'lUk%'`
string	`NOT LIKE`	Negation of `LIKE`, equivalent to `NOT (x LIKE y)`	`name NOT LIKE 'Leia'`
string	`NOT ILIKE`	Negation of `ILIKE`, equivalent to `NOT (x ILIKE y)`	`name NOT ILIKE 'lEIA'`
any	`<`	Less than	`age < 40`
any	`<=`	Less or equal to	`age <= 40`
any	`>`	Greater than	`age > 40`
any	`>=`	Greater or equal to	`age >= 40`
any	`<>` or `!=`	Not equal to	`age <> 40`
any	`BETWEEN`	The value is between a range. It's equivalent to `<field> >= <from-value> AND <field> <= <to-value>`	`price BETWEEN 10 AND 30`
any	`IS`	Used to test if a value is `NULL`	`children IS NULL`

Apply to	Operator	Description	Example
any	`IS NOT`	Used to test if a value is not `NULL`	`children IS NOT NULL`
record, string (as type name)	`INSTANCEOF`	Used to check if the record extends a type	`@this INSTANCEOF 'Customer' OR @type INSTANCEOF 'Provider'`
collection	`IN`	contains any of the elements listed. Uses the index on the property, including the leading columns of a composite index (`transactionId IN (...)` on an index over `(transactionId, date)`), with every list form: `IN (a, b)`, `IN [a, b]` and a bound parameter. The listed values are looked up in index order, so an `ORDER BY` on the indexed columns needs no extra sort step. The searched value is converted to the type of the listed items when needed, so `? IN doubleList` with the String `'7'` finds `7.0` with or without an index, like `CONTAINS`. A property on the left side is never converted: `stringProp IN [7.0]` does not match `'7'`, exactly like `stringProp = 7.0` and the index. Without an index, `IN (?)` with a single parameter and `IN (SELECT ...)` compare numbers and dates by value too, so a Long `7` matches an Integer `7`, and `1.50` matches `1.5`	`name IN ['European','Asiatic']`

Apply to	Operator	Description	Example
collection	`NOT IN`	contains all of the elements not listed. Does not use index: an index can look up the listed values, not their complement, so the type is scanned. If the type is large, add a second indexed condition to the query to drive the lookup	`name NOT IN ['European','Asiatic']`
collection	`CONTAINS`	true if the collection contains at least one element that satisfy the next condition. Use `@this` as right item for lists. Condition can be a single item: in this case the behaviour is like the `IN` operator. A one-element list on the right, `CONTAINS [x]`, compares `x` under the `=` rule, so a `BINARY` value matches by content and an Integer matches a stored Long *(since v26.11.1)*	`children CONTAINS (name = 'Luke')`; `map.values() CONTAINS (name = 'Luke')`; `['a','b'] CONTAINS (@this ILIKE 'A')`

Apply to	Operator	Description	Example
collection	`CONTAINSALL`	true if every element of the right collection is contained in the left collection. A single value on the left is treated as a one-element collection. `CONTAINSALL (<condition>)` is true when every element satisfies any condition, e.g. `items CONTAINSALL (name IS NOT NULL)` *(any condition; before, only OR conditions)*. On a list indexed `BY ITEM` the index looks up one of the elements and every record it returns is checked for the others, so a right side of several elements uses the index too, and answers as a scan does whether it is a list, a set or an array parameter such as `int[]` *(since v26.11.1; before, several elements were scanned, and an array of several elements answered like `CONTAINSANY`)*. An empty right side is true for every collection and is answered by a scan. A `BINARY` (`byte[]`) right side is one value searched for, as `CONTAINS` and `CONTAINSANY` read it *(since v26.11.1; before, it was expanded into its bytes)*	`children.name CONTAINSALL ['Luke','Leia']`

Apply to	Operator	Description	Example
collection	`CONTAINSANY`	true if at least one element of the right collection is contained in the left collection. A single value on the left is treated as a one-element collection, so `uuid CONTAINSANY ['u1','u2']` answers like `uuid IN ['u1','u2']`. `CONTAINSANY (<condition>)` is true when at least one element satisfies the condition, so it is false on an empty collection	`children.name CONTAINSANY ['Jacen','Jaina','Ben']`
map	`CONTAINSKEY`	true if the map contains at least one key equals to the requested. You can also use `map.keys() CONTAINS` in place of it	`connections CONTAINSKEY 'Luke'`
map	`CONTAINSVALUE`	true if the map contains at least one value equals to the requested. You can also use `map.values() CONTAINS` in place of it	`connections CONTAINSVALUE 10:3`
string	`CONTAINSTEXT`	When used against an indexed field, a lookup in the index will be performed with the text specified as key. When there is no index a simple Java `indexOf` will be performed. So the result set could be different if you have an index or not on that field	`text CONTAINSTEXT 'jay'`

Apply to	Operator	Description	Example
string	`MATCHES`	Matches the string using a Regular Expression. Use the modifier `(?s)` to make `.` match also newlines, and `(?i)` for case insensitive patterns.	``text MATCHES `\b[A-Z0-9.%+-@[A-Z0-9.-]\.[A-Z]{2,4}\b]` ``

`LIKE`, `ILIKE`, `MATCHES` are not using full-text indexes.

A full-text index matches words, not whole values, so it only answers `CONTAINSTEXT` and the full-text search functions. Exact operators (`=`, `IN`, `CONTAINS`, `CONTAINSANY`, `CONTAINSALL`) never use it and always return exact matches, also on a list indexed `BY ITEM`. To have an exact lookup indexed, create a regular (`UNIQUE`/`NOTUNIQUE`) index.

`CONTAINS` asks whether the collection on the left holds a **single** value: either one matching the condition, or, when the right side is a plain value, one equal to it. Put a collection on the right and it is searched for as one element, so `tags CONTAINS ['red','blue']` matches a row whose `tags` holds the nested list `['red','blue']`, and not a row whose `tags` is `['red','green','blue']`. To compare two collections, use `CONTAINSALL` (every element of the right is present on the left) or `CONTAINSANY` (at least one is). This applies equally when the right side is produced by an expression such as `'red blue'.split(' ')`, and with a list indexed `BY ITEM`, which is used only for a single value or a right side of exactly one element *(since v26.11.1; before, the index answered* `tags CONTAINS ['red','blue']` *like* `CONTAINSANY`*)*.

A nested list is an element of its own kind: a plain value never matches it, so `tags CONTAINS 3` does not match a row whose `tags` is `[[3]]`. A right side of exactly one element is that element or the list it forms, so `tags CONTAINS [3]` matches both `[3]` and `[[3]]`. The scan and a `BY ITEM` index answer alike, on a `LIST` declared without an item type too *(since v26.12.1; before, the scan matched* `[[3]]` *for* `CONTAINS 3`*, and the index missed it for* `CONTAINS [3]` *and matched it for* `CONTAINS '[3]'`*)*.

An index on a property is used whichever side of the comparison the property is written on, and under `NOT`: `5 = id` is looked up like `id = 5`, `:lo < id` like `id > :lo`, `NOT (id <> 5)` like `id = 5` and `NOT (id >= 3)` like `id < 3`. This holds for `SELECT`, the `WHERE` of `UPDATE` and `DELETE`, and the `where:` of a `MATCH` node. When the value is written first, or a range (`<`, `<=`, `>`, `>=`) is under `NOT`, the index is used only when the value is a literal or a parameter of the property's type (a number for a numeric property, a string for a `STRING` one). A comparison converts its right side to the type of its left side, so with a value of another type the two spellings compare differently (`'10' > id` compares text, `id < '10'` compares numbers) and the records are scanned. `NOT (id <> v)` needs no such check. Every spelling returns the same records as before *(since v26.12.1; before, only the property written on the left used the index)*.

Logical Operators

Operator	Description	Example
AND	true if both the conditions are true	`name = 'Luke' AND surname LIKE 'Sky%'`
OR	true if at least one of the condition is true	`name = 'Luke' OR surname LIKE 'Sky%'`
NOT	true if the condition is false. `NOT` needs parenthesis on the right with the condition to negate	`NOT (name = 'Luke')`

NULL and unknown

Comparing with a missing or `NULL` value is neither true nor false but *unknown*, as in standard SQL. This applies to `=`, `<>`, `!=`, `<`, `<=`, `>`, `>=`, `BETWEEN`, `LIKE`, `ILIKE` and `MATCHES` *(`LIKE`, `ILIKE` and `MATCHES` since v26.11.1)*. A `WHERE` keeps a record only when the condition is true, and `NOT` of an unknown stays unknown, so `WHERE age <> 5` and `WHERE NOT (age = 5)` do not return records without `age`, the same as `age NOT IN [5]`. Likewise `age = NULL` and `age <> NULL` match nothing, and a boolean used as a value (`SELECT age <> 5`, `CASE WHEN age <> 5 ...`) is `NULL` for those records.

To test for a missing value use `IS NULL`, `IS NOT NULL` or the null-safe `<=>`. To keep the records without a value, write it out: `WHERE age <> 5 OR age IS NULL`.

`IN` follows the same logic: `3 NOT IN [4, NULL]` is unknown, because the `NULL` might be 3, so a `NOT IN` whose list or sub-query holds a `NULL` returns no record. Use `NOT IN (SELECT k FROM U WHERE k IS NOT NULL)` to leave the `NULL` out *(since v26.11.1 for a sub-query; before, a `NULL` row of the sub-query was ignored)*.

Arithmetic with a `NULL` operand is `NULL`, so `WHERE age - 4 < 6` does not return records without `age` either (see Math Operators).

Two values of one kind that has no order, such as two records, cannot be compared with `<`, `<=`, `>`, `>=` or `BETWEEN`: the statement fails with `Cannot compare a record with a record using '<': the values have no order`, a client error (`400` over HTTP) *(since v26.12.1, before it was `500`)*. `>=` and `<=` still answer true when both sides are the very same record.

Mathematics Operators

Apply to	Operator	Description	Example
Numbers	+	Plus	`age + 34`
Numbers	-	Minus	`salary - 34`
Numbers	*	Multiply	`factor * 1.3`
Numbers	/	Divide	`total / 12`
Numbers	%	Mod	`total % 3`

Methods

Also called "Field Operators", are SQL-Methods.

Variables

ArcadeDB supports variables managed in the context of the command/query. By default, some variables are created. Below the table with the available variables:

Name	Description	Command(s)
`$parent`	Get the parent context from a sub-query. Example: `SELECT FROM V LET $type = (TRAVERSE out() FROM $parent.$current.children)`	SELECT and TRAVERSE
`$current`	Current record to use in sub-queries to refer from the parent's variable	SELECT and TRAVERSE
`$depth`	The number of hops from the root to the current record along `$path`. It is the shortest distance only with `STRATEGY BREADTH_FIRST`, see Traversal Strategies	TRAVERSE
`$path`	The list of the Record IDs the traversal followed from the root to the current record. Example: `[#6:0, #5:0]`. You can also display it with `SELECT $path FROM (TRAVERSE out() FROM V)`	TRAVERSE
`$stack`	The same Record IDs as `$path`, in reverse order: the current record first, the root last	TRAVERSE

To set custom variable use the LET keyword.

Wildcards

Symbol	Description	Example
`%`	Matches all strings that contain an unknown substring of any length at the position of `%`	"%DB" "A%DB" "Arcade%" all match "ArcadeDB"
`?`	Matches all strings that contain an unknown character at the position of `?`	"N?SQL" matches "NoSQL" but not "NewSQL"

Filtering for strings containing wildcards characters can be done by escaping with backslash, i.e. `\%`, `\?`.

14.22. SQL Script

SQL Script (`sqlscript`) is ArcadeDB's server-side scripting engine. A script is a sequence of SQL statements — including the regular DML/DDL commands plus a small set of script-only constructs (transactions, control flow, variables, locks) — that the server parses and runs as a single unit.

14.22.1. Why use SQL Script

Single-statement SQL is fine for ad-hoc queries, but most real workloads (especially graph workloads) need to do several things together: look up a vertex, create another, connect them with an edge, then return the result. Doing that as separate client calls has two problems: each call pays a network round-trip, and there is no transaction across them.

SQL Script solves both:

- **One round-trip instead of many.** The whole script is sent to the server once. A graph mutation that takes 5 commands no longer pays 5x the latency — the server runs them locally and sends back one response.
- **Implicit transaction.** A script is automatically wrapped in a transaction (unless you open one yourself with `BEGIN`), so it is all-or-nothing for records: any failure rolls back every record change in the script. Schema changes are the exception, see below.
- **Automatic retry on conflict.** Wrap the body in `BEGIN ⋯ COMMIT RETRY <n>` and ArcadeDB will retry the transaction on a `NeedRetryException` that can clear on its own (typically a concurrent-modification or transient lock-timeout). A refusal that would repeat on every attempt, such as an HA follower refusing leader-only work and naming the leader, is raised at once instead (see COMMIT — commit, retry, fallback). This is what lets you run the same script in parallel from many clients without writing your own retry loop — the server resolves the contention.
- **Server-side variables and control flow.** `LET`, `IF`, `WHILE`, `FOREACH`, `RETURN`, etc. let you express small pieces of imperative logic without leaving SQL.

A typical use case is a graph upsert: find or create a vertex, then attach an edge to it, possibly conditionally. With SQL Script that is one network call and one transaction; the server handles concurrency.

A script is a transaction. The atomicity guarantee — "all-or-nothing" — applies to the records a script changes: if any statement fails (and is not caught by a retry/else clause), every record change made by the script is rolled back.

Schema DDL (`CREATE TYPE`, `CREATE PROPERTY`, `CREATE INDEX`, `ALTER ⋯`, `DROP ⋯`) is not transactional and is never rolled back. If a script fails part way, the schema statements that ran before the failure stay applied, and the error is returned. Do not write a migration script that assumes a failed run leaves the schema untouched: make each statement safe to run again (for example with `IF NOT EXISTS`) and rerun the script. See reference/storage.pdf.

14.22.2. Schema-only scripts run as one schema change

A script of two or more statements that are **all** schema definition DDL runs as a single schema change: the schema file is written once for the whole script and, under High Availability, the batch is replicated as one Raft entry instead of one per statement. The database write lock is held for the whole script, so keep such scripts to schema definition and out of data loads. Before it takes the lock, the script lets the asynchronous workers (`database.async()`) commit what they have queued and pauses them until the script ends, so an index the script builds never waits on a pending async write. If a statement fails, the statements before it are kept and published to the cluster as one change, so every node agrees on the same partially applied schema, and the original error is returned. Set the database setting `schemaBulkDDLScript` to `false` to go back to one schema change per statement.

A script that contains any statement other than schema definition runs statement by statement, as before. That includes DML and these DDL statements, which work on records or files rather than definitions:

- `REBUILD INDEX`, `REBUILD TYPE`, `REBUILD GRAPH ANALYTICAL VIEW`
- `TRUNCATE TYPE`, `TRUNCATE BUCKET`
- `REFRESH MATERIALIZED VIEW`, `REFRESH CONTINUOUS AGGREGATE`
- `COMPACT INDEX`
- `CREATE MATERIALIZED VIEW`, `CREATE CONTINUOUS AGGREGATE`
- the other graph analytical view statements
- `ALTER TYPE … WITH repartition = true`
- `CREATE INDEX` on a type that existed before the script started

For a **bulk** load of vertices and edges (as opposed to the occasional multi-statement upsert above), `sqlscript` is not the fastest path: each `CREATE VERTEX`/`CREATE EDGE` statement is parsed and planned individually. The `/api/v1/batch` HTTP endpoint and the `GraphBatch` Java API are purpose-built for this and measure several times faster on large imports.

14.22.3. Differences between `sql` and `sqlscript`

Feature	`sql` **(single command)**	`sqlscript` **(batch)**
Number of statements	Exactly one	One or more, separated by `;`
Transaction	Not transactional by itself	Implicit transaction (or explicit `BEGIN`/`COMMIT`)
Variables	Only inside `SELECT … LET`	`LET` and `SET GLOBAL` across statements
Control flow	None	`IF`, `WHILE`, `FOREACH`, `BREAK`, `RETURN`

Feature	`sql` (single command)	`sqlscript` (batch)
Explicit transaction control	Not allowed	`BEGIN`, `COMMIT [RETRY … ELSE …]`, `ROLLBACK`
Retry on conflict	No	`COMMIT RETRY <n>`
Locks	No	`LOCK TYPE` / `LOCK BUCKET`

To pick the engine, pass `"sqlscript"` as the language parameter (Java driver, HTTP `language` field, console `LANGUAGE` setting, etc.).

14.22.4. Syntax reference

A SQL Script supports every SQL command plus the following script-only statements.

Statement	Purpose
`BEGIN [ISOLATION <level>]`	Open an explicit transaction (`READ_COMMITTED` or `REPEATABLE_READ`).
`COMMIT [RETRY <n> [ELSE { … } [AND] (FAIL|CONTINUE)]]`	Commit the transaction; optionally retry on conflict and run a fallback block.
`ROLLBACK`	Abort the current transaction.
`LOCK TYPE|BUCKET <name>[, <name>]*`	Take an explicit type-level or bucket-level lock for the rest of the transaction.
`LET <var> = <SQL or expression>`	Bind the result of a statement or expression to a script variable. Reference it as `$var`.
`SET GLOBAL <var> = <expression>`	Set a database-scoped variable that survives across scripts and queries until the database is closed.
`IF (<cond>) { … } [ELSE { … }]`	Conditional execution.
`FOREACH (<var> IN <expression>) { … }`	Loop over the items of a collection or query result.
`WHILE (<cond>) { … }`	Loop while a boolean condition holds.
`BREAK`	Exit the innermost `FOREACH` or `WHILE`.
`RETURN [<expression>]`	Stop the script and return a value (literal, variable, query, array or map).
`SLEEP <ms>`	Pause the script for the given number of milliseconds.
`CONSOLE.(log|output|error|warn|debug) <expression>`	Write a message to the server log or stdout/stderr (server-side only).

Statements are separated by `;`. Inside `IF`/`WHILE`/`FOREACH` blocks the trailing `;` after each inner statement is optional. Keywords are case insensitive.

14.22.5. BEGIN — start an explicit transaction

```
BEGIN [ISOLATION <READ_COMMITTED | REPEATABLE_READ>]
```

By default a script runs inside an implicit `READ_COMMITTED` transaction. Use `BEGIN` only when you want to set a different isolation level, or to mark the start of a `BEGIN ⋯ COMMIT RETRY` block.

- `READ_COMMITTED` (default): each `SELECT` sees the latest committed data.
- `REPEATABLE_READ`: subsequent reads of the same record return the same content for the lifetime of the transaction, and so does `count(*)` on a type *(since v26.11.1)*. It is not a snapshot of the whole database: see Isolation, and `LOCK TYPE` for a reader that needs one committed state of a whole type.

```
BEGIN ISOLATION REPEATABLE_READ;
LET v1 = SELECT FROM Account WHERE id = 1;
-- ... other work ...
LET v2 = SELECT FROM Account WHERE id = 1;  -- guaranteed to match v1
COMMIT;
```

Transactions nest up to a limit (3 by default). A `BEGIN` inside a loop that never commits goes past it, and the script fails with `Cannot execute BEGIN: Exceeded number of 3 nested transactions⋯`, a client error (`400` over HTTP) *(since v26.12.1, before it was `500`)*.

14.22.6. COMMIT — commit, retry, fallback

```
COMMIT [RETRY <n> [ELSE { <statements> } [AND] (FAIL | CONTINUE)]]
```

- Without arguments, `COMMIT` simply commits the current transaction.
- `RETRY <n>` makes the server re-run the whole `BEGIN ⋯ COMMIT` block on a retryable failure, up to `n` times. Retryable failures include `NeedRetryException` (concurrent modification, MVCC conflicts) and transient transaction-lock timeouts. A `DuplicatedKeyException` is retried once, as by the Java API's `db.transaction()`: for example an `UPSERT` that lost a race to a concurrent insert of the same key finds the committed record on the second attempt *(since v26.11.1)*. Between attempts the server waits a random delay capped by `arcadedb.txRetryDelay` (default 100 ms).
- A refusal that holds on this server until something outside it changes is not retried: an HA follower that refuses leader-only work (a schema change, a security mutation) and names the leader raises its `ServerIsNotTheLeaderException` on the first attempt, without running `ELSE` and without `AND CONTINUE` suppressing it, so the client can send the work to the leader. A refusal raised during an election, which names no leader, is still retried. *(Since v26.10.2)*
- `ELSE { ⋯ }` is the fallback block executed when retries are exhausted.
- `AND FAIL` (the default when `ELSE` is omitted) re-throws the original exception after the fallback runs.
- `AND CONTINUE` swallows the exception and lets the script keep going.

- A `COMMIT` with no transaction open fails with `Found COMMIT statement without a BEGIN`, a client error (`400` over HTTP). A top-level `COMMIT` is refused before the script runs; one inside an `IF`, `WHILE` or `FOREACH` is refused when it is reached, because only then is it known whether a transaction is open *(since v26.12.1, before the nested form answered* `Transaction not begun` *with* `500`*)*.

This is the building block that makes server-side scripts safe to run from many concurrent clients: each conflict is resolved on the server, and the client only sees the final result.

A retry re-runs the block from the beginning, so it is only safe while the block has published nothing yet. A statement with an explicit batch boundary — `UPDATE`, `DELETE` or `MOVE VERTEX` with `BATCH <n>` — commits as it goes, so part of the block is already stored by the time a later statement fails, and a rollback cannot take it back.

When that has happened the block is **not** retried: the conflict is raised to the caller, who is the only one able to compensate for the part that is already stored. In that case the `ELSE` block does not run and `AND CONTINUE` does not suppress the error either — `ELSE` statements are written for a block that did nothing, and returning an empty result would hide a conflict you need to see. A block that committed nothing before failing is still retried the full `<n>` times, as before.

Examples:

```
-- retry up to 100 times, fail if still conflicting
BEGIN;
LET account = CREATE VERTEX Account SET name = 'Luke';
LET city    = SELECT FROM City WHERE name = 'London';
LET e       = CREATE EDGE Lives FROM $account TO $city;
COMMIT RETRY 100;
RETURN $e;
```

```
-- retry, otherwise log to a dead-letter type and continue
BEGIN;
UPDATE Account SET balance = balance - 100 WHERE id = 1;
UPDATE Account SET balance = balance + 100 WHERE id = 2;
COMMIT RETRY 10 ELSE {
  INSERT INTO TransferDLQ SET fromId = 1, toId = 2, amount = 100, ts = sysdate();
} AND CONTINUE;
```

14.22.7. LOCK — explicit type or bucket lock

```
LOCK TYPE   <type-name>  [, <type-name>]*
LOCK BUCKET <bucket-name>[, <bucket-name>]*
```

`LOCK` takes an exclusive lock on one or more types or buckets for the rest of the current transaction. The lock is released automatically on `COMMIT` or `ROLLBACK`.

Use it when an MVCC retry would not be enough — typically for a "read-modify-write" sequence that must see a consistent snapshot of an entire type, or for serializing work on a hot bucket.

```
BEGIN;
LOCK TYPE Account, Audit;                          -- nobody else can modify Account or
Audit until we commit
LET total = SELECT sum(balance) FROM Account;
INSERT INTO Audit SET kind = 'snapshot', total = $total[0].sum, ts = sysdate();
COMMIT;
```

A `LOCK` has to come before the first modification of the transaction, and only one is allowed per transaction. A `LOCK` after a modification (or after the script ended its transaction), or a second `LOCK`, fails with `Cannot execute LOCK TYPE …: Explicit lock must be acquired before any modification` (or `Explicit lock already acquired`), a client error (`400` over HTTP) *(since v26.12.1, before it was `500`)*. A lock that cannot be taken in time is still a retryable conflict.

Every type written in the transaction has to be locked. Committing a transaction that modified a type (or bucket) that was not locked fails with `Cannot commit transaction because not all the modified resources were locked`. In the example above `Audit` is written, so it is listed in `LOCK TYPE` together with `Account`.

`LOCK TYPE` also locks every subtype, and for a vertex type every file its vertices write: the edge-list buckets and the super-node stripe buckets. *(Since v26.10.2: adding an edge to a vertex of a subtype, or to a super-node, under `LOCK TYPE` failed the commit with the error above.)*

```
BEGIN;
LOCK BUCKET orders_2026_q2;                        -- serialize writes on a hot bucket
INSERT INTO Order BUCKET orders_2026_q2 SET ...;
COMMIT;
```

14.22.8. LET — script variables

```
LET <var> = <expression>
LET <var> = <SQL statement>
LET <var1> = <expr1>, <var2> = <expr2>, ...
```

`LET` evaluates the right-hand side and binds the result to a variable. Reference it later with `$var`. The right-hand side can be:

- an expression (literal, arithmetic, function call): `LET total = 0;`
- a SQL command: `LET city = SELECT FROM City WHERE name = 'London';`
- multiple bindings in one statement, separated by commas.

A `LET` whose value is a `SELECT` acts as an ephemeral view — the result is materialized into a list and can be referenced repeatedly without re-running the query.

`LET` variables have dynamic scope: they are visible to every statement that runs after them in the same script (and inside nested blocks).

```
LET hubs = SELECT FROM City WHERE category = 'hub';
LET count = $hubs.size();
INSERT INTO Stat SET name = 'hub-count', value = $count;
RETURN $count;
```

Dynamic typing. Type names in `SELECT`, `INSERT`, `CREATE`, etc. can come from a variable — but only when the statement is itself the right-hand side of a `LET`:

```
LET mytype = 'vec';
LET r = SELECT FROM $mytype WHERE num > 0;
RETURN $r;
```

```
LET myedge = 'Lives';
LET r = CREATE EDGE $myedge FROM #1:0 TO #1:1;
RETURN $r;
```

14.22.9. SET GLOBAL — database-scoped variables

```
SET GLOBAL <var> = <BOOLEAN | STRING | NUMBER | LIST | MAP>
```

`SET GLOBAL` stores a transient variable on the database itself. It is visible to every subsequent SQL or Cypher query running against that database, until the database is closed or the variable is set to `NULL`. Use it for small bits of session-like state (a feature flag, a tenant id, a timestamp).

```
SET GLOBAL tenant = 'acme';
-- later, in any query on this database:
SELECT FROM Account WHERE tenantId = $tenant;
```

14.22.10. RETURN — end the script with a value

```
RETURN [<expression>]
```

`RETURN` ends script execution immediately. The argument can be:

- a literal: `RETURN 3;`
- a variable: `RETURN $a;`
- a query: `RETURN (SELECT FROM Foo);`
- an array (HTTP only): `RETURN [ $a, $b ];`

- a map (HTTP only): `RETURN { 'first': $a, 'second': $b };`
- nothing — exits the script with an empty result.

To return arrays or maps in a way that works on every protocol and driver (Java, Node.js, etc.), wrap the result in a `SELECT`:

```
RETURN (SELECT $a AS first, $b AS second)
```

To return an empty result use `RETURN []`.

14.22.11. IF — conditional execution

```
IF (<sql-predicate>) {
  <statement>;
  <statement>;
  ...
} [ELSE {
  <statement>;
  ...
}]
```

`<sql-predicate>` is any boolean expression that would be valid in a `WHERE` clause.

```
LET existing = SELECT FROM User WHERE email = 'luke@example.com';
IF ($existing.size() = 0) {
  INSERT INTO User SET email = 'luke@example.com', createdAt = sysdate();
} ELSE {
  UPDATE User SET lastSeen = sysdate() WHERE email = 'luke@example.com';
}
```

```
IF ($a.size() > 0) {
  ROLLBACK;
}
```

14.22.12. FOREACH — iterate a collection

```
FOREACH (<variable> IN <expression>) {
  <statement>;
  <statement>;
  ...
}
```

The expression must evaluate to a collection: a literal array, a variable, or the result of a sub-query.

```
FOREACH ($i IN [1, 2, 3]) {
  INSERT INTO Foo SET value = $i;
}
```

```
-- bulk-create edges from a query result
LET people = SELECT FROM Person WHERE city = 'London';
FOREACH ($p IN $people) {
  CREATE EDGE LivesIn FROM $p TO (SELECT FROM City WHERE name = 'London');
}
```

14.22.13. WHILE — loop on a condition

```
WHILE (<condition>) {
  <statement>;
  <statement>;
  ...
}
```

```
LET $i = 0;
WHILE ($i < 10) {
  INSERT INTO Foo SET value = $i;
  LET $i = $i + 1;
}
```

14.22.14. BREAK — exit the current loop

`BREAK` exits the innermost `FOREACH` or `WHILE`.

```
FOREACH ($i IN [1, 2, 3, 4, 5]) {
  IF ($i > 2) {
    BREAK;
  }
  CONSOLE.log $i;
}
```

```
-- sum positive amounts until you hit a negative one
LET total = 0;
FOREACH ($record IN (SELECT FROM CustomerAccount)) {
  IF ($record.amount < 0) {
    BREAK;
  }
  LET total = $total + $record.amount;
}
```

```
RETURN $total;
```

14.22.15. SLEEP — pause execution

```
SLEEP <milliseconds>
```

Pauses the script for the given number of milliseconds. Mostly useful for tests, throttling, or staggering retries.

The argument has to be a number. `SLEEP` of `null`, a string or a list fails with `Invalid value for SLEEP: … SLEEP takes a number of milliseconds`, a client error (`400` over HTTP) *(since v26.12.1, before it was `500`)*.

```
FOREACH ($url IN $urls) {
  INSERT INTO FetchJob SET url = $url, status = 'queued';
  SLEEP 50;                                    -- be polite to a downstream system
}
```

14.22.16. CONSOLE — log from a script

```
CONSOLE.log    <expression>
CONSOLE.output <expression>
CONSOLE.error  <expression>
CONSOLE.warn   <expression>
CONSOLE.debug  <expression>
```

Writes a message from the server side. `log`/`warn`/`debug` go to the server log at the matching level, `output` to stdout, `error` to stderr (and the log at SEVERE level). Useful for debugging long-running or scheduled scripts.

Any other level (`CONSOLE.nope`) is refused when the script is parsed, before any of its statements runs, with `Unsupported CONSOLE level 'nope': use one of log, output, error, warn, debug`, a client error (`400` over HTTP) *(since v26.12.1: the script used to run up to the `CONSOLE` statement and then fail with `500`)*.

```
LET count = SELECT count(*) AS c FROM Order WHERE status = 'pending';
IF ($count[0].c > 1000) {
  CONSOLE.warn 'Pending order backlog is ' + $count[0].c;
}
```

14.22.17. End-to-end example: a graph upsert with retry

This is the canonical use case for SQL Script: a single script that **upserts** two vertices (insert if missing, update if present), idempotently connects them with an edge, retries on conflict, and

returns the result. Many clients can run this in parallel against the same server — each conflict triggers a server-side retry, no client-side bookkeeping required.

The script uses the `UPDATE … UPSERT` clause for the vertices. `UPSERT` is atomic only when the `WHERE` condition matches a `UNIQUE` index, so the schema needs those indexes first (one-time setup):

```
CREATE VERTEX TYPE City   IF NOT EXISTS;
CREATE VERTEX TYPE Account IF NOT EXISTS;
CREATE EDGE   TYPE Lives   IF NOT EXISTS;

CREATE INDEX IF NOT EXISTS ON City    (name)  UNIQUE;
CREATE INDEX IF NOT EXISTS ON Account (email) UNIQUE;
```

The upsert script itself:

```
BEGIN;

-- 1. Upsert the city. UNIQUE index on City.name makes this atomic.
--    `RETURN AFTER` returns the upserted row directly, so we capture
--    the vertex (and its RID) in $city without a second roundtrip.
LET city    = UPDATE City SET name = 'London', country = 'UK'
              UPSERT RETURN AFTER WHERE name = 'London';

-- 2. Upsert the person. UNIQUE index on Account.email makes this atomic.
LET account = UPDATE Account SET name = 'Luke', email = 'luke@example.com',
                                 updatedAt = sysdate()
              UPSERT RETURN AFTER WHERE email = 'luke@example.com';

-- 3. Idempotently connect them. CREATE EDGE has no UPSERT clause,
--    so check for an existing edge first.
LET existing = SELECT FROM Lives
               WHERE @out = $account[0] AND @in = $city[0];
IF ($existing.size() = 0) {
  CREATE EDGE Lives FROM $account TO $city SET since = sysdate();
}

COMMIT RETRY 100;

RETURN (SELECT FROM Lives
        WHERE @out = $account[0] AND @in = $city[0]);
```

What makes this a **real** upsert:

- **Vertices** — `UPDATE … UPSERT WHERE` either updates the existing row matched by the unique key or inserts a new one. The `RETURN AFTER` clause makes the statement emit the upserted row (with `@rid`) regardless of which branch was taken, so we capture it into a `LET` variable in one statement instead of issuing a separate `SELECT` lookup. Run the script twice with the same `email`/`name` and you still end up with exactly one `Account` and one `City`.

- **Edge** — `CREATE EDGE` does not have its own `UPSERT` clause, so the script does the equivalent by hand: a `SELECT` against the edge's `@out`/`@in` endpoints, then `CREATE EDGE` only when the lookup is empty. Re-running the script does not produce duplicate edges.
- **Concurrency** — the whole `BEGIN … COMMIT RETRY 100` block is the unit that gets retried. If two clients run this script at the same time and one of them hits a `NeedRetryException` (typically from the unique-index conflict), the server transparently rolls back and re-executes the block up to 100 times.

The result of the final `SELECT` (the existing or newly-created `Lives` edge) is what the client receives.

14.22.18. Running scripts

You can run a SQL script from any of the supported entry points:

- **Java driver**: `database.command("sqlscript", "BEGIN; … COMMIT;")`.
- **HTTP API**: `POST /api/v1/command/<db>` with `{"language": "sqlscript", "command": "…"}`.
- **Console**: switch the language with `set language sqlscript;` then paste the script.
- **Studio**: pick "SQL Script" in the language selector before running.

In an HA cluster, scripts are routed to the leader so that the implicit transaction can be applied with quorum.

14.23. Functions

SQL Functions are all the functions bundled with ArcadeDB SQL engine. Look also to SQL Methods.

Extended Functions Available

ArcadeDB provides 100+ additional functions organized by namespace (`text.*`, `map.*`, `convert.*`, `date.*`, `util.*`, `agg.*`, etc.). These functions are available in both SQL and Cypher queries. See reference/extended-functions.pdf for the complete reference.

These extended functions are also APOC-compatible, supporting the `apoc.` prefix for seamless Neo4j migration.

The functions `expand()` and `distinct()` are special functions, as those have constraints in terms of admissible use inside projections.

SQL Functions can work in 2 ways based on the fact that they can receive one or more parameters:

14.23.1. Aggregated mode

When only one parameter is passed, the function aggregates the result in only one record. The classic example is the `sum()` function:

```
SELECT sum(salary) FROM employee
```

This will always return one record: the sum of salary fields across every employee record.

14.23.2. Inline mode

When two or more parameters are passed:

```
SELECT sum(salary, extra, benefits) AS total FROM employee
```

This will return the sum of the field "salary", "extra" and "benefits" as "total".

In case you need to use a function inline, when you only have one parameter, then add "null" as the second parameter:

```
SELECT first(out('friends').name, NULL) AS firstFriend FROM Profiles
```

In the above example, the `first()` function doesn't aggregate everything in only one record, but rather returns one record per `Profile`, where the `firstFriend` is the first item of the collection received as the parameter.

14.23.3. Function Reference

abs() math

Returns the absolute value. It works with Integer, Long, Short, Double, Float, BigInteger, BigDecimal, and null.

Syntax: `abs(<field>)`

Examples

```
SELECT abs(score) FROM Account
```

```
SELECT abs(-2332) FROM Account
```

```
SELECT abs(999) FROM Account
```

---

astar() graph

The A* algorithm describes how to find the cheapest path from one node to another node in a

directed weighted graph with a heuristic function.

The first parameter is source record. The second parameter is destination record. The third parameter is a name of property that represents weight and fourth represents the map of options.

An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule. The resulting path is returned as list of RIDs.

When several edges connect the same pair of vertices, the cheapest one is used.

Syntax: `astar(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName>, [<options>])`

options:

```
{
  direction:"OUT", //the edge direction (OUT, IN, BOTH)
  edgeTypeNames:[],
  vertexAxisNames:[],
  parallel : false,
  tieBreaker:true,
  maxDepth:99999,
  dFactor:1.0,
  customHeuristicFormula:'custom_Function_Name_here'  // (MANHATTAN, MAXAXIS,
DIAGONAL, EUCLIDEAN, EUCLIDEANNOSQR, CUSTOM)
}
```

`edgeTypeNames` and `vertexAxisNames` accept either a list (`['Road']`) or a comma-separated string (`'Road,Rail'`). Any other type is rejected with an error.

The boolean options (`parallel`, `tieBreaker`, `emptyIfMaxDepth`) accept a boolean or the strings `'true'` and `'false'` (in any case), and `maxDepth` a non-negative whole number (a long, so `3000000000` is valid). Any other value, such as `'yes'`, `1` or `'ture'` for a boolean option, or `2.5`, `'abc'` or `-1` for `maxDepth`, is rejected with an error that names the option. *(Since v26.11.1. Before, a boolean option read any other value as* `false`*.)*

Every key of the options map can be written unquoted, `parallel` included: `{ parallel: true }`. *(Since v26.11.1; before,* `parallel` *was a syntax error unless quoted, see map literals.)*

`maxDepth` is the number of edges a returned path may have, the same bound `shortestPath()` and the Cypher `*..n` use: the result is the cheapest path of at most `maxDepth` edges, which can be a dearer path than the cheapest one overall when that one is longer than the bound. When there is no such path the result is an empty list. A path is returned only when it runs from the source to the destination.

Before this rule `maxDepth` counted the vertices the search had examined, not the edges of the path. On a graph that branches, `astar()` could return a path that stopped short of the destination, and `dijkstra()` could answer an empty list for a

> path that existed. Both now answer by the number of edges. The `emptyIfMaxDepth` option is still accepted, but it no longer changes the answer: a path that does not end at the destination is never returned. The `depth` a `customHeuristicFormula` function receives is the same count, the number of edges between the source and the vertex being estimated, where it used to be the number of vertices examined so far.

`DIAGONAL` is the octile distance (a diagonal step costs `sqrt(2) * dFactor`), admissible on 8-connected grids. The two-axis `tieBreaker` penalizes a node's distance from the straight line between source and destination.

Examples

```
SELECT astar($current, #8:10, 'weight') FROM Vehicle

SELECT astar($current, #8:10, 'weight', { direction: 'OUT', parallel: true, maxDepth:
20 }) FROM Vehicle
```

See also: Graph Algorithms: A* for the Cypher procedure variant with geographic heuristics.

---

avg() math

Returns the arithmetic mean. The mean of integer values is a `DOUBLE`, even when their sum is too large for a `LONG`; the mean of `DECIMAL` values is a `DECIMAL`.

Syntax: `avg(<field|value> [,<field|value>]*)`

Examples

```
SELECT avg(salary) FROM Account
```

```
SELECT avg(1, 2, 3)
```

---

bool_and() misc

Aggregates a field, an expression or value by the logical AND operator and returns true or false, null values are ignored.

Syntax: `bool_and(<field|expression|value>)`

Examples

Test if all salaries are greater than zero:

```
SELECT bool_and((salary > 0)) FROM Account
```

bool_or() misc

Aggregates a field, an expression or value by the logical OR operator and returns true or false, null values are ignored.

Syntax: `bool_or(<field|expression|value>)`

Examples

Test if a null value is present in the salary field:

```
SELECT bool_or((salary IS NULL)) FROM Account
```

both() graph

Get the adjacent outgoing and incoming vertices starting from the current record as vertex.

Syntax: `both([<label1>] [,<label-n>]*)`

Examples

Get all the incoming and outgoing vertices from vertex with RID `#13:33`:

```
SELECT both() FROM #13:33
```

Get all the incoming and outgoing "Vehicle" vertices connected by edges with label (class) "Trucks" and "Cars":

```
SELECT both('Trucks', 'Cars') FROM Vehicle
```

bothE() graph

Get the adjacent outgoing and incoming edges starting from the current record as vertex.

Syntax: `bothE([<label1>] [,<label-n>]*)`

Examples

Get both incoming and outgoing edges from all the "Vehicle" vertices:

```
SELECT bothE() FROM Vehicle
```

Get all the incoming and outgoing edges of type "Friend" from the profiles with "nickname" "Jay"

```
SELECT bothE('Friend') FROM Profile WHERE nickname = 'Jay'
```

---

bothV() graph

Get the adjacent outgoing and incoming vertices starting from the current record as edge.

Syntax: `bothV()`

Examples

Get both incoming and outgoing vertices from the "Friend" edges:

```
SELECT bothV() FROM Friend
```

---

cchShortestPath() graph

Returns the cheapest path between two vertices, answered by a Customizable Contraction Hierarchy when a Graph Analytical View keeps one for the weight property and edge types asked for, and by bidirectional Dijkstra otherwise. Direction can be OUT (default), IN or BOTH. The resulting path is returned as list of RIDs, empty when the destination cannot be reached.

An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule. Among parallel edges only the cheapest counts.

Syntax: `cchShortestPath(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName> [, <direction> | { direction, edgeTypeNames }])`

Examples

```
SELECT cchShortestPath(#12:0, #12:5, 'distance', { direction: 'OUT', edgeTypeNames:
['ROAD'] }) AS path
```

See also: algo.cch.shortestPath for the Cypher procedure variant, and reference/graph-algorithms/sql-path-functions.pdf for the full SQL function reference.

---

circle() geo

Creates a 2D circle from two numbers specifying X- and Y-coordinate of circle's center and a number describing the circle's radius.

Syntax: `circle(<center-x>, <center-y>, <radius>)`

Examples

```
SELECT circle(10, 10, 10) AS circle
```

---

coalesce() misc

Returns the first field/value argument not being null parameter. If no field/value is not null, `null` is returned.

Syntax: `coalesce(<field|value> [,<field-n|value-n>]*)`

Examples

```
SELECT coalesce(amount, amount2, amount3) FROM Account
```

---

concat() text

Aggregates field (or string) by implicitly casting to string and concatenate. Optionally, a second field or string can be passed and is record-wise appended.

Syntax: `concat(<field|string> [,<field|string>]*)`

Examples

```
SELECT concat(name) FROM names
```

---

count() math

Counts the records that match the query condition. If `*` is used as field, then all record will be counted, otherwise only records with field content that is not null.

Syntax: `count([DISTINCT] <field>)`

With `DISTINCT` each distinct value is counted once, with or without `GROUP BY`. Numbers that are equal count once whatever their type or scale (`1` and `1.0`, `19.9` and `19.90`), as in `SELECT DISTINCT` and `GROUP BY`; a zone-less instant counts once with the UTC `ZONED_DATETIME` of the same instant, under the rule

described in SELECT *(since v26.11.1)*; a null is never counted. DISTINCT is accepted by the other aggregate functions too, e.g. sum(DISTINCT <field>) and avg(DISTINCT <field>).

Examples

```
SELECT count(*) FROM Account
SELECT count(DISTINCT country) FROM Account
SELECT type, count(DISTINCT country) AS countries FROM Account GROUP BY type
```

---

cypherRID() graph

Converts a Cypher-style numeric id (the value returned by the OpenCypher id() function and the SQL .asCypherRID() method) back into a native RID. It is the explicit inverse of .asCypherRID() and shares its configurable bit split (arcadedb.opencypher.idBucketBits). Its main use is as the **target** of a SELECT/UPDATE/DELETE, where the record is then resolved in O(1) - the same cost as a native #bucketId:offset literal. A negative input is rejected, since a valid Cypher id is never negative.

Syntax: cypherRID(<cypherId>)

Examples

Resolve a record by its Cypher id (named parameter):

```
SELECT FROM cypherRID(:id) LIMIT 1
```

Use a bare numeric literal, or update/delete by Cypher id:

```
SELECT FROM cypherRID(30064771072) LIMIT 1
UPDATE cypherRID(:id) SET verified = true
DELETE FROM cypherRID(:id)
```

The value passed to cypherRID() is exactly what id() (Cypher) or .asCypherRID() (SQL) produced for that record.

---

date() date

Returns a date as DATETIME from a string. <date-as-string> is the date in string format, and <format> is the date format of <date-as-string> following these rules. If no format is specified, then the default database format is used. To know more about it, look at Managing Dates.

Syntax: date(<date-as-string> [,<format>] [,<timezone>])

If no argument is given, date() returns the current time as a UTC wall clock, like sysdate().

A text that names an instant, because the format reads an offset (`XXX`, `Z`) or a zone (`VV`, `z`) out of it, answers the UTC wall clock of that instant, as the call without a format does: `date('2020-01-02T03:04:05+02:00', "yyyy-MM-dd'T'HH:mm:ssXXX")` is `2020-01-02T01:04:05` *(since v26.11.1; before, the offset was dropped and it was `03:04:05`)*. `<timezone>` is the zone a text that names none was written in: `date('2020-01-02 03:04:05', 'yyyy-MM-dd HH:mm:ss', 'Asia/Tokyo')` is `2020-01-01T18:04:05` *(since v26.11.1; before, it was ignored)*. Without it a text with no zone keeps its wall clock, and a text that names its own offset ignores it. The same holds for `{ format: '…', timezone: '…' }`.

Examples

```
SELECT FROM Account WHERE created <= date('2012-07-02', 'yyyy-MM-dd')
```

---

decode() misc

Decodes a value into binary data (`base64` and `base64url` are the only supported formats). The `<value>` must contain base64 encoded information.

Syntax: `decode(<value>, <format>)`

The decode function returns a binary type, which can be converted to a string via `asString()`.

Examples

Decode a value into binary format from base64.

```
SELECT decode('QXJjYWRlREI=', 'base64')
```

```
SELECT decode('QXJjYWRlREI', 'base64url').asString()
```

---

difference() collections

Syntax: `difference(<field> [,<field-n>]*)`

Works as aggregate or inline. If only one argument is passed then it aggregates, otherwise it executes and returns the DIFFERENCE between the collections received as parameters. An `ARRAY_OF_*` argument is read as its elements, as a `LIST` is *(since v26.12.1; before, the whole array was one element)*.

Examples

```
SELECT difference(tags) FROM book
```

```
SELECT difference(inEdges, outEdges) FROM OGraphVertex
```

---

dijkstra() graph

Returns the cheapest path between two vertices using the Dijkstra's algorithm where the **weightEdgeFieldName** parameter is the field containing the weight. Direction can be OUT (default), IN or BOTH. The resulting path is returned as list of RIDs.

When several edges connect the same pair of vertices, the cheapest one is used. An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule.

Syntax: `dijkstra(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName> [, <direction>])`

Examples

```
SELECT dijkstra($current, #8:10, 'weight') FROM Vehicle
```

See also: Graph Algorithms: Dijkstra for the Cypher procedure variant, and reference/graph-algorithms/sql-path-functions.pdf for the full SQL function reference in the appendix.

---

distance() geo

Returns the distance between two points in the globe using the Haversine algorithm. Coordinates must be in degrees.

Syntax: `distance(<x-field>, <y-field>, <x-value>, <y-value>)`

Examples

```
SELECT FROM POI WHERE distance(x, y, 52.20472, 0.14056 ) <= 30
```

---

distinct() collections

Retrieves only unique data entries depending on the field you have specified as argument. The main difference compared to standard SQL DISTINCT is that with ArcadeDB, a function with parenthesis and only one field can be specified.

Syntax: `distinct(<field>)`

The `distinct()` function has to be the sole projection component if used.

Examples

```
SELECT distinct(name) FROM City
```

---

duration() date

Returns a Java duration object, which can be useful to compare periods of time, and to move a datetime by an amount of time (see Arithmetic with dates).

Syntax: `duration(<field|integer>, '<string>')` | `duration('<iso-8601-duration>')` | `duration(<map>)`

The two-argument form takes an amount and a time unit. The one-argument forms are the ones openCypher's `duration()` accepts, read by the same parser: an ISO-8601 duration text (`'PT1H30M'`, `'P2DT3H'`, `'P1W'`, `'PT0.5S'`, `'-PT2M'`) or a map of components (`{hours: 1, minutes: 30}`, with keys `weeks`, `days`, `hours`, `minutes`, `seconds`, `milliseconds`, `microseconds`, `nanoseconds`).

The admissible second argument values are given here. A duration is an exact amount of time, so the units with no fixed length, years and months, are refused in every form (`duration(1, 'month')`, `duration('P1M')`, `duration({years: 1})`): use days, hours, minutes, seconds or smaller.

Examples

```
SELECT FROM UserActivity WHERE sysdate() - lastActivity < duration(1, 'hour')

SELECT sysdate() - duration(7, 'day') AS aWeekAgo

SELECT name FROM Event ORDER BY dt + duration('PT1H')

SELECT dt + duration({days: 4, minutes: 6}) AS later FROM Event
```

---

duanSSSP() graph

Returns the cheapest path between two vertices, through the same engine as cchShortestPath() (a Customizable Contraction Hierarchy when a Graph Analytical View keeps one, bidirectional Dijkstra otherwise). The weight property is optional and defaults to `weight`. Direction can be OUT (default), IN or BOTH. The resulting path is returned as list of RIDs, empty when the destination cannot be reached.

An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule.

Syntax: `duanSSSP(<sourceVertex>, <destinationVertex> [, <weightEdgeFieldName>] [, <direction> | { direction, edgeTypeNames }])`

Examples

```
SELECT duanSSSP(#12:0, #12:5, 'distance', { edgeTypeNames: ['ROAD'] }) AS path
```

See also: reference/graph-algorithms/sql-path-functions.pdf for the full SQL function reference.

---

encode() misc

Encode binary data into the specified format (`base64` and `base64url` are the only supported formats). The `<binaryfields>` must be a property containing binary data.

Syntax: `encode(<binaryfield|stringfield|string>, <format>)`

To encode RIDs, they need to be converted to strings first via `asString()` otherwise the link target is encoded.

Examples

Encode binary data into base64.

```
SELECT encode(raw, 'base64') FROM Blob
```

---

expand() collections

This function has two meanings:

- When used on a collection field, it unwinds the collection in the field <field> and use it as result.
- When used on a link (RID) field, it expands the document pointed by that link.

A single non-collection, non-link value (a number, string, boolean or map) is treated as a one-element result: it produces one row, exposed as a `value` property (or the alias, if one is given).

Syntax: `expand(<field>)`

You can also use the SQL operator `UNWIND` in select to obtain the same result.

As `expand()` may change its return type based on the argument, no modifiers (method calls, suffix identifiers or array indexing) are permitted on the return value of `expand()`.

Examples

```
SELECT expand(addresses) FROM Account
```

---

first() collections

Retrieves only the first item of multi-value fields (arrays, collections and maps). For non multi-value types just returns the value.

Syntax: `first(<field>)`

Examples

```
SELECT first(addresses) FROM Account
```

format() text

Formats a value using the String.format() conventions. Look here for more information on the format string syntax.

Syntax: `format(<format> [,<arg1> ] [,<arg-n>]*)`

To escape the percent symbol (`%`) use `%%`.

Examples

```
SELECT format("%d - Mr. %s %s (%s)", id, name, surname, address) FROM Account
```

if() misc

Syntax: `if(<expression>, <result-if-true> [,<result-if-false>])`

Evaluates a condition (first parameter) and returns the second parameter if the condition is true, and the third parameter otherwise. The third parameter is optional: if it is omitted and the condition is false, `null` is returned. *(Since v26.11.1)* A numeric condition is read by its sign, without truncation: `if(0.5, a, b)` and a condition of `3000000000` return `a`, and `-3000000000` returns `b`. Before, the number was cast to an int, which dropped the fraction and wrapped a value past the int range.

Examples

```
SELECT if( (name = 'John'), "My name is John", "My name is not John") FROM Person
```

Omitting the false branch, so that non-matching records get `null`:

```
SELECT if( (name = 'John'), "My name is John") FROM Person
```

ifempty() misc

Returns the passed field/value, or optional parameter *return_value_if_not_empty*. If field/value is an empty string or collection, *return_value_if_empty* is returned.

Syntax: `ifempty(<field|value>, <return_value_if_empty> [,<return_value_if_not_empty>])`

Examples

```
SELECT ifempty(name, "No Name") FROM Account
```

ifnull() misc

Returns the passed field/value, or optional parameter *return_value_if_not_null*. If field/value is null, *return_value_if_null* is returned.

Syntax: `ifnull(<field|value>, <return_value_if_null> [,<return_value_if_not_null>])`

Examples

```
SELECT ifnull(salary, 0) FROM Account
```

in() graph

Get the adjacent incoming vertices starting from the current record as vertex.

Syntax: `in([<label-1>] [,<label-n>]*)`

Examples

Get all the incoming vertices from all the "Vehicle" vertices:

```
SELECT in() FROM Vehicle
```

Get all the incoming vertices connected with edges with label (class) "Trucks" and "Cars":

```
SELECT in('Trucks', 'Cars') FROM Vehicle
```

`in()`, `inE()`, `both()` and `bothE()` return what the vertex stores, as the Java vertex API does. An edge type declared `UNIDIRECTIONAL` stores no incoming side, so on its own `SELECT in('Follows') FROM Account` returns nothing for such a type. A hop of a SQL

> `MATCH` pattern (`{as: a}.in('Follows'){as: b}`, `{as: a}<-Follows-{as: b}`) and a Cypher pattern return the incoming edges of a unidirectional type too. An `in()` written inside a `where:` or `while:` condition of a `MATCH` is an expression, and answers as the function does anywhere else. A Graph Analytical View does not change these answers, it only makes them faster. The path-finding functions `dijkstra()`, `astar()`, `bellmanFord()`, `cchShortestPath()` and `duanSSSP()` always follow the incoming edges of a unidirectional type (since 26.10.2); see `CREATE TYPE`.

---

inE() graph

Get the adjacent incoming edges starting from the current record as Vertex.

Syntax: `inE([<label1>] [,<label-n>]*)`

Examples

Get all the incoming edges from all the "Vehicle" vertices:

```
SELECT inE() FROM Vehicle
```

Get all the incoming edges of type "Eats" from the "Restaurant" "Bella Napoli":

```
SELECT inE('Eats') FROM Restaurant WHERE name = 'Bella Napoli'
```

---

intersect() collections

Works as aggregate or inline. If only one argument is passed then it aggregates, otherwise executes and returns the INTERSECTION of the collections received as parameters.

Syntax: `intersect(<field> [,<field-n>]*)`

Examples

```
SELECT intersect(friends) FROM profile WHERE jobTitle = 'programmer'
```

```
SELECT intersect(inEdges, outEdges) FROM GraphVertex
```

---

inV() graph

Get incoming vertices starting from the current record as edge.

Syntax: `inV()`

Examples

Get incoming vertices from the "Friend" edges

```
SELECT inV() FROM Friend
```

---

last() collections

Retrieves only the last item of multi-value fields (arrays, collections and maps). For non multi-value types just returns the value.

Syntax: `last(<field>)`

Examples

```
SELECT last(addresses) FROM Account
```

---

list() collections

Creates or adds a value to a list. If `<field|value>` is a collection, then is merged with the list, otherwise `<field|value>` is added to the list.

Syntax: `list(<field|value>[,]*)`

Examples

```
SELECT name, list(roles.name) AS roles FROM User
```

---

lineString() geo

Creates a chain of 2D lines from a list of points. A string of lines is not necessarily closed.

Syntax: `lineString([<point>*])`

Examples

```
SELECT lineString([point(10, 10), point(20, 10), point(20, 20), point(10, 20), point
(30, 30)]) AS linesString
```

---

map() collections

Creates a map. The arguments have to be pairs of keys and values, hence the number of arguments has to be even. The `<key>` argument(s) have to be strings.

Syntax: `map(<key>, <value>[,]*)`

Examples

```
SELECT map(name, roles.name) FROM User
```

---

max() math

Returns the maximum value. If invoked with more than one parameter, the function doesn't aggregate, but returns the maximum value between all the arguments. A collection argument (a `LIST`, or an `ARRAY_OF_*` value such as `ARRAY_OF_INTEGERS`) is read as its elements, so `max()` answers its largest element; a `BINARY` value is one value.

Values are ordered as `ORDER BY`, the indexes and the comparison operators order them: numbers by value whatever their type, a `BINARY` value by its unsigned bytes, a `MAP` by its entries, and two Java representations of one instant (for example a `DATETIME` and a `ZONED_DATETIME` of the same moment in a schemaless property) by that instant. An `EMBEDDED` document has no order and is refused with an error naming its type, as are two values of different kinds, such as a number and a string.

(since v26.11.1) Before, `max()` and `min()` failed with "cannot compare a value of type byte[] with a value of type byte[]" (and likewise for maps and mixed temporal types) as soon as two rows reached them, and an `ARRAY_OF_*` value was compared as a whole instead of by its elements.

Syntax: `max(<field> [,<field-n>]*)`

Examples

Returns the maximum salary of all the "Account" records:

```
SELECT max(salary) FROM Account.
```

Returns the maximum value between "salary1", "salary2" and "salary3" fields.

```
SELECT max(salary1, salary2, salary3) FROM Account
```

---

median() math

Returns the middle value or an interpolated value that represent the middle value after the values

are sorted. Nulls are ignored in the calculation.

Syntax: `median(<field>)`

Examples

```
SELECT median(salary) FROM Account
```

---

min() math

Returns the minimum value. If invoked with more than one parameter, the function doesn't aggregate but returns the minimum value between all the arguments. A collection argument (a `LIST`, or an `ARRAY_OF_*` value) is read as its elements, so `min()` answers its smallest element. Values are ordered as described for max(), the order of `ORDER BY` *(since v26.11.1)*.

Syntax: `min(<field> [, <field-n>]*)`

Examples

Returns the minimum salary of all the "Account" records:

```
SELECT min(salary) FROM Account
```

Returns the minimum value between "salary1", "salary2" and "salary3" fields.

```
SELECT min(salary1, salary2, salary3) FROM Account
```

---

mode() math

Returns the values that occur with the greatest frequency. Nulls are ignored in the calculation.

Syntax: `mode(<field>)`

Examples

```
SELECT mode(salary) FROM Account
```

---

out() graph

Get the adjacent outgoing vertices starting from the current record as vertex.

Syntax: `out([<label-1>] [,<label-n>]*)`

A `null` label names no edge type and is skipped, so `out('Eats', null)` is `out('Eats')`, and labels that are all `null` follow every edge type, as `out()` does. The same holds for `in()`, `both()`, `outE()`, `inE()` and `bothE()` *(since v26.12.1; before, a `null` next to another label failed with a `NullPointerException`, a parameter bound to `null` among several labels included).*

Examples

Get all the outgoing vertices from all the "Vehicle" vertices:

```
SELECT out() FROM Vehicle
```

Get all the outgoing vertices connected with edges with label (class) "Eats" and "Favorited" from all the "Restaurant" vertices in "Rome":

```
SELECT out('Eats', 'Favorited') FROM Restaurant WHERE city = 'Rome'
```

---

outE() graph

Get the adjacent outgoing edges starting from the current record as vertex.

Syntax: `outE([<label1>] [,<label-n>]*)`

Examples

Get all the outgoing edges from all the "Vehicle" vertices:

```
SELECT outE() FROM Vehicle
```

Get all the outgoing edges of type "Eats" from all the "SocialNetworkProfile" vertices:

```
SELECT outE('Eats') FROM SocialNetworkProfile
```

---

outV() graph

Get outgoing vertices starting from the current record as edge.

Syntax: `outV()`

Examples

Get outgoing vertices from the "Friend" edges

```
SELECT outV() FROM Friend
```

---

percentile() math

Returns the nth percentiles (the values that cut off the first n percent of the field values when it is sorted in ascending order). Nulls are ignored in the calculation. The quantiles have to be in the range 0—1

Syntax: `percentile(<field> [, <quantile-n>]*)`

Examples

```
SELECT percentile(salary, 0.95) FROM Account
```

```
SELECT percentile(salary, 0.25, 0.75) AS IQR FROM Account
```

---

point() geo

Creates a 2D point from two numbers specifying X- and Y-coordinate.

Syntax: `point(<x>, <y>)`

Examples

```
SELECT point(10, 20) AS point
```

---

polygon() geo

Creates a 2D polygon from a list of points. The lines making up a polygon are closed.

Syntax: `polygon([<point>*])`

Examples

```
SELECT polygon([point(10, 10), point(20, 10), point(20, 20), point(10, 20), point(10,
10)]) AS polygon
```

---

pow() math

Returns the power of the `base` argument to the `exponent` argument. It works with Integer, Long, Short, Byte, Double, Float, BigInteger, BigDecimal, and null: a null `base` or `exponent` returns null.

(since v26.11.1): the result depends on the values of the arguments, not on the type the base is stored as:

- an integral base and a whole exponent that is not negative return the exact power, widened as the arithmetic operators widen a result: an Integer while it fits, then a Long, then an exact Decimal (`pow(2, 31)` is `2147483648`, `pow(10, 30)` is `1000000000000000000000000000000`). Before, the power was narrowed back into the type of the base (`pow(2, 31)` returned `2147483647`, and `pow(s, 3)` on a SHORT holding 300 returned `-832`)
- a Decimal base and a whole exponent return the exact power, rounded to 34 digits past 1024 digits or for a negative exponent
- a Float or Double base, a fractional exponent (`pow(2, 2.5)`, read as 2 before) or a negative exponent of an integral base (`pow(2, -1)` is `0.5`) return a Double
- a negative base is accepted (`pow(-2, 3)` is `-8`; it returned null before); only a power with no real value, a negative base to a fractional exponent, returns null
- a power no type can hold, or zero to a negative power of a Decimal, raises an arithmetic error (HTTP 400), like an overflow of the `+` and `*` operators

Syntax: `pow(<base>, <exponent>)`

Examples

```
SELECT pow(2, 10) AS oneKibi
```

---

randomInt() math

Returns an integer drawn from a uniform pseudo-random distribution in the range from (inclusively) zero up to (exclusively) the argument `max`.

Syntax: `randomInt(<max>)`

(Since v26.11.1) `max` must be a whole number inside the int range: `10`, `10.0` and `'10'` work, while `2.5`, `0.9`, `3000000000` and `'abc'` are refused with an error of their own. Before, `2.5` silently became 2.

Examples

```
SELECT randomInt(10) AS rand
```

You can use it in SQL Scripts to wait a random amount of milliseconds.

```
SLEEP randomInt(500);
```

---

rectangle() geo

Creates a 2D rectangle from four numbers: the west (left) boundary X, one boundary Y, the east (right) boundary X and the other boundary Y. The two Y values can be given in either order, so `rectangle(0, 10, 10, 0)` and `rectangle(0, 0, 10, 10)` make the same box. The X values keep their order: a west X greater than the east X makes a rectangle that crosses the antimeridian. *(Since v26.12.1; before, the first Y had to be the lower one, and the top-then-bottom order failed with "maxY must be >= minY")*

Syntax: `rectangle(<west-x>, <y1>, <east-x>, <y2>)`

Examples

```
SELECT rectangle(10, 10, 20, 20) AS rectangle
```

---

set() collections

Creates or adds a value to a set. If `<value>` is a collection, then it is merged with the set, otherwise `<field|value>` is added to the set.

Syntax: `set(<field|value>[,]*)`

Examples

```
SELECT name, set(roles.name) AS roles FROM User
```

Two numbers that are equal are one item whatever their type (`1`, `1L` and `1.0`), here and in `intersect()`, `difference()` and `symmetricDifference()`. A `DECIMAL` is compared exactly, as `DISTINCT` and `GROUP BY` compare it: `9007199254740993` and `9007199254740992` are two items. *(Since v26.12.1; before, these functions compared items by their Java type, so `1` and `1L` were two items, and openCypher `RETURN DISTINCT`, `count(DISTINCT)` and `collect(DISTINCT)` merged two decimals that a floating point number cannot tell apart)*

---

shortestPath() graph

Returns the shortest path between two vertices. Direction can be OUT (default), IN or BOTH.

Syntax: `shortestPath(<sourceVertex>, <destinationVertex> [, <direction> [, <edgeClassName> [, <additionalParams>]]])`

Where:

- `sourceVertex` is the source vertex where to start the path
- `destinationVertex` is the destination vertex where the path ends - `direction`, optional, is the direction of traversing. By default is "BOTH" (in+out). Supported values are "BOTH" (incoming and outgoing), "OUT" (outgoing) and "IN" (incoming)
- `edgeClassName`, optional, is the edge class to traverse. By default all edges are crossed. This can also be a list of edge class names (eg. `["edgeType1", "edgeType2"]`)
- `additionalParams`, optional, here you can pass a map of additional parameters (Map<String, Object> in Java, JSON from SQL). Currently allowed parameters are
- 'maxDepth': integer, maximum depth for paths (ignore path longer that 'maxDepth'). It limits the number of edges on the path, so `maxDepth: 1` finds a direct edge *(since v26.11.1; before, it counted vertices, so a path of exactly `maxDepth` edges was not found)*. It must be a non-negative whole number, and `null` means no bound *(since v26.11.1; before, `null` was read as 0)*
- 'edge': boolean, include the edges in the result. A boolean or the strings `'true'` and `'false'`; any other value is rejected with an error *(since v26.11.1; before, it was read as `false`)*

Examples

on finding the shortest path between vertices `#8:32` and `#8:10`

```
SELECT shortestPath(#8:32, #8:10)
```

Examples

on finding the shortest path between vertices `#8:32` and `#8:10` only crossing outgoing edges

```
SELECT shortestPath(#8:32, #8:10, 'OUT')
```

Examples

on finding the shortest path between vertices `` `#8:32 `` and `` `#8:10 `` only crossing incoming edges of type "Friend"

```
SELECT shortestPath(#8:32, #8:10, 'IN', 'Friend')
```

Examples

on finding the shortest path between vertices `` `#8:32 `` and `` `#8:10 `` only crossing incoming edges of type "Friend" or "Colleague"

```
SELECT shortestPath(#8:32, #8:10, 'IN', ['Friend', 'Colleague'])
```

Examples

on finding the shortest path between vertices `#8:32` and `#8:10`, long at most five hops

```
SELECT shortestPath(#8:32, #8:10, null, null, {"maxDepth": 5})
```

See also: Graph Algorithms: shortestPath() for the full function reference, and Graph Algorithms Appendix for a comprehensive reference of all graph algorithms.

---

sqrt() math

Returns the square root. It works with Integer, Long, Short, Byte, Double, Float, BigInteger, BigDecimal, and null.

Negative arguments result in null. *(since v26.11.1)*: the root of an integral or Float argument is a Double (`sqrt(2)` is `1.4142135623730951`, `sqrt(4)` is `2.0`), and that of a BigInteger or BigDecimal a Decimal rounded to 34 digits. Before, the root was rounded down into the type of the argument (`sqrt(2)` returned `1`) and a Decimal root had 10 digits.

Syntax: `sqrt(<field>)`

Examples

```
SELECT sqrt(score) FROM Account
```

```
SELECT sqrt(2.0)
```

```
SELECT sqrt(63)
```

---

stddev() math

Returns the standard deviation: the measure of how spread out values are. Nulls are ignored in the calculation.

Syntax: `stddev(<field>)`

Examples

```
SELECT stddev(salary) FROM Account
```

---

strcmpci() text

Compares two string ignoring case. Return value is -1 if first string ignoring case is less than second, 0 if strings ignoring case are equals, 1 if second string ignoring case is less than first one. Before comparison both strings are transformed to lowercase and then compared.

Syntax: `strcmpci(<first_string>, <second_string>)`

Examples

Select all records where state name ignoring case is equal to "washington"

```
SELECT * FROM State WHERE strcmpci('washington', name) = 0
```

---

sum() math

Returns the sum of all the values returned.

Integers widen instead of wrapping: a sum of `INTEGER` values that does not fit an `INTEGER` is returned as a `LONG`, and a sum of `LONG` values that does not fit a `LONG` is returned as a `DECIMAL` (`BigDecimal`). Clients that read the result as a `LONG` can see a `DECIMAL` when the total overflows.

Over an empty group, or a group where every value is `NULL`, `sum()` returns `NULL` (as `avg()`, `min()` and `max()` do), not `0`. *(Since v26.10.2)* Use `ifnull(sum(<field>), 0)` to get `0`. In OpenCypher, `sum()` still returns `0`.

Syntax: `sum(<field>)`

Examples

```
SELECT sum(salary) FROM Account
```

---

symmetricDifference() collections

Works as aggregate or inline. If only one argument is passed then it aggregates, otherwise executes and returns the SYMMETRIC DIFFERENCE between the collections received as parameters. An `ARRAY_OF_*` argument is read as its elements, as a `LIST` is, in both forms *(since v26.12.1; before, the whole array was one element)*.

Syntax: `symmetricDifference(<field> [,<field-n>]*)`

Examples

```
SELECT symmetricDifference(tags) FROM book
```

```
SELECT symmetricDifference(inEdges, outEdges) FROM GraphVertex
```

---

sysdate() date

Returns the current date time as DateTime object. To know more about it, look at Managing Dates.

The clock is read as an instant, so the value stored in a `DATETIME` property is the real current moment whatever the time zone of the JVM (a `DATETIME` is stored as a UTC wall clock). With a zone argument, `sysdate('Asia/Seoul')`, the clock is read in that zone.

The default format is controlled by the setting `arcadedb.dateFormat`.

Syntax: `sysdate([<zoneid>])`

Examples

```
SELECT sysdate() FROM Account
```

---

unionall() collections

Works as aggregate or inline. If only one argument is passed then aggregates, otherwise executes and returns a UNION of all the collections received as parameters. Also works with no collection values.

Syntax: `unionall(<field> [,<field-n>]*)`

Examples

```
SELECT unionall(friends) FROM profile
```

```
SELECT unionall(inEdges, outEdges) FROM GraphVertex WHERE label = 'test'
```

---

uuid() misc

Generates a UUID as a 128-bits value using the Leach-Salz variant. For more information look at: http://docs.oracle.com/javase/6/docs/api/java/util/UUID.html.

Syntax: `uuid()`

Examples

Insert a new record with an automatic generated id:

```
INSERT INTO Account SET id = UUID()
```

variance() math

Returns the middle variance: the average of the squared differences from the mean. Nulls are ignored in the calculation.

Syntax: `variance(<field>)`

Examples

```
SELECT variance(salary) FROM Account
```

Vector Function Namespace

Every vector function has two equivalent names: the namespaced `vector.<name>()` form and a camelCase alias (`vectorXXX()`) kept for backward compatibility. Both resolve to the same function, are case-insensitive, and can be called without back-ticks.

Use the namespaced `vector.<name>()` form in new queries: it is consistent with the other extended namespaces (`geo.*`, `math.*`, ...) and reads more clearly. The full reference is in reference/extended-functions.pdf. The headings below keep the legacy `vectorXXX()` spelling for continuity; the mapping is:

Legacy alias	Recommended (namespaced)
`vectorNeighbors()`	`vector.neighbors()`
`vectorDimension()`	`vector.dimension()`
`vectorNormalize()`	`vector.normalize()`
`vectorL2Distance()`	`vector.l2Distance()`
`vectorCosineSimilarity()`	`vector.cosineSimilarity()`
`vectorDotProduct()`	`vector.dotProduct()`
`vectorMagnitude()`	`vector.magnitude()`
`vectorL2Norm()`	`vector.l2Norm()` (alias of `vector.magnitude()`)
`vectorAdd()`	`vector.add()`
`vectorSubtract()`	`vector.subtract()`
`vectorScale()`	`vector.scale()`
`vectorClamp()`	`vector.clamp()` (alias of `vector.clip()`)

Legacy alias	Recommended (namespaced)
`vectorQuantizeInt8()`	`vector.quantizeInt8()`
`vectorQuantizeBinary()`	`vector.quantizeBinary()`

NULL and empty vectors

The vector functions follow one rule on degenerate arguments *(since v26.11.1)*:

- A `NULL` vector argument returns `NULL`, the SQL convention, so a row whose embedding is not set yet (a type being backfilled) answers `NULL` instead of failing the whole query. `vectorDimension(null)` returns `0`. The arguments of a search (`vector.neighbors()`, `vector.sparseNeighbors()`, `vector.recommend()`, `vector.discover()`, `vector.rerank()`), such as the index name and the query vector, are still required.
- An empty vector returns what the function returns on a zero vector: `0` for a norm or a distance (`vectorL1Norm([])`, `vectorL2Norm([])` and `vectorLInfNorm([])` are all `0`), `false` for a predicate (`vectorHasNaN()`, `vectorHasInf()`, `vectorIsNormalized()`), an empty vector for an element-wise operation (`vectorNormalize()`, `vectorClip()`) and `[]` for `vectorToString()`. Only the functions with no value on it refuse it: `vectorCosineSimilarity()`, `vectorVariance()`, `vectorStdDev()`, `vectorSparsity()` and the quantizations.

vectorAdd() vector

Returns vector of element-wise sums of argument vectors, which are expected to be of the same size. Alternatively, when one argument is a scalar the scalar is added to every element (broadcasting).

Syntax: `vectorAdd(<vector|field>, <vector|field|scalar>)`

Examples

```
SELECT vectorAdd([1.0, 2.0, 3.0], [2.0, 3.0, 4.0])

SELECT vectorAdd([1.0, 2.0, 3.0], 4.0)
-- Returns: [5.0, 6.0, 7.0]
```

---

vectorApproxDistance() vector

Computes approximate distance between quantized vectors without intermediate full dequantization. When the arguments are the result objects of `vectorQuantizeInt8()` /`vectorQuantizeBinary()` the mode is inferred and the third argument can be omitted; raw byte arrays still need the explicit mode. Two modes are supported:

- `'INT8'` L2 distance computed on the int8 codes without materialising the dequantized vectors.

Each `vectorQuantizeInt8()` result carries its own `min`/`max` band, and every code is mapped back through its own band, so the answer approximates the true L2 distance (within the quantization step) whatever the scale and offset of the two vectors. Two raw int8 byte arrays carry no band: they must have been quantized on one shared scale, and their distance is measured in code units. A result object and a raw array cannot be compared and are rejected.

- `'BINARY'` very fast (count differing bits), reduces operations eight-fold, returns normalized Hamming distance. Requires the `vectorQuantizeBinary()` result object (raw packed bits are not enough). Each bit is drawn against that vector's own median, so the distance compares the shape of the two vectors and ignores their scale and offset: a cosine-like proxy, not an L2 one

Syntax: `vectorApproxDistance(<quantized>, <quantized> [, <string>])`

Examples

```
SELECT vectorApproxDistance(vectorQuantizeInt8([1.0, 2.0, 3.0]), vectorQuantizeInt8([
1.0, 3.0, 3.0]))
```

---

vectorAvg() vector

Aggregate function computing element-wise arithmetic average of vectors.

Syntax: `vectorAvg(<field>)`

Examples

```
SELECT vectorAvg(myvector) FROM doc
```

---

vectorClip() vector

Forces every vector element to the range between min and max arguments, where min less or equal to max is assumed. Elements exceeding this ranged are set to min or max respectively. Also available as `vectorClamp()` (the standard Java/Math term).

Syntax: `vectorClip(<vector|field>, <scalar>, <scalar>)`

Examples

```
SELECT vectorClip([1.0, 5.0, 10.0], 2.0, 8.0)

SELECT vectorClamp([1.0, 5.0, 10.0], 2.0, 8.0)
-- Returns: [2.0, 5.0, 8.0]  (clamp is an alias of clip)
```

---

vectorCosineSimilarity() vector

Computes the cosine similarity between two vectors. Returns a value between -1 and 1, where 1 means same proportional, 0 means orthogonal, and -1 means opposite vectors.

Syntax: `vector(<vector|field>, <vector|field>)`

Examples

```
SELECT vectorCosineSimilarity([1.0, 2.0, 3.0], [2.0, 2.0, 3.0])
```

---

vectorDenseToSparse() vector

converts a dense to a sparse vector representation, whereas a sparse element is determined by the optional threshold argument which is exact zero (0.0) by default.

Syntax: `vectorDenseToSparse(<vector|field>)`

Examples

```
SELECT vectorDenseToSparse([0.5, 0.0, 0.1], 0.2)
```

---

vectorDequantizeInt8() vector

Dequantized byte array to float vector using min-max scaling, recovering original vector only approximately. Pass the result of `vectorQuantizeInt8()` directly (min/max read from it), or supply the bytes with explicit `min`/`max`. *Since v26.11.1*, codes supplied as a list of numbers must be integers between -128 and 127: any other value (such as `200` or `1.5`) is reported as an error instead of being silently wrapped or truncated.

Syntax: `vectorDequantizeInt8(<result>)` or `vectorDequantizeInt8(<bytes>, <min>, <max>)`

Examples

```
SELECT vectorDequantizeInt8(vectorQuantizeInt8([1.0, 2.0, 3.0]))
```

---

vectorDimension() vector

Returns the dimension of argument vector, being the length of its underlying array. A `NULL` argument returns `0` (consistent with `.size()`/`.length()`).

Syntax: `vectorDimension(<vector|field>)`

Examples

```
SELECT vectorDimension([1.0, 2.0, 3.0])

SELECT vectorDimension(null)
-- Returns: 0
```

---

vectorDotProduct() vector

Computes standard dot product (inner product) between two argument vectors.

Syntax: `vectorDotProduct(<vector|field>, <vector|field>)`

Examples

```
SELECT vectorDotProduct([1.0, 0.0, 1.0], [0.0, 1.0, 1.0])
```

---

vectorHasInf() vector

Returns true if one or more elements in argument vector are positive or negative infinity. A `NULL` element is treated as NaN (not infinite) and does not trigger a true result.

Syntax: `vectorHasInf(<vector|field>)`

```
SELECT vectorHasInf([1.0, 10e400, 2.0])
```

---

vectorHasNaN() vector

Returns true if one or more elements in argument vector are NaN (not-a-number), a special value in IEEE floating point numbers. A `NULL` element (e.g. from an invalid math op such as `sqrt(-1.0)` coerced to `NULL` inside a collection) is treated as NaN.

Syntax: `vectorHasNaN(<vector|field>)`

```
SELECT vectorHasNaN([1.0, sqrt(-1.0), 3.0])
-- Returns: true
```

---

vectorHasNull() vector

Returns true if one or more elements in argument vector are `NULL`. Unlike `vectorHasNaN()`, this

distinguishes a genuine `NULL` (missing element) from a NaN float value. Primitive array inputs can never hold a `NULL` and therefore always return false.

Syntax: `vectorHasNull(<vector|field>)`

```
SELECT vectorHasNull([1.0, sqrt(-1.0), 3.0])
-- Returns: true
```

---

vectorHybridScore() vector

Computes weighted average of two numbers with argument weight alpha.

Syntax: `vectorHybridScore(<scalar>, <scalar>, <scalar>)`

Examples

```
SELECT vectorHybridScore(1.0, 2.0, 0.5)
```

---

vectorIsNormalized() vector

Checks if the argument vector is normalized, meaning if its length (L2 norm) is one, up to an optional threshold argument, which is taken as `0.001` by default.

Syntax: `vectorIsNormalized(<vector|field> [, <tolerance>])`

Examples

```
SELECT vectorIsNormalized([1.0, 0.0, 1.0], 1.0)
```

---

vectorL1Norm() vector

Computes L1 norm (Manhattan norm) of argument `vector`.

Syntax: `vectorL1Norm(<vector|field>)`

Examples

```
SELECT vectorL1Norm([1.0, 2.0, 3.0])
```

---

vectorL2Distance() vector

Computes the L2 distance (Euclidean distance) between two argument vectors.

Syntax: `vectorL2Distance(<vector|field>, <vector|field>)`

Examples

```
SELECT vectorL2Distance([1.0, 2.0, 3.0], [2.0, 2.0, 3.0])
```

---

vectorLInfNorm() vector

Computes L∞-norm (Maximum norm) of argument vector.

Syntax: `vectorLInfNorm(<vector|field>)`

Examples

```
SELECT vectorLInfNorm([1.0, 2.0, 3.0])
```

---

vectorMagnitude() vector

Computes L2 norm (Euclidean norm) of argument vector. Also available as `vectorL2Norm()`, symmetric with `vectorL1Norm()` and `vectorLInfNorm()`: the three return `NULL` for a `NULL` vector and `0` for an empty one (see NULL and empty vectors).

Syntax: `vectorMagnitude(<vector|field>)`

Examples

```
SELECT vectorMagnitude([1.0, 2.0, 3.0])

SELECT vectorL2Norm([3.0, 4.0])
-- Returns: 5.0  (l2Norm is an alias of magnitude)
```

---

vectorMax() vector

Aggregate function computing element-wise maximum of vectors.

Syntax: `vectorMax(<field>)`

Examples

```
SELECT vectorMax(myvector) FROM doc
```

vectorMin() vector

Aggregate function computing element-wise minimum of vectors.

Syntax: `vectorMin(<field>)`

Examples

```
SELECT vectorMin(myvector) FROM doc
```

vectorMultiScore() vector

Combines multiple scores form different vectors using a fusion method. The scores are assumed to be represented in a vector, the fusion method is selected by a string. The weights vector is only required for the weighted average method and expected to be of the same size as the score vector. Available fusion methods:

- `MAX` maximum score (ColBERT style)
- `AVG` arithmetic average of scores
- `MIN` minimum score
- `WEIGHTED` weighted average

Syntax: `vectorMultiScore(<vector>, <string> [, <vector>])`

Examples

```
SELECT vectorMultiScore([0.9, 0.7, 0.8], 'MAX')
```

vectorNeighbors() vector

Returns the k nearest neighbors from a vector index. When an index is defined on a parent type with inheritance, you can search across all types (using the parent type name) or filter to a specific child type (using the child type name).

Syntax: `vectorNeighbors(<index-spec>, <vector|key>, <k>)`

Where:

- `<index-spec>` - Index specification in the format `'TypeName[propertyName]'`. Use the parent type to

search all records, or a child type to search only that type's records.

- `<vector|key>` - Query vector (array of floats) or a key identifier to look up the vector.
- `<k>` - Number of nearest neighbors to return.

Each row reads its record on the first property asked of it (since v26.11.1), so `SELECT @rid, @type, distance FROM (SELECT expand(vectorNeighbors(…)))` reads no document. `@type` answers the neighbour's type there, in a projection, `WHERE` and `ORDER BY` alike (before v26.11.1 it answered null on these rows). See Returning Only RIDs and Distances for when the record is still read up front, and for what a row answers when its record was removed behind the index's back.

Examples

Basic usage:

```
SELECT vectorNeighbors('Document[embedding]', [0.1, 0.2, 0.3], 5)
```

Type-Specific Search with Inheritance

When a vector index is created on a parent type, sub-indexes are automatically created for child types. You can search within a specific child type by specifying the child type name:

```
-- Schema: EMBEDDING (parent) with children EMBEDDING_IMAGE, EMBEDDING_DOCUMENT
-- Index created on: EMBEDDING.vector

-- Search ONLY in EMBEDDING_IMAGE records
SELECT vectorNeighbors('EMBEDDING_IMAGE[vector]', :queryVector, 10) AS neighbors
FROM EMBEDDING_IMAGE LIMIT 1

-- Search across ALL types (parent + all children)
SELECT vectorNeighbors('EMBEDDING[vector]', :queryVector, 10) AS neighbors
FROM EMBEDDING LIMIT 1
```

---

vectorNormalize() vector

Normalizes vector to unit length (1.0), in the L2 norm.

Syntax: `vectorNormalize(<vector|field>)`

Examples

```
SELECT vectorNormalize([1.0, 2.0, 3.0])
```

---

vectorNormalizeScores() vector

Normalizes a vector to unit range ([0, 1]) using min-max normalization.

Syntax: `vectorNormalizeScores(<vector>)`

Examples

```
SELECT vectorNormalizeScores([1.0, 2.0, 3.0])
```

---

vectorQuantizeInt8() vector

Quantize argument vector to byte elements (8 bit per dimension) using min-max scaling. The result carries the vector's own `min` and `max`, which `vectorDequantizeInt8()` and `vectorApproxDistance()` apply. Every element must be finite: since 26.10.2 a `NaN` or an infinite value is rejected with an error, because it has no position inside a min-max band.

Syntax: `vectorQuantizeInt8(<vector|field>)`

Examples

```
SELECT vectorQuantizeInt8([0.1, 0.5, 0.9])
```

---

vectorQuantizeBinary() vector

Quantize argument vector to binary elements (1 bit per dimension) using median threshold. The median is used (rather than a fixed value like 0) so the bits split ~50/50, maximizing the information retained per bit for arbitrarily-centered embeddings.

Syntax: `vectorQuantizeBinary(<vector|field>)`

Examples

```
SELECT vectorQuantizeBinary([0.1, 0.5, 0.9])
```

---

vectorDequantizeBinary() vector

Reconstructs an approximate float vector from a binary-quantized result. Lossy: binary quantization keeps only the sign relative to the median, so each bit maps to `highValue` (default `1.0`) when set or `lowValue` (default `-1.0`) when clear. Accepts the result of `vectorQuantizeBinary()` directly.

Syntax: `vectorDequantizeBinary(<quantized> [, <lowValue>, <highValue>])`

Examples

```
SELECT vectorDequantizeBinary(vectorQuantizeBinary([0.1, 0.5, 0.9]))
-- Returns: [-1.0, 1.0, 1.0]
```

---

vectorRRFScore() vector

Compute reciprocal rank fusion (RRF) for combining multiple ranks. Ranks may be passed as variadic arguments or grouped in a single array/list. Every positional numeric argument is a rank. The center constant `k` (default `60`) is set only via a trailing options map `{ k: <long> }` - a bare trailing number is always a rank, never `k`. `k` must not be negative: a negative value is refused with an error *(since v26.12.1)*.

Syntax: `vectorRRFScore(<rank>, <rank>, ⋯ [, { k }])` or `vectorRRFScore([<ranks>], [{ k }])`

Examples

```
SELECT vectorRRFScore(1, 2, 4)

SELECT vectorRRFScore([1, 2, 4], { k: 100 })
```

---

vectorScale() vector

Scales argument vector element-wise by argument scalar.

Syntax: `vectorScale(<vector|field>, <scalar|field>)`

Examples

```
SELECT vectorScale([1.0, 2.0, 3.0], 4.0)
```

---

vectorScoreTransform() vector

Transform score to reshape score distribution. Supported transformations:

- `'LINEAR'` no transformation
- `'SIGMOID'` S-shaped curve, see Logistic function
- `'TANH'` hyperbolic tangent, S-shaped curve to (-1, 1)
- `'LOG'` / `'LN'` natural logarithm (requires a positive score); `LN` is the clearer synonym
- `'EXP'` exponential function

Syntax: `vectorScoreTransform(<scalar>, <string>)`

Examples

```
SELECT vectorScoreTransform(0.5, 'SIGMOID')

SELECT vectorScoreTransform(0.5, 'TANH')

SELECT vectorScoreTransform(2.5, 'LN')
```

---

vectorSparseCreate() vector

Create sparse vector from indices and value arrays. By default the dimension is inferred form the index array. Optionally, the dimension can be set via the dimension argument.

Syntax: `vectorSparseCreate(<vector|field>, <vector|field> [, <integer>])`

(Since v26.11.1) The dimension, when given, must be a whole positive number in the int range. Every index must be a whole number, not negative and below the dimension. A fractional, out-of-range, zero or negative dimension, a bad index, and a null index element are refused with an error that names the function, instead of being cast or raising a raw `IndexOutOfBoundsException` or `NullPointerException`. The indices of `vector.sparseNeighbors` are checked the same way.

Examples

```
SELECT vectorSparseCreate([0, 2, 5], [0.5, 0.3, 0.8], 7)
```

---

vectorSparseDot() vector

Computes dot product (inner product) between two sparse vectors.

Syntax: `vectorSparseDot(<sparsevector>, <sparsevector>)`

Examples

```
SELECT vectorSparseDot(vectorDenseToSparse([1.0, 2.0, 3.0]), vectorDenseToSparse([1.0,
1.0, 1.0]))
```

---

vectorSparseToDense() vector

Converts a sparse vector to dense vector representation.

Syntax: `vectorSparseToDense(<sparsevector>)`

Examples

```
SELECT vectorSparseToDense(vectorSparseCreate([0, 2], [0.5, 0.3]))
```

---

vectorSparsity() vector

Returns a sparsity measure of an argument vector. The threshold is optional and defaults to `sqrt(eps)` for float (~3.45e-4). An optional third argument selects the measure: `'FRACTION'` (default, fraction of elements with `|x| < threshold`), `'L0'` (the L0 pseudonorm, count of elements with `|x| >= threshold`), or `'GMEAN'` (geometric mean of absolute values, `0` if any element is `0`).

Syntax: `vectorSparsity(<vector> [, <scalar> [, <string>]])`

Examples

```
SELECT vectorSparsity([0.01, 0.1, 0.05, 0.02], 0.06)
-- Returns: 0.75

SELECT vectorSparsity([0.0, 1.0, 2.0, 3.0])
-- Returns: 0.25  (uses the default threshold)

SELECT vectorSparsity([0.0, 1.0, 2.0, 3.0], 0.5, 'L0')
-- Returns: 3

SELECT vectorSparsity([1.0, 2.0, 4.0], 0.0, 'GMEAN')
-- Returns: 2.0
```

---

vectorStdDev() vector

Computes the standard deviation (square-root of variance) of an argument vector.

Syntax: `vectorStdDev(<vector|field>)`

Examples

```
SELECT vectorStdDev([1.0, 2.0, 3.0])
```

---

vectorSubtract() vector

Returns vector of element-wise differences of argument vectors, which are expected to be of the same size. Alternatively, when one argument is a scalar it is broadcast against the vector, preserving order: `vector - scalar` subtracts the scalar from each element, while `scalar - vector`

subtracts each element from the scalar.

Syntax: `vectorSubtract(<vector|field|scalar>, <vector|field|scalar>)`

Examples

```
SELECT vectorSubtract([1.0, 2.0, 3.0], [2.0, 3.0, 4.0])

SELECT vectorSubtract([1.0, 2.0, 3.0], 1.0)
-- Returns: [0.0, 1.0, 2.0]

SELECT vectorSubtract(10.0, [1.0, 2.0, 3.0])
-- Returns: [9.0, 8.0, 7.0]
```

---

vectorSum() vector

Aggregate function computing element-wise sum of vectors.

Syntax: `vectorSum(<field>)`

Examples

```
SELECT vectorSum(myvector) FROM doc
```

---

vectorToString() vector

Convert argument vector to string representation in various formats. Equivalent to the `asString()` method on a vector value (e.g. `embedding.asString('PYTHON')`); both share the same formatter.

- `'COMPACT'` Single line, i.e. `"[1.0, 2.0, 3.0]"` (default)
- `'PRETTY'` Multi-line with formatting
- `'PYTHON'` Python list format
- `'MATLAB'` space-separated row vector `[1.0 2.0 3.0]`
- `'MATLAB_COLUMN'` semicolon-separated column vector `[1.0; 2.0; 3.0]`
- `'JULIA'` Julia vector literal `[1.0, 2.0, 3.0]`
- `'NUMPY'` bare comma-separated `1.0, 2.0, 3.0` (no brackets), for `numpy.fromstring(…, sep=",")`

Syntax: `vectorToString(<vector|field> [, <string>])`

Examples

```
SELECT vectorToString([0.5, 0.25, 0.75])
```

```
SELECT vectorToString([0.5, 0.25, 0.75], 'NUMPY')
-- Returns: 0.5, 0.25, 0.75
```

---

vectorVariance() vector

Computes the variance (average spread) of an argument vector.

Syntax: `vectorVariance(<vector|field>)`

Examples

```
SELECT vectorVariance([1.0, 2.0, 3.0])
```

---

version() misc

Returns the ArcadeDB version number and build as string.

Syntax: `version()`

Examples

```
SELECT version()
```

Full-Text Function Namespace

The four full-text search functions live under the `fulltext.*` namespace. The original snake_case names remain available as aliases for backward compatibility.

Legacy alias	Canonical name
`search_fields()`	`fulltext.searchFields()`
`search_index()`	`fulltext.searchIndex()`
`search_fields_more()`	`fulltext.searchFieldsMore()`
`search_index_more()`	`fulltext.searchIndexMore()`

Both forms dispatch to the same implementation. The namespaced form is recommended for new code.

fulltext.searchIndex() search

Searches a named full-text index using Lucene query syntax. Returns `true` for matching documents;

use `$score` to rank results. See how-to/data-modeling/full-text-index.pdf for full documentation, Lucene syntax, and examples.

Syntax: `fulltext.searchIndex(indexName, query)` (alias: `search_index`)

Examples

```
SELECT title, $score
FROM Article
WHERE `fulltext.searchIndex`('Article[content]', '+java +spring')
ORDER BY $score DESC
```

---

fulltext.searchFields() search

Searches a full-text index resolved automatically from field names, restricted to those fields: on an index over more properties, the unqualified clauses of the query are searched in the given fields only. Equivalent to `fulltext.searchIndex` with the clauses qualified by those fields, but without needing to know the index name. See how-to/data-modeling/full-text-index.pdf for full documentation.

Syntax: `fulltext.searchFields(fieldNames, query)` (alias: `search_fields`)

Examples

```
SELECT title, $score
FROM Article
WHERE `fulltext.searchFields`(['title', 'body'], 'database tutorial')
ORDER BY $score DESC
```

---

fulltext.searchIndexMore() search

Finds documents similar to one or more source documents ("More Like This"). Use `$similarity` (0.0-1.0) to rank results. See how-to/data-modeling/full-text-index.pdf for full documentation including configuration options.

Syntax:

```
fulltext.searchIndexMore(indexName, sourceRIDs)
fulltext.searchIndexMore(indexName, sourceRIDs, { <option>: <value>, ... })
```

Alias: `search_index_more`.

The optional trailing argument is a map of tuning options. Accepted keys: `minTermFreq`, `minDocFreq`, `maxDocFreqPercent`, `maxQueryTerms`, `minWordLen`, `maxWordLen`, `boostByScore`, `excludeSource`, `maxSourceDocs`.

Unknown keys are rejected with a descriptive error.

Examples

```
SELECT title, $similarity
FROM Article
WHERE `fulltext.searchIndexMore`('Article[content]', [#10:3], { minTermFreq: 1,
boostByScore: true })
ORDER BY $similarity DESC
LIMIT 5
```

---

fulltext.searchFieldsMore() search

Same as `fulltext.searchIndexMore` but resolves the full-text index automatically from field names. See how-to/data-modeling/full-text-index.pdf for full documentation.

Syntax:

```
fulltext.searchFieldsMore(fieldNames, sourceRIDs)
fulltext.searchFieldsMore(fieldNames, sourceRIDs, { <option>: <value>, ... })
```

Alias: `search_fields_more`. Options map keys are the same as `fulltext.searchIndexMore`.

Examples

```
SELECT title, $similarity
FROM Article
WHERE `fulltext.searchFieldsMore`(['content'], [#10:3], { minTermFreq: 1,
excludeSource: true })
ORDER BY $similarity DESC
```

14.24. Methods

SQL Methods are similar to SQL Functions but they apply to values. In Object-Oriented paradigm they are called "methods", as functions related to a type. So what's the difference between a function and a method?

This is a SQL Function:

```
SELECT FROM sum( salary ) FROM employee
```

This is a SQL method:

```
SELECT FROM salary.asJSON() FROM employee
```

As you can see the method is executed against a field/value. Methods can receive parameters, like functions. You can concatenate N operators in sequence.

`<name>.<method>(⋯)` is read as a call of the namespaced function `<name>.<method>` when `<name>` is one of the function namespaces (`ts`, `geo`, `text`, `math`, `convert`, `date`, `util`, `coll`, `map`, `agg`, `node`, `rel`, `path`, `create`), even if `<name>` is a property of the record. Do not name properties like a namespace when you call methods on them (`date.asLong()` fails with `Unknown function name 'date.asLong'`); use another property name, such as `created`. Enclosing the name in backticks works for some of them (e.g. `date`), but not for all (`map` still fails).

methods are case-insensitive.

A method called with an argument it is not defined for - a value of the wrong kind, a time unit, time zone, hash algorithm or format pattern nothing names, a text that is not JSON for `asJSON()` - fails as the caller's error: HTTP 400, and the client-error category of every other protocol. Before 26.12.1 `format()`, `precision()`, `hash()` and `asJSON()` / `toJSON()` answered some of these as an internal error (HTTP 500).

14.24.1. Method Reference

[]

Execute an expression against the item. An item can be a multi-value object like a map, a list, an array or a document. For documents and maps, the item must be a string. For lists and arrays, the index is a number.

Syntax: `<value>[<expression>]`

Applies to the following types:

- document,
- map,
- list,
- array

Examples

Get the item with key "phone" in a map:

```
SELECT FROM Profile WHERE '+39' IN contacts[phone].left(3)
```

Get the first 10 tags of posts:

```
SELECT FROM tags[0...9] FROM Posts
```

---

append()

Appends one or more string to the result. A `null` argument is skipped, wherever it appears, and the remaining arguments are still appended: `'x'.append(null, 'A', 'B')` returns `xAB`. A `null` receiver returns `null`.

Syntax: `<value>.append(<value>[,<value>]*)`

Applies to the following types:

- string

Examples

```
SELECT name.append(' ').append(surname) FROM Employee
```

---

asBoolean()

Transforms the field into a Boolean type. If the origin type is a string, then `"true"` (or `"TRUE"`, `"True"`, ...) means `TRUE`, all other strings transform to `FALSE`. If it is a number type, then `0` means `FALSE` while all other numbers transform to `TRUE`. In case a `NULL` is passed then `NULL` is also returned.

Syntax: `<value>.asBoolean()`

Applies to the following types:

- string,
- short,
- int,
- long

Examples

```
SELECT FROM Users WHERE online.asBoolean() = true
```

---

asByte()

Transforms the field into a Byte type.

Syntax: `<value>.asByte()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row is kept and the query does not fail.

Applies to the following types:

- short,
- int,
- long,
- float,
- double

---

asCypherRID()

Converts a record or RID into the numeric (`Long`) form returned by the native OpenCypher `id()` function, so SQL queries can produce values that round-trip with Cypher. The two components of an ArcadeDB RID (`bucketId`, `position`) are packed into a single non-negative number using 63 of the 64 bits (the sign bit is kept clear). The split between bucketId and position is controlled by the global setting `arcadedb.opencypher.idBucketBits` (default `16`, see Settings); the same value must be used to encode and decode, and the method fails with an error if a component does not fit the reserved bits.

Syntax: `<value>.asCypherRID()`

Applies to the following types:

- record,
- RID,
- string (in the `#bucketId:position` form),
- number (returned unchanged)

Examples

```
SELECT @rid.asCypherRID() AS ident FROM Person WHERE name = 'Alice'
```

To resolve a record back from a Cypher id, use the inverse `cypherRID()` function, e.g. `SELECT FROM cypherRID(:id)`.

---

asDate()

Transforms the field into a Date type.

Syntax: `<value>.asDate([<format>])`

Where:

- `format`, optional, is the format of the date to convert if the value is a string

Applies to the following types:

- string,
- long

Examples

Returns all the records where time is before the year 2010:

```
SELECT FROM Log WHERE time.asDate() < '01-01-2010'
```

---

asDateTime()

Transforms the field into a Date type but parsing also the time information.

Syntax: `<value>.asDateTime([<format>])`

Where:

- `format`, optional, is the format of the date to convert if the value is a string

When the string names an instant, because the format reads an offset (`XXX`, `Z`) or a zone (`VV`, `z`) out of it, the result is the UTC wall clock of that instant, whether or not a format is named: `'2020-01-02T03:04:05+02:00'.asDateTime("yyyy-MM-dd'T'HH:mm:ssXXX")` is `2020-01-02T01:04:05` *(since v26.11.1; before, a named format dropped the offset and it was* `03:04:05`*)*. `asDate()` keeps the day written.

Applies to the following types:

- string,
- long

Examples

Time is stored as long type measuring milliseconds since a particular day. Returns all the records where time is before the year 2010:

```
SELECT FROM Log WHERE time.asDateTime() < '01-01-2010 00:00:00'
```

This example returns the dates stored as strings following the ISO 8601 format:

```
SELECT timeAsString.asDateTime("yyyy-MM-dd'T'HH:mm:ss'Z'") AS time FROM Log
```

---

asDecimal()

Transforms the field into an Decimal type. Use Decimal type when treat currencies.

Syntax: `<value>.asDecimal()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row is kept and the query does not fail.

Applies to the following types:

- any

Examples

```
SELECT salary.asDecimal() FROM Employee
```

---

asDouble()

Transforms the field into a double type.

Syntax: `<value>.asDouble()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row is kept and the query does not fail.

Applies to the following types:

- any

Examples

```
SELECT (ray.asDouble() > 3.14) AS bigger FROM Rays
```

---

asFloat()

Transforms the field into a float type.

Syntax: `<value>.asFloat()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row

is kept and the query does not fail.

Applies to the following types:

- any

Examples

```
SELECT (ray.asFloat() > 3.14) AS bigger FROM Rays
```

---

asInteger()

Transforms the field into an integer type.

Syntax: `<value>.asInteger()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row is kept and the query does not fail.

Applies to the following types:

- any

Float values are rounded towards zero (truncated).

Examples

Converts the first 3 chars of 'value' field in an integer:

```
SELECT value.left(3).asInteger() FROM Log
```

---

asJSON()

Returns the record(s) in JSON format. If it's executed on a result set, then the result set is completely browsed and the result set iteration exhausted (if browsed again, it will contain no records). If you need to access to the result set multiple times, transform it into a list with `asList()` and work with the list instead.

Syntax: `<value>.asJSON()`

The method takes no arguments.

Applies to the following types:

- record

Examples

```
CREATE VERTEX TYPE Test

INSERT INTO Test CONTENT {"attr1": "value 1", "attr2": "value 2"}

SELECT @this.asJSON() FROM Test
```

---

asList()

Transforms the value in a List. A collection, an array (for example a list parameter) or an iterable becomes a list of its elements. If it's a single item, a new list is created.

Syntax: `<value>.asList()`

Applies to the following types:

- any

Examples

```
SELECT tags.asList() FROM Friend
```

---

asLong()

Transforms the field into a Long type.

Syntax: `<value>.asLong()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row is kept and the query does not fail.

Applies to the following types:

- any

Examples

```
SELECT created.asLong() FROM Log
```

---

asMap()

Transforms the value in a Map where even items are the keys and odd items are values.

Syntax: `<value>.asMap()`

Applies to the following types:

- collections

Examples

```
SELECT tags.asMap() FROM Friend
```

---

asRecord()

Transforms the field into the linked record type

Syntax: `<value>.asRecord()`

Applies to the following types:

- link
- string

Examples

Transform link to a record:

```
SELECT "#1:0".asRecord()
```

---

asRID()

Transforms the field into an RID link type.

Syntax: `<value>.asRID()`

Applies to the following types:

- string

Examples

Transform string holding an RID to a link:

```
SELECT "#1:0".asRID()
```

---

asSet()

Transforms the value in a Set. If it's a single item, a new set is created. Sets do not allow duplicates.

Syntax: `<value>.asSet()`

Applies to the following types:

- any

Examples

```
SELECT tags.asSet() FROM Friend
```

Two numbers that are equal are one item, whatever their type, as with the set() function: `[1, 1L, 1.0].asSet()` holds one item. A `DECIMAL` is compared exactly. *(Since v26.12.1; before, the items were compared by their Java type, so `[1, 1L, 1.0].asSet()` held three)*

---

asShort()

Transforms the field into a short type.

Syntax: `<value>.asShort()`

A `null` value, an empty string, a string of only whitespace, or a text that is not a number of this type (for example `'abc'`, or `'300'` for a byte) returns `null`. The row is kept and the query does not fail.

Applies to the following types:

- any

Float values are rounded towards zero (truncated).

Examples

Converts the first 3 chars of 'value' field in a short integer:

```
SELECT value.left(3).asShort() FROM Log
```

---

asString()

Transforms the field into a string type.

For numeric vectors (a `float[]`/`double[]` field, or an array/list of numbers) it produces the same output as `vector.toString()`. An optional format argument selects the layout: `'COMPACT'` (default), `'PRETTY'`, `'PYTHON'`, `'MATLAB'` (space-separated row), `'MATLAB_COLUMN'` (semicolon-separated column), `'JULIA'`, or `'NUMPY'` (bare comma-separated for `numpy.fromstring`). Non-vector values are returned in their natural string form. A non-numeric array is rendered as the list of its elements (`['a','b']`), exactly like a list. The inverse (string to vector) is `asVector()`.

Syntax: `<value>.asString([<format>])`

Applies to the following types:

- any

Examples

Get all the salaries with decimals:

```
SELECT (salary.asString().indexof('.') > -1) AS hasDecimals FROM Employee
```

Render an embedding vector field in different formats:

```
SELECT embedding.asString() FROM docs
-- e.g. [0.5, 0.25, 0.75]

SELECT embedding.asString('PYTHON') FROM docs

SELECT embedding.asString('NUMPY') FROM docs
-- e.g. 0.5, 0.25, 0.75   (no brackets, for numpy.fromstring)
```

---

asVector()

Parses a value into a vector (`float[]`). This is the inverse of `asString()` for vectors and accepts every format `asString()` / `vector.toString()` can emit: bracketed or bare, comma-separated (COMPACT/PYTHON/JULIA/NUMPY), space-separated (MATLAB), semicolon-separated (MATLAB_COLUMN) and multi-line (PRETTY). It also accepts an array/list of numbers (normalized to `float[]`) and a single number (wrapped into a one-element vector).

Syntax: `<value>.asVector()`

Applies to the following types:

- string
- array / list of numbers

- number

Examples

```
SELECT '[1.0, 2.0, 3.0]'.asVector()
-- Returns: [1.0, 2.0, 3.0]

SELECT '[1.0 2.0 3.0]'.asVector()
-- Returns: [1.0, 2.0, 3.0]   (MATLAB form)

SELECT '1.0, 2.0, 3.0'.asVector()
-- Returns: [1.0, 2.0, 3.0]   (NUMPY form)

-- Round-trips with asString():
SELECT embedding.asString('MATLAB').asVector() FROM docs
```

---

asSparse()

Converts a dense vector value into a sparse vector, dropping elements whose absolute value is at or below an optional threshold (default `0.0`, i.e. drops exact zeros). The method form of `vector.denseToSparse()`, consistent with `asString()` / `asVector()`.

Syntax: `<value>.asSparse([<threshold>])`

Applies to the following types:

- array / list of numbers

Examples

```
SELECT [0.5, 0.0, 0.3].asSparse()
-- sparse vector with indices [0, 2] and values [0.5, 0.3]

SELECT embedding.asSparse(0.01) FROM docs
-- drops values with |value| <= 0.01
```

---

capitalize()

Returns a string where each word (group of characters bounded by whitespace) is transformed such that its first character (if it is a letter) is converted to upper case, and the remaining characters (that are letters) are converted to lower case.

Syntax: `<value>.capitalize()`

Applies to the following types:

- string

Examples

Return a capitalized string:

```
SELECT 'hi there'.capitalize()
```

---

charAt()

Returns the character of the string contained in the position 'position'. 'position' starts from 0 to string length.

Syntax: `<value>.charAt(<position>)`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- string

Examples

Get the first character of the users' name:

```
SELECT FROM User WHERE name.charAt( 0 ) = 'L'
```

---

convert()

Convert a value to another type.

Syntax: `<value>.convert(<type>)`

Applies to the following types:

- any

Examples

```
SELECT dob.convert( 'date' ) FROM User
```

---

exclude()

Answers a map with every property of the value except the excluded ones.

Syntax: `<value>.exclude(<field-name>[,]*)`

Applies to the following types:

- record (document, vertex or edge)
- map
- result row
- collection of the above (each element is projected, and the answer is a list of maps)

The answer is always a map, never a record. It carries no `@rid`, `@type` or `@cat` unless one of the names is a metadata field: `exclude('@cat', 'secret')` answers `@rid`, `@type` (and `@in`/`@out` on an edge) together with the properties. Because the answer is a value, `UPDATE ⋯ SET copy = @this.exclude('secret')` stores it as an embedded map, and a chained access such as `@this.exclude('secret').name` reads the map. A graph traversal like `.out()` needs the record, so call it before projecting: `out().exclude('secret')`, not `exclude('secret').out()`.

(Since v26.10.2.)

Examples

```
SELECT expand( @this.exclude( 'password' ) ) FROM User
```

You can specify a wildcard as ending character to exclude all the fields that start with a certain string. Example to exclude all the outgoing and incoming edges:

```
SELECT expand( @this.exclude( 'out_*', 'in_*' ) ) FROM V
```

This function can be used to remove internal properties like `@rid`, `@type`, etc.

```
SELECT @this.exclude('@*') FROM doc
```

On a row of a query (a sub-query result, for example) `exclude()` and `include()` read the row, not the record behind it: a column the projection overrode answers its projected value, and one it excluded (`!@rid` included) stays out. Before v26.10.2 they read the record, so `(SELECT *, !m FROM D1)[0].include('m')` answered the stored `m`.

---

field()

Returns a sub-property from an embedded document property. This method is useful for dynamic property names of types.

Syntax: `<property>.field(<string>)`

Applies to the following types:

- embedded document

Examples

Returns a field named `test` from the embedded document property `embdoc`:

```
SELECT embdoc.field('test') FROM doc
```

---

first()

Returns the first item of a collection (list, set, array, or the values of a map), or `null` when the collection is empty. For scalar types this method is an identity. This is the method form of the `first()` SQL function.

Syntax: `<value>.first()`

Applies to the following types:

- collection

Examples

```
SELECT 'a,b,c'.split(',').first()
SELECT tags.first() FROM Post
```

---

format() [Method]

Returns the value formatted using the common "printf" syntax. For the complete reference goto Java Formatter JavaDoc.

Syntax: `<value>.format(<format> [, <time-zone>])`

Applies to the following types:

- any

On a date, a datetime or a list of them the format is a date pattern, and the optional time zone (a zone name such as `'Europe/Rome'` or an offset such as `'+02:00'`, as text) is the zone the value is rendered in. A time zone that is not text, or that names no zone, is refused as the caller's error.

Number formatting (`%f`, `%e`, `%d`, etc.) is always locale-independent (e.g. `.` as decimal separator), regardless of the server's configured locale.

Examples

Formats salaries as number with 11 digits filling with 0 at left:

```
SELECT salary.format("%-011d") FROM Employee
```

---

hash()

Returns the hash of the field. Supports all the algorithms available in the JVM.

Syntax: <value>.hash([<algorithm>])`

Applies to the following types:

- string

Examples

Get the SHA-512 of the field "password" in the type User:

```
SELECT password.hash('SHA-512') FROM User
```

---

ifempty()

Return argument if empty results from value/field/expression, otherwise return result.

Syntax: <value/field/expression>.ifempty(<value>)

Applies to the following types:

- string
- list

Examples

```
SELECT "".ifempty("Empty")
```

---

ifnull() [Method]

Return argument if null results from value/field/expression, otherwise return result.

Syntax: <value/field/expression>.ifnull(<value>)

Applies to the following types:

- any

Examples

```
SELECT name.ifnull("John Doe") FROM names
```

---

include()

Answers a map with only the included properties of the value.

Syntax: `<value>.include(<value>[,]*)`

Applies to the following types:

- record (document, vertex or edge)
- map
- result row
- collection of the above (each element is projected, and the answer is a list of maps)

As with `exclude()`, the answer is always a map with no `@rid`, `@type` or `@cat` unless one of the names is a metadata field (`include('@rid', 'name')`). A name the value does not have answers a `null` entry. A chained access reads the map, so `@this.include('n', 'm').m` works, but a graph traversal such as `.out()` must come before the projection. *(Since v26.10.2)*

Examples

```
SELECT expand( @this.include('name') ) FROM User
```

You can specify a wildcard as ending character to include all the fields that start with a certain string. Example to include all the fields that starts with `amount`:

```
SELECT expand( @this.include('amount*') ) FROM V
```

A `null` field name is skipped wherever it appears, as by `exclude()`: `include(null, 'name')` is `include('name')`, and `include(null)` answers an empty map. Before v26.10.2 a `null` as the first argument made `include()` answer `null`.

---

indexOf()

Returns the position of the 'string-to-search' inside the value. It returns -1 if no occurrences are found. 'begin-position' is the optional position where to start, otherwise the beginning of the string

is taken (=0).

Syntax: `<value>.indexOf(<string-to-search> [,<begin-position>])`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- string

Examples

Returns all the UK numbers:

```
SELECT FROM Contact WHERE phone.indexOf('+44') > -1
```

---

intersectsWith()

Returns Boolean answering if argument shape intersects with shape instance.

Syntax:
`<point|circle|rectangle|linestring|polygon>.intersectsWith(<point|circle|rectangle|linestring|polygon>)`

Examples

```
SELECT linestring( [ [10,10], [20,10], [20,20], [10,20], [10,10] ] ).intersectsWith(
rectangle(10,10,20,20) ) AS collision
```

---

isWithin()

Returns Boolean answering if argument shape is fully inside shape instance. It answers like `geo.within()`: a shape is within itself (before 26.12.1 it answered `false` for two equal shapes).

Syntax:
`<point|circle|rectangle|linestring|polygon>.isWithin(<point|circle|rectangle|linestring|polygon>)`

Examples

```
SELECT point(11,11).isWithin( circle(10,10,10) ) AS inside
```

---

javaType()

Returns the corresponding Java Type.

Syntax: `<value>.javaType()`

Applies to the following types:

- any

Examples

Prints the Java type used to store dates:

```
SELECT created.javaType() FROM Events
```

---

join()

Returns all elements of a collection converted to a string and concatenated, with strings separated by the argument string that has the default value `','`. A scalar receiver is returned as its string form.

Syntax: `<value>.join([<separator>])`

Applies to the following types:

- lists
- sets
- arrays (for example a list parameter)

Examples

```
SELECT [1,2,3].join()
SELECT [1,2,3].join('|')
SELECT 'a,b,c'.split(',').join('-')
```

---

keys()

Returns the map's keys as a separate set. Useful to use in conjunction with IN, CONTAINS and CONTAINSALL operators.

Syntax: `<value>.keys()`

Applies to the following types:

- maps
- documents

Examples

```
SELECT FROM Actor WHERE 'Luke' IN attributes.keys()
```

---

last()

Returns the last item of a collection (list, set, array, or the values of a map), or `null` when the collection is empty. For scalar types this method is an identity. This is the method form of the `last()` SQL function.

Syntax: `<value>.last()`

Applies to the following types:

- collection

Examples

```
SELECT 'a,b,c'.split(',').last()
SELECT tags.last() FROM Post
```

---

lastIndexOf()

Returns the position of the 'string-to-search' inside the value starting from from the end. It returns -1 if no occurrences are found. 'begin-position' is the optional position where to start, otherwise the end of the string is taken (=0).

Syntax: `<value>.lastIndexOf(<string-to-search> [,<begin-position>])`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- string

```
SELECT 'Hello Albert'.lastIndexOf('l')
```

---

left()

Returns a substring of the original cutting from the begin and getting 'len' characters.

Syntax: `<value>.left(<length>)`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- string

Examples

```
SELECT FROM Actors WHERE name.left( 4 ) = 'Luke'
```

---

length()

Returns the string length of the input. If `null`, `null` is returned, as SQL's `LENGTH()` does. An empty string returns 0.

Syntax: `<value>.length()`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- any

This method first converts its input to a string and then returns the string's length. For obtaining a collection's lengths use the size() method.

Examples

```
SELECT FROM Providers WHERE name.length() > 0
```

This behavior changed in 26.11.1 (issue #10193). Before, `length()` on a `null` value returned 0. A condition such as `WHERE name.length() = 0` no longer matches records where `name` is `null`: write `WHERE name IS NULL OR name.length() = 0` to keep

matching them.

---

normalize()

Unicode normalization form can be NDF, NFD, NFKC, NFKD, the default is NDF. Pattern-matching, if not defined is the regular expression `"\p{InCombiningDiacriticalMarks}+"`. For more information look at Unicode Standard.

Syntax: `<value>.normalize( [<form>] [,<pattern-matching>] )`

Applies to the following types:

- string

Examples

```
SELECT FROM V WHERE name.normalize() AND name.normalize('NFD')
```

---

precision()

Adapts the returned date or time format to the argument target precision.

Syntax: `<value>.precision('<string>')`

The admissible argument values are given here.

Applies to the following types:

- datetime
- date

Examples

```
SELECT sysdate().precision('millisecond')
```

---

prefix()

Prefixes a string to another one. A `null` base value results in a `null`.

Syntax: `<value>.prefix('<string>')`

Applies to the following types:

- string

Examples

```
SELECT name.prefix('Mr. ') FROM Profile
```

---

remove()

Removes the first occurrence of the passed items.

Syntax: `<value>.remove(<item>*)`

Applies to the following types:

- collection

Examples

```
SELECT [1,2,3,1].remove(1)
```

```
SELECT out().in().remove( @this ) FROM V
```

A `null` argument is honoured wherever it appears in the list: it removes a `null` element from the collection, and never discards the other arguments.

An item is found when it is equal to the argument as a number, whatever the type of either: `remove(2)` removes the `2` of a list written by openCypher, whose integers are `LONG`, as it does in a list written by SQL. A `DECIMAL` is compared exactly. Nothing else is converted: `['2'].remove(2)` removes nothing. `UPDATE … REMOVE list = value` and `list - value` find the item the same way. *(Since v26.12.1; before, the item was compared by its Java type, so a list written by openCypher kept the `2`, and `UPDATE` still reported the record as updated)*

A map argument is one item, found by an element equal to it entry by entry: `[{'a':1},{'b':2}].remove({'a':1})` answers `[{'b':2}]`. *(Since v26.12.1; before, the keys of the map were removed instead)*

---

removeAll()

Removes all the occurrences of the passed items.

Syntax: `<value>.removeAll(<item>*)`

Applies to the following types:

- collection

Examples

```
SELECT [1,2,3,1].removeAll(1)
```

```
SELECT out().in().removeAll( @this ) FROM V
```

A `null` argument is honoured wherever it appears in the list: it removes every `null` element from the collection, and never discards the other arguments.

Items are found as with remove(): `[1, 2, 1L, 1.0].removeAll(1)` answers `[2]`. *(Since v26.12.1)*

---

replace()

Replace a string with another one. If the value is `null`, or either argument is `null`, `null` is returned, as SQL's `REPLACE()` does.

Syntax: `<value>.replace(<to-find>, <to-replace>)`

Applies to the following types:

- string

Examples

```
SELECT name.replace('Mr.', 'Ms.') FROM User
```

This behavior changed in 26.11.1 (issue #10193). Before, `replace()` on a `null` value returned its first argument (`<to-find>`), so `name.replace('Mr.', 'Ms.')` answered `'Mr.'` for a record without a `name`.

(Since v26.11.1, issue #10485) A character outside the Basic Multilingual Plane, such as an emoji, is never cut. With an empty `<to-find>` the replacement goes before every character and after the last, so `'a🙂'.replace('', '-')` is `'-a-🙂-'`, where it used to put a replacement between the two halves of the emoji. An occurrence is never found inside a character either. `replace()` of the SQL method, the `replace()` function of openCypher and `text.replace()` share one implementation.

---

right()

Returns a substring of the original cutting from the end of the string 'length' characters.

Syntax: `<value>.right(<length>)`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two.

> Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- string

Examples

Returns all the vertices where the name ends by "ke".

```
SELECT FROM V WHERE name.right( 2 ) = 'ke'
```

---

size()

Returns the size of the collection, or the number of characters of a string.

Syntax: `<value>.size()`

Applies to the following types:

- collection
- string

> The size of a string counts Unicode characters, as length() does: a character outside the Basic Multilingual Plane, such as an emoji, is one character. Before 26.11.1 (issue #10378) it counted UTF-16 code units, so such a character counted 2. The size of `NULL` is 0, since the method cannot tell a missing collection from a missing string; use length() to get `NULL` for a string that is `NULL`.

Examples

Returns all the items in a tree with children:

```
SELECT FROM TreeItem WHERE children.size() > 0
```

---

sort()

Returns a sorted copy of the collection as a list. The direction is controlled by a boolean argument, which sets ascending direction for a `true` value (default), and descending for `false`.

> Only list of a single type can be sorted.

> For scalar types this method is an identity.

Syntax: `<value>.sort([<dir>])`

Applies to the following types:

- lists
- sets
- arrays (for example a list parameter)

Examples

Sort inline lists:

```
SELECT [4,1,9,23].sort()
SELECT ["a","Y","b","9"].sort(false)
SELECT 'c,a,b'.split(',').sort()
```

---

split()

Returns a list from a string separated by the provided delimiter. The delimiter is matched literally, not as a regular expression. The result is a list, so every collection method can be chained on it: `'a,b'.split(',').join('-')`, `.sort()`, `.first()`, `.last()`, `.size()`.

Syntax: `<value>.split(<string>)`

Applies to the following types:

- string

Examples

Returns string of comma separated values as list

```
SELECT 'a,b,c,d,e'.split(',')
```

(Since v26.11.1, issue #10485) The empty pieces are kept, as openCypher `split()` keeps them: `'a,'.split(',')` is `['a', '']` and `','.split(',')` is `['', '']`. Before, the empty pieces at the end of the list were dropped (`['a']` and `[]`). An empty delimiter splits a string into its characters, and a character outside the Basic Multilingual Plane, such as an emoji, is one character: `'a'.split('')` is `['a', '']`, where it used to be `['a', '\uD83D', '\uDE00']`, the two halves of the emoji. The `split()` method and function and `text.split()` of SQL and the `split()` function of openCypher share one implementation, so they answer alike.

---

subString()

Returns a substring of the original cutting from 'begin' index up to 'end' index (not included).

Syntax: `<value>.subString(<begin> [,<end>] )`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10248) the count was in UTF-16 code units, so such a character counted 2.

Applies to the following types:

- string

Examples

Get all the items where the name begins with an "L":

```
SELECT FROM StockItems WHERE name.substring( 0, 1 ) = 'L'
```

Substring of `ArcadeDB`

```
SELECT "ArcadeDB".substring(0,6)
```

returns `Arcade`

---

transform()

Returns the underlying collection to which one or more methods (passed as string arguments) are applied element-wise.

Syntax: `<value>.transform(<string>[,<string>]*)`

Applies to the following types:

- collection

The argument methods cannot take arguments themselves.

Examples

```
SELECT FROM Car WHERE options.transform( 'trim', 'toLowercase' ) CONTAINSALL ['a/c',
'airbags']
```

---

trim()

Returns the original string removing white spaces from the begin and the end.

Syntax: `<value>.trim()`

Applies to the following types:

- string

Examples

```
SELECT name.trim() FROM Actors
```

---

trimPrefix()

Returns the original string removing argument from the front, if exists.

Syntax: `<value>.trimPrefix(<value>)`

Applies to the following types:

- string

Examples

```
SELECT name.trimPrefix("Mr. ") FROM Actors
```

---

trimSuffix()

Returns the original string removing argument from the end, if exists.

Syntax: `<value>.trimSuffix(<value>)`

Applies to the following types:

- string

Examples

```
SELECT name.trimSuffix(";") FROM Actors
```

---

toLowerCase()

Returns the string in lower case.

Syntax: `<value>.toLowerCase()`

Applies to the following types:

- string

Examples

```
SELECT name.toLowerCase() FROM Actors
```

---

toUpperCase()

Returns the string in upper case.

Syntax: `<value>.toUpperCase()`

Applies to the following types:

- string

Examples

```
SELECT name.toUpperCase() FROM Actors
```

---

type()

Returns the value's ArcadeDB type as string. The potential return values are listed as "**SQL type**" here.

The returned strings are all capitals.

Syntax: `<value>.type()`

Applies to the following types:

- any

Examples

Prints the type used to store dates:

```
SELECT created.type() FROM Events
```

values()

Returns the map's values as a separate collection. Useful to use in conjunction with IN, CONTAINS and CONTAINSALL operators.

Syntax: `<value>.values()`

Applies to the following types:

- maps
- documents
- query result rows (a sub-query row, a `LET` variable)

`keys()` and `values()` are an index-aligned pair: the n-th value belongs to the n-th key. For a document or a result row the values are those of the properties `keys()` lists, without the record metadata (`@cat`, `@type`, `@rid`, and `@in` and `@out` of an edge). A row that overrides a property of its record answers the row's value in that position (*since v26.10.2*; before, a result row also returned the metadata and the record's value).

Examples

```
SELECT FROM Clients WHERE attributes.values() CONTAINSALL (@this IS NOT NULL)
```

A property named like a function namespace (`map`, `geo`, `math`, ...) must be back-ticked to be read as a property: `` `map`.values() ``. Unquoted, `map.values()` calls the `map.*` function.

14.25. Triggers

ArcadeDB supports database triggers that automatically execute SQL statements, JavaScript code, or Java classes in response to record events. Triggers enable you to implement business logic, data validation, audit trails, and automated workflows directly in your database.

14.25.1. Overview

A **trigger** is a named database object that automatically executes when specific events occur on records of a particular type. Triggers can execute SQL statements, JavaScript code, or Java classes, giving you flexibility in how you implement your logic.

Key Features

- **Event-driven**: Triggers fire automatically on CREATE, READ, UPDATE, or DELETE operations
- **Timing control**: Execute BEFORE or AFTER the event
- **Multi-language**: Choose between SQL statements, JavaScript code, or Java classes
- **High performance**: Java class triggers offer maximum performance with compiled code
- **Persistent**: Triggers are stored in the schema and survive database restarts
- **Type-specific**: Each trigger applies to a document/vertex/edge type and to all its subtypes (see Subtypes)

14.25.2. Trigger Events

Triggers can respond to eight different combinations of timing and events:

Event	Description
`BEFORE CREATE`	Before a new record is created
`AFTER CREATE`	After a new record is created
`BEFORE READ`	Before a record is read from database
`AFTER READ`	After a record is read from database
`BEFORE UPDATE`	Before a record is modified
`AFTER UPDATE`	After a record is modified
`BEFORE DELETE`	Before a record is deleted
`AFTER DELETE`	After a record is deleted

Re-pointing an edge (`UPDATE … SET @in = …` / `@out = …`, or `set("@in", …)` from Java) replaces its record with a new one. `AFTER CREATE` fires for the replacement and `AFTER DELETE` for the edge it replaces. `BEFORE DELETE` does not fire, because a veto there would leave both edges in place. *(Since v26.10.2: only the replacement's `AFTER CREATE` fired, so a listener keeping derived state, such as a graph analytical view, kept the old edge too.)*

Creating or deleting an edge fires the edge's triggers only. The vertices it connects keep a pointer to their edge list, which the database rewrites when a vertex gets its first edge of a direction or its edge list grows a new segment; that is internal bookkeeping, not an update of the vertex, so it fires no `UPDATE` trigger on the vertex and a `BEFORE UPDATE` trigger cannot veto it. *(Since v26.11.1: the rewrite fired the vertex's `UPDATE` triggers on some edge creations and not on others, and a `BEFORE UPDATE` trigger that vetoed it left the new edge unreachable from the vertex.)*

`INSERT INTO <type> FROM SELECT …` fires `CREATE` triggers once per inserted row, like any other `INSERT`. *(Since v26.11.1: it also fired the `UPDATE` triggers of every inserted row.)*

14.25.3. Subtypes

A trigger declared `ON TYPE P` fires for the records of `P` and of every type that extends it, directly or through other types: the same records `SELECT FROM P` returns. A `BEFORE` trigger that validates the writes to `P` therefore cannot be bypassed by writing to a subtype.

```
CREATE VERTEX TYPE Person;
CREATE VERTEX TYPE Employee EXTENDS Person;
CREATE TRIGGER person_audit AFTER CREATE ON TYPE Person
  EXECUTE SQL 'INSERT INTO AuditLog SET action = "created", ts = sysdate()';

INSERT INTO Employee SET name = 'Jay';   -- fires person_audit
```

- A trigger on a subtype does not fire for the records of its super type.
- A record fires the triggers of its own type first, then those of its super types, nearest first. A veto by any `BEFORE` trigger stops the operation.
- Each trigger fires once per record, even when its type is reached through more than one path. This is the case of an openCypher node with several labels: `CREATE (:A:B)` creates a record of the composite type `A~B`, which extends both `A` and `B`, so it fires the triggers of `A` and of `B`, and those of a type both labels extend only once.
- The coverage follows the type hierarchy as it changes: a type created later with `EXTENDS P`, or given `P` as a super type with `ALTER TYPE … SUPERTYPE +P`, is covered from then on, and one that stops extending `P` is no longer.
- Listeners registered from Java on a type (`database.getSchema().getType("P").getEvents()`) are not polymorphic: they keep firing for the records of that exact type only.

(Since v26.11.1: a trigger fired only for the records of its exact type, so the records of its subtypes, openCypher multi-label nodes included, skipped it.)

`DROP TYPE` drops the triggers declared on the type, as it drops its indexes. The subtypes of a dropped type are linked to its super types, so they keep firing the triggers of those. *(Since v26.11.1: the trigger stayed in the schema, naming a type that no longer existed, and a type created later under the same name took it over at the next open.)*

14.25.4. SQL Triggers

SQL triggers execute SQL statements. They have access to the current record through context variables.

Context Variables

- `$record` - The current record being operated on. Supports property and attribute navigation, e.g. `$record.name` or `$record.@rid`
- `$oldRecord` - The original record (for UPDATE operations), not set otherwise

- `$rid` - For `BEFORE READ` only: the identity of the record about to be read. `BEFORE READ` fires before the record exists, so `$record` is not set for this timing - see BEFORE READ triggers below

Return Value

A SQL trigger body's result decides whether the operation continues, for `BEFORE` triggers only. A body that evaluates to a single scalar `false` - one row with one property, holding `false` - aborts the operation, silently: no exception is thrown, the record is simply not created/updated/deleted (or, for `BEFORE READ`, not read). `SELECT false` and `SELECT (<condition>) AS ok` both have this shape when `<condition>` evaluates to `false`.

Any other result - multiple rows, multiple columns, no rows, or a `DML` statement (`INSERT`/`UPDATE`/`DELETE`) - is not treated as a veto and the operation proceeds, even though the body still runs. This keeps the contract unambiguous: only a body that looks like a single yes/no answer can say no.

BEFORE READ triggers

`BEFORE READ` fires from inside the read itself, before the record has been materialized, so the body cannot see `$record` - only `$rid`, the identity of the record about to be read:

```
CREATE TRIGGER audit_read BEFORE READ ON TYPE Document
EXECUTE SQL 'SELECT ($rid IS NOT NULL) AS ok';
```

A trigger that needs the record's content wants `AFTER READ` instead, which does receive `$record`.

The body of `EXECUTE SQL` and `EXECUTE JAVASCRIPT` is a quoted string, and a literal line break inside a quoted string is not allowed. Write the body on one line, or use `\n` for a line break (as the examples below do for multi-line JavaScript bodies).

Example: Audit Trail

Create an audit log that tracks all user creations:

```
-- Create the audit log type
CREATE DOCUMENT TYPE AuditLog;

-- Create trigger to log user creations
CREATE TRIGGER user_audit AFTER CREATE ON TYPE User
EXECUTE SQL 'INSERT INTO AuditLog SET action = "user_created", userName =
$record.name, timestamp = sysdate()';
```

Now every time a User is created, an entry is automatically added to the AuditLog:

```
-- Create a user
INSERT INTO User SET name = 'Alice', email = 'alice@example.com';

-- Check the audit log
```

```
SELECT * FROM AuditLog;
-- Returns: {action: "user_created", userName: "Alice", timestamp: ...}
```

Example: Auto-increment Counter

Automatically set a sequence number on new documents:

```
-- Create a counter type
CREATE DOCUMENT TYPE Counter;
INSERT INTO Counter SET name = 'order_sequence', value = 1000;

-- Create trigger to auto-increment order numbers
CREATE TRIGGER order_number BEFORE CREATE ON TYPE Order
EXECUTE SQL 'UPDATE Counter SET value = value + 1 WHERE name = "order_sequence";
UPDATE $record SET orderNumber = (SELECT value FROM Counter WHERE name =
"order_sequence")';
```

Example: Cascade Updates

Update related records when the parent changes:

```
-- Update all orders when customer email changes
CREATE TRIGGER customer_email_update AFTER UPDATE ON TYPE Customer
EXECUTE SQL 'UPDATE Order SET customerEmail = $record.email WHERE customerId =
$record.@rid';
```

Example: Data Validation

Enforce business rules using BEFORE triggers. The body evaluates a condition against `$record` and projects it as a single named column; when it evaluates to `false`, the operation is silently aborted (see Return Value above):

```
-- Ensure product prices are positive
CREATE TRIGGER validate_price BEFORE CREATE ON TYPE Product
EXECUTE SQL 'SELECT ($record.price > 0) AS ok';
```

```
-- Rejected: nothing is inserted, no exception is thrown
INSERT INTO Product SET name = 'Bad', price = -5;

-- Accepted
INSERT INTO Product SET name = 'Good', price = 5;
```

14.25.5. JavaScript Triggers

JavaScript triggers offer more flexibility and can implement complex logic with conditional

statements, loops, and calculations.

Since v26.10.2 terminating the statement that fired a JavaScript trigger (HTTP `terminate query`, Cypher `TERMINATE TRANSACTIONS` or Studio) interrupts the trigger's script too, through GraalVM, so a trigger caught in an endless loop does not hold the statement forever.

Context Variables

- `record` or `$record` - The current record being operated on
- `oldRecord` or `$oldRecord` - The original record (for UPDATE operations), null otherwise
- `database` - The database instance

Return Value

JavaScript triggers can return `false` to abort the operation (for BEFORE triggers only).

Example: Data Validation

Validate email format before creating users:

```
CREATE TRIGGER validate_email BEFORE CREATE ON TYPE User
EXECUTE JAVASCRIPT 'if (!record.email || !record.email.includes("@")) {\nthrow new
Error("Invalid email address");\n}';
```

Example: Auto-populate Fields

Automatically set timestamps and computed fields:

```
CREATE TRIGGER user_defaults BEFORE CREATE ON TYPE User
EXECUTE JAVASCRIPT '// Set creation timestamp\nrecord.createdAt = new Date();\n//
Generate username from email if not provided\nif (!record.username && record.email)
{\nrecord.username = record.email.split("@")[0];\n}\n// Set default role\nif
(!record.role) {\nrecord.role = "user";\n}';
```

Example: Complex Business Logic

Implement discount rules based on order total:

```
CREATE TRIGGER calculate_discount BEFORE CREATE ON TYPE Order
EXECUTE JAVASCRIPT 'var total = record.total || 0;\nvar discount = 0;\n// Apply
discount based on order total\nif (total > 1000) {\ndiscount = 0.15;  // 15%
discount\n} else if (total > 500) {\ndiscount = 0.10;  // 10% discount\n} else if
(total > 100) {\ndiscount = 0.05;  // 5% discount\n}\nrecord.discountPercent =
discount * 100;\nrecord.discountAmount = total * discount;\nrecord.finalTotal = total
- (total * discount);';
```

Example: Conditional Abort

Prevent operations based on business rules:

```
CREATE TRIGGER prevent_weekend_orders BEFORE CREATE ON TYPE Order
EXECUTE JAVASCRIPT 'var day = new Date().getDay();\nif (day === 0 || day === 6) {
\nthrow new Error("Orders cannot be placed on weekends");\n}';
```

Example: Audit with Details

Create detailed audit logs with JavaScript:

```
CREATE TRIGGER audit_update AFTER UPDATE ON TYPE Product
EXECUTE JAVASCRIPT 'database.command("sql",\n"INSERT INTO AuditLog SET action = ?,
productId = ?, productName = ?, timestamp = sysdate()",\n"product_updated",
\nrecord["@rid"],\nrecord.name\n);';
```

14.25.6. Java Triggers

Java triggers offer maximum performance by executing compiled Java code. They require implementing the `JavaTrigger` interface and must be available in the classpath.

Creating a Java Trigger Class

First, create a Java class that implements the `com.arcadedb.schema.trigger.JavaTrigger` interface:

```
package com.example.triggers;

import com.arcadedb.database.Database;
import com.arcadedb.database.Document;
import com.arcadedb.database.Record;
import com.arcadedb.schema.trigger.JavaTrigger;

public class EmailValidationTrigger implements JavaTrigger {

  @Override
  public boolean execute(Database database, Record record, Record oldRecord) throws
Exception {
    if (record instanceof Document) {
      Document doc = (Document) record;
      String email = doc.getString("email");

      if (email == null || !email.contains("@")) {
        throw new IllegalArgumentException("Invalid email address");
      }
    }
    return true;  // Continue with the operation
  }
```

```
}
```

JavaTrigger Interface

```
public interface JavaTrigger {
  /**
   * Execute the trigger logic.
   *
   * @param database  The database instance
   * @param record    The current record being operated on
   * @param oldRecord The original record (for UPDATE operations), null otherwise
   * @return true to continue the operation, false to abort (BEFORE triggers only)
   * @throws Exception to abort the operation with an error message
   */
  boolean execute(Database database, Record record, Record oldRecord) throws
Exception;
}
```

Registering Java Triggers

Once your class is compiled and in the classpath, register it using SQL:

```
CREATE TRIGGER validate_email BEFORE CREATE ON TYPE User
EXECUTE JAVA 'com.example.triggers.EmailValidationTrigger';
```

Example: Data Validation

Validate complex business rules with full Java capabilities:

```
package com.example.triggers;

import com.arcadedb.database.Database;
import com.arcadedb.database.Document;
import com.arcadedb.database.Record;
import com.arcadedb.schema.trigger.JavaTrigger;
import java.math.BigDecimal;

public class PriceValidationTrigger implements JavaTrigger {

  private static final BigDecimal MIN_PRICE = new BigDecimal("0.01");
  private static final BigDecimal MAX_PRICE = new BigDecimal("999999.99");

  @Override
  public boolean execute(Database database, Record record, Record oldRecord) throws
Exception {
    if (record instanceof Document) {
      Document doc = (Document) record;
```

```
      BigDecimal price = doc.get("price");

      if (price == null) {
        throw new IllegalArgumentException("Price is required");
      }

      if (price.compareTo(MIN_PRICE) < 0) {
        throw new IllegalArgumentException("Price must be at least " + MIN_PRICE);
      }

      if (price.compareTo(MAX_PRICE) > 0) {
        throw new IllegalArgumentException("Price cannot exceed " + MAX_PRICE);
      }
    }
    return true;
  }
}
```

Example: Database Queries

Execute queries within the trigger:

```
package com.example.triggers;

import com.arcadedb.database.Database;
import com.arcadedb.database.Document;
import com.arcadedb.database.Record;
import com.arcadedb.query.sql.executor.ResultSet;
import com.arcadedb.schema.trigger.JavaTrigger;

public class StockCheckTrigger implements JavaTrigger {

  @Override
  public boolean execute(Database database, Record record, Record oldRecord) throws
Exception {
    if (record instanceof Document) {
      Document doc = (Document) record;
      String productId = doc.getString("productId");
      Integer quantity = doc.getInteger("quantity");

      ResultSet result = database.query("sql",
          "SELECT stock FROM Product WHERE @rid = ?", productId);

      if (result.hasNext()) {
        Document product = result.next().toElement().asDocument();
        Integer stock = product.getInteger("stock");

        if (stock < quantity) {
          throw new IllegalStateException(
              "Insufficient stock. Available: " + stock + ", Requested: " + quantity);
```

```
        }
      }
    }
    return true;
  }
}
```

Java Trigger Advantages

1. **Performance**: Compiled code executes faster than interpreted JavaScript
2. **Type Safety**: Compile-time type checking prevents runtime errors
3. **Full Java Ecosystem**: Access to all Java libraries and frameworks
4. **IDE Support**: Code completion, refactoring, and debugging
5. **Testability**: Unit test your triggers like any Java class
6. **Reusability**: Share trigger code across projects

Java Trigger Considerations

1. **Classpath**: Trigger classes must be in the ArcadeDB classpath at runtime
2. **Deployment**: Requires redeployment when trigger logic changes
3. **Error Handling**: Exceptions abort the operation and rollback the transaction
4. **Thread Safety**: Trigger instances may be reused across threads; ensure thread-safety
5. **No State**: Avoid instance variables; triggers should be stateless

14.25.7. Use Cases

1. Audit Trails

Track who changed what and when:

```
CREATE TRIGGER audit_all AFTER UPDATE ON TYPE ImportantData
EXECUTE SQL 'INSERT INTO AuditLog SET tableName = "ImportantData", recordId =
$record.@rid, modifiedAt = sysdate()';
```

2. Data Integrity

Ensure referential integrity and business rules:

```
CREATE TRIGGER check_inventory BEFORE CREATE ON TYPE OrderItem
EXECUTE JAVASCRIPT 'var result = database.query("sql",\n"SELECT stock FROM Product
WHERE @rid = ?",\nrecord.productId\n);\nif (result.hasNext()) {\nvar product =
result.next();\nif (product.stock < record.quantity) {\nthrow new Error("Insufficient
stock");\n}\n}';
```

3. Denormalization

Maintain computed or cached values:

```
CREATE TRIGGER update_order_total AFTER CREATE ON TYPE OrderItem
EXECUTE SQL 'UPDATE Order SET totalAmount = (SELECT sum(price * quantity) FROM
OrderItem WHERE orderId = $record.orderId) WHERE @rid = $record.orderId';
```

4. Notifications

Send alerts or trigger external processes:

```
CREATE TRIGGER low_stock_alert AFTER UPDATE ON TYPE Product
EXECUTE JAVASCRIPT 'if (record.stock < 10) {\ndatabase.command("sql",\n"INSERT INTO
Notification SET type = ?, productId = ?, message = ?",\n"low_stock",\n
record["@rid"],\n"Product stock low: " + record.stock\n);\n}';
```

5. Workflow Automation

Automatically progress through workflow states:

```
CREATE TRIGGER order_workflow AFTER UPDATE ON TYPE Order
EXECUTE JAVASCRIPT '// Auto-approve small orders\nif (record.status === "pending" &&
record.total < 100) {\nrecord.status = "approved";\nrecord.approvedAt = new Date();
\n}';
```

14.25.8. Performance Considerations

Benchmark Results

Performance tests measuring trigger execution overhead on document creation with identical operations (100,000 iterations, Java 21, macOS). All triggers perform the same operation: `INSERT INTO AuditLog SET triggered = true`.

Trigger Type	Avg Time (µs)	Overhead (µs)	Overhead (%)
No Trigger (Baseline)	95	—	—
Java Trigger	**147**	**+52**	**+54.7%**
SQL Trigger	150	+55	+57.9%
JavaScript Trigger	187	+92	+96.8%

Key Findings:

1. **Java and SQL triggers have nearly identical performance**: Only 2% difference (147 vs 150 µs), both execute compiled code paths efficiently.
2. **JavaScript triggers are ~27% slower**: GraalVM JavaScript execution adds noticeable overhead

compared to native execution.

3. **All triggers add overhead**: Expect ~50-95% overhead depending on trigger type, which is acceptable for most use cases.
4. **Trigger overhead is predictable**: The performance impact is consistent and can be factored into capacity planning.

Performance Recommendations

- **Minimize Work**: Keep trigger code as lightweight as possible
- **Choose the Right Type**:
  - Use **Java triggers** when you need type safety, IDE support, and debugging capabilities
  - Use **SQL triggers** for database operations - performance is nearly identical to Java
  - Use **JavaScript triggers** for dynamic logic where ~30% slower performance is acceptable
- **Avoid Complex Queries**: Heavy queries in triggers can slow down operations
- **Consider Batch Operations**: Triggers fire for each record, which can be expensive in bulk operations
- **Monitor Impact**: Test performance with realistic data volumes
- **Profile Your Workload**: Measure actual impact in your specific use case

Internal Optimizations

ArcadeDB optimizes JavaScript trigger performance through engine pooling:

- **Shared GraalVM Engine**: All JavaScript triggers share a single GraalVM Polyglot Engine instance across the entire database process, reducing memory overhead and initialization time
- **Lightweight Contexts**: Each trigger creates a lightweight execution context that reuses the shared engine
- **Lazy Initialization**: Engine and context creation is deferred until the trigger first executes
- **Automatic Resource Management**: Contexts are properly closed when triggers are removed, while the shared engine persists for the lifetime of the database process

This architecture ensures that creating multiple JavaScript triggers does not linearly increase memory consumption or initialization overhead.

When to Use Each Type

Java Triggers - Best for:

- Complex validation requiring type safety and compile-time checks
- Integration with existing Java libraries
- Code that benefits from IDE support and refactoring
- Unit testing requirements
- Team prefers strongly-typed languages

SQL Triggers - Best for:

- Simple database operations (audit logs, denormalization)
- Prototyping and development (no compilation step)
- Deployment simplicity (embedded in schema)
- Operations that are primarily SQL-based
- **Performance-critical paths** (nearly identical performance to Java)

JavaScript Triggers - Best for:

- Moderate complexity business logic
- Rapid development and iteration
- Dynamic validation rules that change frequently
- Scenarios where scripting flexibility outweighs performance
- Teams comfortable with JavaScript

14.25.9. Best Practices

1. Keep Triggers Simple

Triggers execute synchronously and can impact performance. Keep logic simple and fast:

```
-- Good: Simple, fast operation
CREATE TRIGGER set_timestamp BEFORE CREATE ON TYPE Document
EXECUTE JAVASCRIPT 'record.createdAt = new Date();';

-- Avoid: Complex calculations that could be done elsewhere
```

2. Use Meaningful Names

Name triggers clearly to indicate their purpose:

```
-- Good naming
CREATE TRIGGER audit_user_creation AFTER CREATE ON TYPE User ...
CREATE TRIGGER validate_email BEFORE CREATE ON TYPE User ...

-- Poor naming
CREATE TRIGGER trigger1 AFTER CREATE ON TYPE User ...
```

3. Handle Errors Gracefully

In JavaScript triggers, throw descriptive errors:

```
CREATE TRIGGER validate_age BEFORE CREATE ON TYPE User
```

```
EXECUTE JAVASCRIPT 'if (!record.age || record.age < 18) {\nthrow new Error("User must
be at least 18 years old");\n}';
```

4. Avoid Infinite Loops

Be careful not to create circular trigger dependencies:

```
-- DANGER: This could create an infinite loop if not careful
-- Trigger A updates Type B, which triggers B updates Type A, etc.
```

5. Test Thoroughly

Test triggers with various scenarios:

- Normal operations
- Edge cases (null values, empty strings, etc.)
- Error conditions
- Performance with large datasets

14.25.10. Managing Triggers

List All Triggers

Use the Java API to list all triggers:

```
Trigger[] triggers = database.getSchema().getTriggers();
for (Trigger trigger : triggers) {
  System.out.println(trigger.getName() + " on " + trigger.getTypeName());
}
```

Check if Trigger Exists

```
boolean exists = database.getSchema().existsTrigger("trigger_name");
```

Get Triggers for a Type

```
Trigger[] triggers = database.getSchema().getTriggersForType("User");
```

Remove a Trigger

```
DROP TRIGGER user_audit;
```

14.25.11. Limitations

1. **Synchronous Execution**: Triggers execute synchronously within the transaction. Long-running triggers can impact performance.
2. **Type Matching**: A trigger covers its type and all its subtypes (see Subtypes). A record that has a trigger on both its type and a super type fires both.
3. **Order of Execution**: When multiple triggers exist for the same event on the same type, they execute in alphabetical order by trigger name.
4. **BEFORE READ Limitation**: BEFORE READ triggers receive only the RID and must load the record, which can cause a double-read.
5. **Transaction Context**: Triggers execute within the same transaction as the triggering operation. If a trigger fails, the entire transaction rolls back.
6. **JavaScript Sandboxing**: JavaScript triggers run in a sandboxed GraalVM environment with a restricted host-class allow-list. The allow-list is limited to the benign value packages `java.util.*`, `java.time.*` and `java.math.*`; `java.lang.*` was removed so a trigger can no longer look up `java.lang.Runtime` and reach `Runtime.getRuntime().exec(…)` for OS command execution (advisory `GHSA-x9f9-r4m8-9xc2`). JavaScript's native `String`, `Math` and `Number` cover most trigger logic, so dropping `java.lang.*` rarely affects real triggers.

14.25.12. See Also

- CREATE TRIGGER
- DROP TRIGGER
- SQL Script
- User Functions

14.26. User Functions

ArcadeDB can be extended with user-defined functions written in SQL, JavaScript, OpenCypher, or Java. Functions defined in any language are callable from ALL query languages (SQL, Cypher, Gremlin, GraphQL, MongoDB).

14.26.1. Overview

User functions provide a powerful way to encapsulate business logic and reuse it across different query languages:

- **SQL Functions**: Use SQL queries as function bodies
- **JavaScript Functions**: Use JavaScript for complex calculations
- **OpenCypher Functions**: Use Cypher queries for graph operations
- **Java Functions**: Use Java for maximum performance

Universal Function Registry

Functions are stored in a global registry and are accessible from any query language:

```
-- Define once in SQL
DEFINE FUNCTION math.sum "SELECT :a + :b" PARAMETERS [a,b] LANGUAGE sql

-- Call from SQL
SELECT `math.sum`(3, 5) AS result

-- Call from Cypher
RETURN math.sum(3, 5) AS result

-- Call from Gremlin, GraphQL, MongoDB queries...
```

14.26.2. Database's Functions

Look at the Database Interface page for built-in database functions.

14.26.3. Defining User Functions

User functions can be defined via SQL using the following command:

```
DEFINE FUNCTION <library>.<name> "<body>" [PARAMETERS [<parameter>,*]] LANGUAGE
<language>
```

- `<library>` - A namespace to group related functions
- `<name>` - The function's name
- `<body>` - The function's body as a string in the chosen language's syntax. Inside the quoted body you can use the standard string escapes: `\"` for a double quote, `\'` for a single quote, `\\` for a backslash and `\n` for a newline (a literal line break inside the quotes is not allowed, use `\n` instead)
- `[<parameter>,*]` - A list of parameter identifiers used in the function body. Omit the `PARAMETERS []` block for functions without parameters
- `<language>` - One of: `sql`, `js` (JavaScript), or `opencypher` (alias `cypher`), case-insensitive. Java is not a valid `LANGUAGE` value: Java functions are registered programmatically (see User Functions in Java below). A library holds functions of one language only: the first `DEFINE FUNCTION` on a library sets its language, and defining a function of another language in it (or in a Java library registered programmatically) is refused with an error naming both languages

Persistence

Functions defined with `DEFINE FUNCTION` (SQL, JavaScript and OpenCypher) are stored with the database schema: they survive a restart and, in a High Availability cluster, are automatically replicated to all the servers. `DELETE FUNCTION` removes them permanently. Java functions are the exception, they are registered

programmatically at startup (see User Functions in Java below).

Required privileges

Defining a function over a remote connection is gated by the caller's group permissions (see Groups):

- `DEFINE FUNCTION ... LANGUAGE sql` and `LANGUAGE opencypher` require the `updateSchema` permission, like any other schema change.
- `DEFINE FUNCTION ... LANGUAGE js` requires the stronger `updateSecurity` permission, because a JavaScript body runs host code.

Embedded databases, schema loading and HA replication are unaffected — the guard only applies to remote commands.

SQL Language Functions

SQL functions execute SQL queries and return the result of the first row's projection.

The return value in a SQL function is determined by the projection named `'result'` or the first column if only one column is returned.

Examples

```
-- Simple arithmetic function
DEFINE FUNCTION math.add "SELECT :a + :b AS result" PARAMETERS [a,b] LANGUAGE sql;
SELECT `math.add`(10, 5) AS sum;  -- Returns 15

-- Function returning a string
DEFINE FUNCTION the.answer 'SELECT "forty-two" AS result' LANGUAGE sql;
SELECT `the.answer`();  -- Returns "forty-two"

-- Function with no parameters
DEFINE FUNCTION utils.currentYear 'SELECT sysdate().format("yyyy") AS result' LANGUAGE
sql;
SELECT `utils.currentYear`();
```

JavaScript Language Functions

JavaScript functions use GraalVM's JavaScript engine for complex calculations and data transformations.

Sandbox

The polyglot engine that runs a `LANGUAGE js` function is locked down: it executes with host file/URL access disabled (`IOAccess.NONE`, so `load(path|url)` cannot read local files or reach network resources) and cross-language access disabled (`PolyglotAccess.NONE`, so it cannot pivot into another guest language). Nor can a

> script reach the security state of the request it runs in: the thread context returned by `database.getContext()` and the user objects (the current user and its permissions) are not accessible from a script, so a function cannot read or replace the user it runs as. Combined with the `updateSecurity` requirement above, this confines JavaScript functions to pure in-process computation over their arguments.

Since v26.10.2 a statement that calls a JavaScript function can be stopped while the function runs, by the HTTP `terminate query` server command, Cypher `TERMINATE TRANSACTIONS` or Studio: the function is interrupted through GraalVM, so even an endless loop in it stops.

Examples

```
-- Fused multiply-add function
DEFINE FUNCTION my.fma 'return a + b * c' PARAMETERS [a,b,c] LANGUAGE js;
SELECT `my.fma`(1, 2, 3);  -- Returns 7

-- String manipulation
DEFINE FUNCTION text.greet 'return "Hello, " + name + "!"' PARAMETERS [name] LANGUAGE
js;
SELECT `text.greet`('World');  -- Returns "Hello, World!"

-- Complex JSON processing
DEFINE FUNCTION json.extract 'return JSON.parse(data).users.length' PARAMETERS [data]
LANGUAGE js;
SELECT `json.extract`('{"users":[1,2,3]}');  -- Returns 3
```

OpenCypher Language Functions

OpenCypher functions execute Cypher queries, perfect for graph traversals and pattern matching.

You can use either `opencypher` or `cypher` as the language identifier (they are aliases).

Examples

```
-- Simple calculation using Cypher
DEFINE FUNCTION cypher.double "RETURN $x * 2" PARAMETERS [x] LANGUAGE opencypher;
SELECT `cypher.double`(5);  -- Returns 10

-- Graph query function
DEFINE FUNCTION graph.countNeighbors
  "MATCH (n) WHERE id(n) = $nodeId MATCH (n)-[]->() RETURN count(*) AS cnt"
  PARAMETERS [nodeId] LANGUAGE cypher;

-- Use in SQL query
SELECT `graph.countNeighbors`('#1:0') AS neighbors;

-- Use in Cypher query
```

```
MATCH (n:Person) RETURN n.name, graph.countNeighbors(id(n)) AS friendCount;

-- Complex graph pattern function
DEFINE FUNCTION graph.findCommonFriends
  "MATCH (a)-[:KNOWS]->(common)<-[:KNOWS]-(b)
   WHERE id(a) = $id1 AND id(b) = $id2
   RETURN count(DISTINCT common) AS cnt"
  PARAMETERS [id1, id2] LANGUAGE opencypher;
```

Calling User Functions

Functions can be called from any supported query language:

Language	Syntax	Example
SQL	`library.function(args)`	`SELECT `math.sum(3, 5)`
Cypher	`library.function(args)`	`RETURN math.sum(3, 5)`
Cypher CALL	`CALL library.function(args) YIELD result`	`CALL math.sum(3, 5) YIELD result RETURN result`
Gremlin	Standard function call	Access via SQL bridge
GraphQL	In query fields	Access via SQL bridge

Cross-Language Function Calls

One of ArcadeDB's unique features is the ability to define a function in one language and call it from any other:

```
-- Define in SQL
DEFINE FUNCTION sql.concat "SELECT :a || :b" PARAMETERS [a,b] LANGUAGE sql;

-- Define in JavaScript
DEFINE FUNCTION js.power "return Math.pow(x, y)" PARAMETERS [x,y] LANGUAGE js;

-- Define in Cypher
DEFINE FUNCTION cypher.shortestPath
  "MATCH path=shortestPath((a)-[*]-(b))
   WHERE id(a)=$from AND id(b)=$to
   RETURN length(path)"
  PARAMETERS [from, to] LANGUAGE opencypher;

-- Call all from Cypher
RETURN sql.concat('Hello', ' World') AS greeting,
       js.power(2, 10) AS power,
       cypher.shortestPath('#1:0', '#1:100') AS pathLen;

-- Call all from SQL
SELECT `sql.concat`('foo', 'bar') AS str,
       `js.power`(3, 3) AS cube,
```

```
        `cypher.shortestPath`('#1:0', '#1:100') AS dist;
```

Deleting User Functions

A user function can be deleted via SQL using the following command:

```
DELETE FUNCTION <library>.<name>
```

Example

```
DELETE FUNCTION extra.tsum;
DELETE FUNCTION math.obsolete;
```

14.26.4. User Functions in Java

Java functions provide maximum performance and access to the full Java ecosystem. Unlike the SQL, JavaScript and OpenCypher functions described above, Java functions **cannot** be created with `DEFINE FUNCTION ⋯ LANGUAGE java`: they must be registered programmatically through the Java API. There is no auto-discovery (no `ServiceLoader`, no annotation or classpath scan) for user functions: registration is always an explicit API call.

Where does the JAR go, and when is the function registered?

1. **Deploy the JAR.** Copy the JAR containing your function classes into the `lib/` directory of the ArcadeDB installation (`$ARCADEDB_HOME/lib/`). The server start script (`server.sh` / `server.bat`) puts the whole `lib/` folder on the classpath (`-cp "$ARCADEDB_HOME/lib/*"`), so your classes become loadable with `Class.forName(⋯)`.
2. **Register the function.** Because there is no auto-discovery, you must call the registration API yourself. **When** and **where** depends on how you run ArcadeDB:
   - **Embedded** (ArcadeDB as a library inside your application): register right after opening the database, before running queries that use the function.
   - **Standalone server**: registration has to run inside the server JVM. Package the registration logic as a Server Plugin and register the function from its `startService()` method (see the server example below). Drop the same JAR in `lib/plugins/` and enable it with the `arcadedb.server.plugins` setting.

There are two independent ways to expose Java code as functions.

Option A: Register a Java class or method as a function library

A function library registered on the schema is callable from **all** query languages, exactly like the SQL/JavaScript/Cypher functions above. Use `JavaClassFunctionLibraryDefinition` to expose every public method of a class, or `JavaMethodFunctionLibraryDefinition` to expose a single method. The

library name becomes the namespace and each method name becomes a function in it.

Given this class deployed in a JAR under `lib/`:

```
package com.example;

public class MathFunctions {
  // Instance method: invoked on a new instance created via the no-arg constructor
  public int sum(final int a, final int b) {
    return a + b;
  }

  // Static method: invoked directly, no instance needed
  public static double max(final double a, final double b) {
    return Math.max(a, b);
  }
}
```

Register it on the database schema (embedded usage shown):

```
// Register the whole class as a library named "math".
// Every public method (sum, max, ...) becomes a function in that library.
database.getSchema().registerFunctionLibrary(
    new JavaClassFunctionLibraryDefinition("math", "com.example.MathFunctions"));

// ...or pass the Class directly:
database.getSchema().registerFunctionLibrary(
    new JavaClassFunctionLibraryDefinition("math", com.example.MathFunctions.class));

// ...or register a single method as a one-function library. A non-static method is
invoked on a
// new instance of its class, created by the library through the public no-arg
constructor:
database.getSchema().registerFunctionLibrary(
    new JavaMethodFunctionLibraryDefinition("math",
        com.example.MathFunctions.class.getMethod("sum", Integer.TYPE, Integer.
TYPE)));

// ...or on an instance of your own, when the method needs state or dependencies the
no-arg
// constructor cannot supply:
database.getSchema().registerFunctionLibrary(
    new JavaMethodFunctionLibraryDefinition("math", new com.example.MathFunctions(),
        com.example.MathFunctions.class.getMethod("sum", Integer.TYPE, Integer.
TYPE)));
```

A library name can be registered only once. Re-registering the same name throws `IllegalArgumentException`. Remove it first with

> `database.getSchema().unregisterFunctionLibrary("math")`. *(Since v26.10.2)* unregistering a library defined in SQL, JavaScript or openCypher (one saved with the schema) is saved too, so it survives a reopen, and under HA it is replicated and must run on the leader: on a follower it throws `ServerIsNotTheLeaderException`. Unregistering a Java library like the one above changes only the open database, since such a library is never saved.

If a class has overloaded public methods (e.g. `format(String)` and `format(String, int)`), every overload is registered and the matching one is picked at call time by the number and, if needed, the type of the arguments passed - including varargs and standard Java numeric widening (an `int` argument matches a `long` parameter, for example). When several overloads accept the arguments, the one with the narrowest primitive parameters wins, as in Java: `twice(int)` over `twice(long)` for an `int` argument, `scale(long)` over `scale(double)`. A call that no overload accepts more specifically than the others (e.g. both `format(Object)` and `format(String)` for a `String` argument, whose reference types are not ranked) is rejected as ambiguous rather than silently picking one.

Once registered, call the function from any query language using the `library.name` form:

```
// Direct invocation through the API
Object r = database.getSchema().getFunction("math", "sum").execute(20, 7);  // 27

// From SQL (backtick-quote the library.function reference)
database.command("sql", "SELECT `math.sum`(20, 7) AS result");

// From Cypher
database.query("opencypher", "RETURN math.sum(20, 7) AS result");
```

Option B: The dedicated "java" query engine

The `java` query engine invokes a Java method directly as a command, without going through the schema. For security, every class or method must be added to an allow-list with `registerFunctions(…)` before it can be called; calling anything that was not registered throws a `SecurityException`.

```
// Allow-list a single method (Class::method) or a whole class
database.getQueryEngine("java").registerFunctions("com.example.MathFunctions::sum");
database.getQueryEngine("java").registerFunctions("com.example.MathFunctions"); // all
methods

// Invoke it (positional parameters follow the command string)
ResultSet result = database.command("java", "com.example.MathFunctions::sum", 5, 3);
int value = (int) result.next().getProperty("value");  // 8

// Remove the allow-list when done
database.getQueryEngine("java").unregisterFunctions();
```

The `java` engine supports only `command(…)` (not `query(…)`) and positional parameters; it matches

overloaded methods by argument count and type.

Registering Java functions on a standalone server

On a standalone server there is no application code to call `registerFunctionLibrary(…)`, so wrap the registration in a Server Plugin. Build a JAR containing both the function class and a plugin like the one below, place it in `lib/plugins/`, and enable it.

```
package com.example;

import com.arcadedb.ContextConfiguration;
import com.arcadedb.function.java.JavaClassFunctionLibraryDefinition;
import com.arcadedb.server.ArcadeDBServer;
import com.arcadedb.server.ServerPlugin;

public class MathFunctionsPlugin implements ServerPlugin {
  private ArcadeDBServer server;

  @Override
  public void configure(final ArcadeDBServer arcadeDBServer, final
ContextConfiguration configuration) {
    this.server = arcadeDBServer;
  }

  @Override
  // Run after the databases are open so they are available for registration
  public PluginInstallationPriority getInstallationPriority() {
    return PluginInstallationPriority.AFTER_DATABASES_OPEN;
  }

  @Override
  public void startService() {
    // Register the library on every database that needs it
    server.getDatabaseForOperator("mydb").getSchema().registerFunctionLibrary(
        new JavaClassFunctionLibraryDefinition("math", "com.example.MathFunctions"));
  }
}
```

Enable the plugin with the `arcadedb.server.plugins` setting, using the `<pluginName>:<pluginFullClassName>` format (comma-separate multiple plugins):

```
-Darcadedb.server.plugins=Math:com.example.MathFunctionsPlugin
```

After the server starts, the functions are available from SQL, Cypher and the other query languages (for example ``SELECT `math.sum(20, 7)` ``), including over the HTTP/REST API.

14.26.5. Best Practices

1. **Choose the Right Language**:
   - Use SQL for data transformations and aggregations
   - Use JavaScript for complex business logic and calculations
   - Use OpenCypher for graph traversals and pattern matching
   - Use Java for maximum performance
2. **Namespace Organization**: Group related functions under the same library namespace [source,sql] ---- DEFINE FUNCTION math.sum … DEFINE FUNCTION math.avg … DEFINE FUNCTION utils.formatDate … DEFINE FUNCTION graph.shortestPath … ----
3. **Function Naming**: Use descriptive names that indicate what the function does
4. **Performance Considerations**:
   - User functions are evaluated for each row/result
   - For heavy operations, consider caching or materialized views
   - Java functions provide the best performance
5. **Error Handling**: Functions should handle null inputs gracefully

14.26.6. Advantages Over Neo4j

Unlike Neo4j which only supports Java-based user-defined procedures requiring compilation and deployment:

 Dynamic Definition: Define functions at runtime without restart
 Persistent & Replicated: Survive restarts and are automatically shared across an HA cluster
 4 Languages: SQL, JavaScript, OpenCypher, and Java
 Universal Access: Call from any query language
 No Compilation: Interpreted languages work immediately
 Easy Management: Simple SQL commands to define/delete functions

14.27. SQL SELECT Statements Execution

The execution flow of a SELECT statement is made of many steps. Understanding these steps will help you to write better and more optimized queries.

The SELECT query execution, at a very high level, is made of three steps: - Query optimization - Creation of execution plans - Choice of the optimal execution plan - Actual execution

Query optimization

The first step for the query executor is to run a query optimizer. This operation can change the internal structure of the SQL statement to make it more efficient, preserving the same semantics of the original query.

Typical optimization steps are:

- Early calculation of expressions

eg. consider the following statement

```
SELECT FROM Person WHERE fullName = "John" + " " + "Smith"
```

The result of the string concatenation `"John" + " " + "Smith"` does not depend on the query context (eg. the content of a record in the result set), so it can be calculated only once in the execution phase. The result of the optimization of this query will be the equivalent of

```
SELECT FROM Person WHERE fullName = "John Smith"
```

Early calculation of sub-queries

eg. consider the following statement

```
SELECT FROM Person WHERE father in (SELECT FROM Person WHERE name = 'John')
```

The result of the subquery does not depend on the parent query context, so it can be executed only once, and then use the result as an argument for the parent query:

```
LET $a = (SELECT FROM Person WHERE name = 'John');
SELECT FROM Person WHERE father in $a
```

It is possible only if the subquery does not depend on the context of the parent query, so for example the following cannot be split:

```
SELECT FROM Person WHERE father in (SELECT FROM Person WHERE name = 'John' and surname
= $parent.$current.surname)
```

Refactoring of the WHERE conditions

eg. consider the following:

```
SELECT FROM Person
WHERE
(name = 'John' AND surname = 'Smith')
OR (name = 'John' AND surname = 'Doe')
OR (name = 'John' AND surname = 'Travolta')
OR (name = 'John' AND surname = 'Lennon')
OR (name = 'John' AND surname = 'Nash')
```

If the WHERE condition is evaluated as is, the condition `name = 'John'` has to be evaluated five

times for each record that does not have a 'John' as a name. This query can be rewritten as:

```
SELECT FROM Person
WHERE
name = 'John' AND (
  surname = 'Smith'
  OR surname = 'Doe'
  OR surname = 'Travolta'
  OR surname = 'Lennon'
  OR surname = 'Nash'
)
```

Sometimes, like in case of full type scan, this is convenient. In other cases it's not. Eg. if `Person` type has an index on `<name, surname>`, the original query can be executed as the union of five index lookups. The query optimizer will create multiple versions of optimized conditions, for different execution plans (see below).

Creation of execution plans

An execution plan is a sequence of operations that the query engine has to execute to calculate the query result.

Each step in the execution plan typically does a single operation, eg. fetch data from a type, filter results, calculate projections and so on.

For the same query, ArcadeDB can calculate multiple execution plans, based on involvement of indexes, optimized sorting and so on.

An execution plan has an execution cost that depends on the number of processed records, the number of operations performed and the elaboration time. The query executor uses the execution cost as the main criterion to choose the optimal execution plan.

Choice of the optimal execution plan

If the query executor produces multiple execution plans, then it has to choose the more convenient one to actually execute the query. This choice is made based on the execution cost: the execution plan with the minimum cost is chosen.

Actual execution

After choosing the optimal execution plan, it is just executed.

The execution of an execution plan is just the execution of all the steps that it represents.

Query Execution Plan

As described above, an execution plan is a sequence of steps that have to be executed to calculate a query result.

Different queries will have different execution plans.

The typical execution plan is made of the following steps:

- fetch from query target (that can be a type, a bucket, an index and so on)
- evaluate LET expressions
- calculate query projections
- filter results
- aggregate data (eg. aggregate functions + GROUP BY)
- `UNWIND` projections
- sort result (ORDER BY)
- SKIP
- LIMIT

Obviously, a simple query like `SELECT FROM Person` will have a very simple execution plan made of a single step (the fetch from `Person` type), while a complex query will have an execution plan made of multiple steps

To display the execution plan of a query, without executing it, you can just execute the query prefixing it with `EXPLAIN`, eg.

```
EXPLAIN SELECT FROM Person
```

Chapter 15. Cypher (openCypher)

15.1. Introduction

ArcadeDB provides native support for the Cypher query language through a new, high-performance implementation built from the ground up. This native OpenCypher module uses an ANTLR4-based parser with the official Cypher v25 grammar, providing excellent compatibility and performance.

The query language identifiers `opencypher` and `cypher` are interchangeable: both run on the same native OpenCypher engine.

15.1.1. Key Features

The native OpenCypher implementation provides:

- **High Performance**: Native direct execution, comparable to native SQL
- **Full Read/Write Support**: MATCH, CREATE, MERGE, SET, DELETE with automatic transaction handling
- **Advanced Pattern Matching**: Variable-length paths with path modes (WALK, TRAIL, ACYCLIC), OPTIONAL MATCH, multiple MATCH clauses
- **Query Composition**: UNION/UNION ALL for combining results, WITH for query chaining
- **Procedures and Profiling**: CALL for invoking built-in procedures and custom functions, PROFILE for performance analysis
- **Rich Function Library**: 23+ native Cypher functions plus bridge to 100+ ArcadeDB SQL functions
- **Cost-Based Optimizer**: Intelligent query planning with index selection and join ordering
- **100% Test Coverage**: Comprehensive test suite with 295+ passing tests

15.1.2. Using OpenCypher

To execute OpenCypher queries, use `opencypher` as the query language identifier.

From Java API:

```
ResultSet result = database.query("opencypher",
    "MATCH (p:Person) WHERE p.age >= $minAge RETURN p.name, p.age ORDER BY p.age",
    "minAge", 25);
```

Through Neo4j BOLT Protocol:

Connect using any Neo4j driver via the BOLT protocol:

```
Driver driver = GraphDatabase.driver("bolt://localhost:7687", AuthTokens.basic("root",
"password"));
Session session = driver.session(SessionConfig.forDatabase("mydatabase"));
Result result = session.run("MATCH (p:Person) WHERE p.age >= 25 RETURN p.name, p.age
ORDER BY p.age");
```

Through PostgreSQL Driver:

Prefix your query with `{opencypher}`:

```
"{opencypher} MATCH (p:Person) WHERE p.age >= 25 RETURN p.name, p.age ORDER BY p.age"
```

Through HTTP/JSON API (GET - idempotent queries):

```
curl "http://localhost:2480/query/graph/opencypher/MATCH%20(p:Person)%20RETURN%20p"
```

Through HTTP/JSON API (POST - updates):

```
curl -X POST "http://localhost:2480/command/graph" \
  -d "{'language': 'opencypher', 'command': 'CREATE (p:Person {name: \"Alice\", age:
30})'}"
```

15.1.3. Using Record IDs (RIDs)

You can use ArcadeDB's RecordIDs (RID) in Cypher queries to start from a specific vertex. RIDs in Cypher are always strings, so they must be contained between single or double quotes.

```
MATCH (m:Movie)<-[a:ACTED_IN]-(p:Person) WHERE id(m) = '#1:0' RETURN *
```

15.1.4. Parameters

OpenCypher supports parameterized queries using the `$paramName` syntax:

```
MATCH (p:Person) WHERE p.age >= $minAge AND p.city = $city RETURN p
```

Parameters are passed as key-value pairs after the query string:

```
ResultSet result = database.query("opencypher",
    "MATCH (p:Person) WHERE p.age >= $minAge RETURN p",
    "minAge", 25);
```

15.1.5. Automatic Transaction Handling

A write statement (CREATE, SET, DELETE, MERGE, REMOVE, FOREACH, a write procedure) automatically handles its transaction:

- If no transaction is active, the whole statement runs in one transaction, committed when the statement has run to its end, as over HTTP and as an implicit transaction in Neo4j. A statement that fails leaves nothing behind, and one that meets a concurrent conflict is run again from the committed state, up to `arcadedb.txRetries` times *(since v26.11.1; before, each write clause committed its own transaction row by row, so a failing statement kept the writes made before the error)*
- If a transaction is already active, the statement runs in it and leaves the commit or rollback to its owner
- A `CALL { … } IN TRANSACTIONS` subquery still commits each batch on its own, so a statement using one is not run again on a conflict

```
// Automatic transaction - the whole statement commits once it completes
database.command("opencypher", "CREATE (n:Person {name: 'Alice'})");

// Manual transaction - you control commit/rollback
database.transaction(() -> {
    database.command("opencypher", "CREATE (a:Person {name: 'Alice'})");
    database.command("opencypher", "CREATE (b:Person {name: 'Bob'})");
    database.command("opencypher", "MATCH (a:Person {name: 'Alice'}), (b:Person {name:
'Bob'}) CREATE (a)-[:KNOWS]->(b)");
    // Commits when lambda completes successfully
});
```

15.1.6. Performance Considerations

The native OpenCypher implementation includes a cost-based query optimizer that:

- Automatically selects optimal indexes for node lookups. When several predicates on the same node can use an index (an equality, an `IN` list, an `OR` of equalities or a range), each one is costed and the cheapest drives the lookup, so a `UNIQUE` index resolved by its whole key is preferred over a non-unique one whatever the properties are called or which operator they use. Two candidates of the same cost are decided by the order the query writes them in *(since v26.11.1)*
- Uses range index scans for inequality predicates (`>`, `<`, `>=`, `⇐`)
- Starts a pattern that has a node with no label from the index of a labelled node *(since v26.12.1)*. In `MATCH (p:Person)-[:KNOWS]→(q) WHERE p.id < 500` the unlabelled `q` is reached through the relationship and `p` is read by an index seek, an `IN` seek or a range scan, instead of walking every `Person`. Written from the other end, `(q)-[:KNOWS]→(p:Person)`, the pattern starts from `p` too. A pattern with no labelled node, a labelled node no index serves (the pattern would start from a scan of the whole label) or a variable-length hop into the unlabelled node is still planned step by step, as before. The index lookups of this plan ask the read permission of the type, and of its subtypes, each time the statement runs, so a user who may not read the type is refused

whatever the index holds

- Starts a single fixed-length hop from its far end when the first node has no label and no condition and the far node is pinned on every column of a `UNIQUE` index of its label *(since v26.12.1)*, also in a statement planned step by step such as a `MATCH` after `UNWIND` or `WITH`. `UNWIND $keys AS k MATCH (x)-[:OF]→(:Doc {key: k})` finds each `Doc` through the index and walks its relationships backwards, instead of scanning every vertex of the database once per row. The pin can be an inline property map (literals, parameters or a variable bound by an earlier clause) or a `WHERE` equality or `IN` list. The written order is kept for an `OPTIONAL MATCH`, a path variable, a variable-length hop, a relationship with no type or of a `UNIDIRECTIONAL` type, a non-unique key and a first node that has a label or a condition. The rows of a `MATCH` without `ORDER BY` can come in a different order
- Drives a single-hop relationship pattern from the edge type's own index when the `WHERE` clause pins the relationship's indexed properties and neither endpoint is selective, e.g. `MATCH ()-[t:TRANSFER]→() WHERE t.transactionId IN ['A', 'B'] RETURN t`. An equality, an `IN` list or an `OR` of equalities on the same property all qualify, and a leading prefix of a composite index is enough: with an index on `(transactionId, date)`, filtering `transactionId` alone walks that prefix of the index instead of scanning every edge. The plan shows it as `MATCH EDGE BY INDEX`
- Asks the read permission of the type, and of its subtypes, before any index lookup *(since v26.12.1)*: the lookup of a labelled node (an equality, an `IN` list or a leading prefix of a composite index) asks it for the label, and `MATCH EDGE BY INDEX` asks it for the relationship type and for the labels written on both endpoints. A user who may not read one of them is refused whatever the index holds, as a scan of the type refuses him. Before 26.12.1 the node lookup was refused only when the index held the value (and answered no rows when it did not), and the relationship lookup returned the relationships that end in a type the user may not read
- Optimizes join order based on cardinality estimation
- Detects bounded patterns for efficient relationship traversal
- Joins the parts of a pattern no relationship connects - `MATCH (a:Person), (c:City)`, or two `MATCH` clauses - on the `WHERE` conjuncts that relate them instead of crossing every row with every row. A conjunct that reads one part alone narrows that part before the join. Cypher does not fix the order conjuncts are evaluated in, so a conjunct that reads one part, like either side of a join key, is evaluated once per row of that part rather than per pair, and ahead of the conjuncts written before it: `rand()` there is drawn per row, and an expression that can fail, such as `1 / 0`, can fail on a row the written order would have skipped. A `c.property = expression` conjunct over an indexed property seeks the index once per row of the other part (`IndexNestedLoopJoin` in the plan), which holds nothing in memory; any other `left = right` conjunct is answered by a hash join (`ValueHashJoin`). Only parts nothing relates are crossed (`CartesianProduct`). The rows a join buffers count against `arcadedb.queryMaxHeapElementsAllowedPerOp` and reserve their heap from the budget all the running queries share, `arcadedb.queryMaxHeapRAM`, and past a few thousand rows a query that only reads, or only adds entities (`CREATE`, `MERGE` without `SET` actions), holds them by record identity rather than in full (`[compact buffer]` in the plan)

For best performance:

- Create indexes on frequently queried properties
- Use labeled nodes in MATCH patterns for faster type filtering

- Prefer specific property constraints over full scans
- Use parameters for queries executed multiple times (enables query plan caching). Since v26.10.2 queries that differ only in the literal values they embed share a cached plan too (see the `queryLiteralParameterization` setting), but parameters also keep values out of the query text

15.1.7. Resources

- OpenCypher Specification
- Neo4j Cypher Manual (reference)
- Learn Cypher in Y Minutes

15.2. Cypher Tutorial

Prefer a guided, video-based walkthrough? The ArcadeDB Academy offers a **free** course on Cypher with hands-on exercises and a certification at the end.

This tutorial introduces you to Cypher, a declarative graph query language. By the end of this tutorial, you'll be able to create, query, and manipulate graph data in ArcadeDB using Cypher.

15.2.1. What is Cypher?

Cypher is a pattern-matching query language designed specifically for graphs. It uses ASCII-art syntax to represent graph patterns, making queries intuitive and readable. For example, `(a)-[:KNOWS]→(b)` visually represents a KNOWS relationship from node `a` to node `b`.

15.2.2. Setting Up

Before running Cypher queries, ensure you have:

1. ArcadeDB server running
2. A database created (we'll use `tutorial` in examples)

Create vertex types for our tutorial:

```
CREATE VERTEX TYPE Person
CREATE VERTEX TYPE Movie
CREATE VERTEX TYPE Company
CREATE EDGE TYPE ACTED_IN
CREATE EDGE TYPE DIRECTED
CREATE EDGE TYPE WORKS_AT
CREATE EDGE TYPE KNOWS
```

15.2.3. Part 1: Creating Data

Creating Nodes

Use the `CREATE` clause to add nodes (vertices) to your graph.

Create a single node:

```
CREATE (p:Person {name: 'Alice', age: 30, city: 'New York'})
```

This creates a node with:

- Label `Person` (the type)
- Properties: `name`, `age`, and `city`

Create multiple nodes:

```
CREATE (a:Person {name: 'Bob', age: 35}),
       (b:Person {name: 'Charlie', age: 28}),
       (c:Person {name: 'Diana', age: 32})
```

Creating Relationships

Relationships (edges) connect nodes. Create them with the arrow syntax.

Create a relationship between new nodes:

```
CREATE (a:Person {name: 'Eve'})-[:KNOWS {since: 2020}]->(b:Person {name: 'Frank'})
```

Create a relationship between existing nodes:

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2019}]->(b)
```

Sample Dataset

Let's create a sample movie database:

```
// Create people
CREATE (keanu:Person {name: 'Keanu Reeves', born: 1964})
CREATE (carrie:Person {name: 'Carrie-Anne Moss', born: 1967})
CREATE (laurence:Person {name: 'Laurence Fishburne', born: 1961})
CREATE (lana:Person {name: 'Lana Wachowski', born: 1965})
CREATE (lilly:Person {name: 'Lilly Wachowski', born: 1967})

// Create movies
CREATE (matrix:Movie {title: 'The Matrix', released: 1999})
CREATE (reloaded:Movie {title: 'The Matrix Reloaded', released: 2003})
```

```
// Create relationships
MATCH (keanu:Person {name: 'Keanu Reeves'}), (matrix:Movie {title: 'The Matrix'})
CREATE (keanu)-[:ACTED_IN {role: 'Neo'}]->(matrix)

MATCH (carrie:Person {name: 'Carrie-Anne Moss'}), (matrix:Movie {title: 'The Matrix'})
CREATE (carrie)-[:ACTED_IN {role: 'Trinity'}]->(matrix)

MATCH (laurence:Person {name: 'Laurence Fishburne'}), (matrix:Movie {title: 'The
Matrix'})
CREATE (laurence)-[:ACTED_IN {role: 'Morpheus'}]->(matrix)

MATCH (lana:Person {name: 'Lana Wachowski'}), (matrix:Movie {title: 'The Matrix'})
CREATE (lana)-[:DIRECTED]->(matrix)

MATCH (lilly:Person {name: 'Lilly Wachowski'}), (matrix:Movie {title: 'The Matrix'})
CREATE (lilly)-[:DIRECTED]->(matrix)
```

15.2.4. Part 2: Reading Data

Basic MATCH Queries

The `MATCH` clause finds patterns in your graph.

Find all nodes of a type:

```
MATCH (p:Person)
RETURN p
```

Find nodes with specific properties:

```
MATCH (p:Person {name: 'Alice'})
RETURN p
```

Return specific properties:

```
MATCH (p:Person)
RETURN p.name, p.age
```

Filtering with WHERE

Use `WHERE` for complex conditions:

```
MATCH (p:Person)
WHERE p.age > 30
```

```
RETURN p.name, p.age
```

Multiple conditions:

```
MATCH (p:Person)
WHERE p.age >= 25 AND p.age <= 35 AND p.city IS NOT NULL
RETURN p.name, p.age, p.city
```

String matching:

```
// Names starting with 'A'
MATCH (p:Person) WHERE p.name STARTS WITH 'A' RETURN p.name

// Names containing 'an'
MATCH (p:Person) WHERE p.name CONTAINS 'an' RETURN p.name

// Email pattern matching
MATCH (p:Person) WHERE p.email =~ '.*@gmail.com' RETURN p.name, p.email
```

IN operator:

```
MATCH (p:Person)
WHERE p.name IN ['Alice', 'Bob', 'Charlie']
RETURN p
```

Traversing Relationships

Find connected nodes by specifying relationship patterns:

Find direct connections:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, m.title
```

Find connections with relationship properties:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p.name, r.role, m.title
```

Find bidirectional connections (either direction):

```
MATCH (p:Person)-[:KNOWS]-(other:Person)
RETURN p.name, other.name
```

Multi-hop traversals:

```
MATCH (a:Person)-[:KNOWS]->(b:Person)-[:KNOWS]->(c:Person)
WHERE a.name = 'Alice'
RETURN a.name, b.name, c.name
```

Variable-Length Paths

Find paths of varying lengths:

```
// Find all people within 1 to 3 hops
MATCH (a:Person {name: 'Alice'})-[:KNOWS*1..3]->(b:Person)
RETURN b.name
```

Named paths:

```
MATCH p = (a:Person)-[:KNOWS*1..3]->(b:Person)
WHERE a.name = 'Alice'
RETURN p
```

OPTIONAL MATCH

Like a LEFT OUTER JOIN - returns NULL for missing matches:

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[:ACTED_IN]->(m:Movie)
RETURN p.name, m.title
```

This returns all people, with `m.title` as NULL for those who haven't acted in any movie.

Sorting and Pagination

ORDER BY:

```
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC
```

Multiple sort keys:

```
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC, p.name ASC
```

SKIP and LIMIT:

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 10
LIMIT 5
```

15.2.5. Part 3: Aggregations

Basic Aggregations

```
// Count all people
MATCH (p:Person)
RETURN count(p) AS totalPeople

// Count, sum, average
MATCH (p:Person)
RETURN count(p) AS total, sum(p.age) AS totalAge, avg(p.age) AS averageAge

// Min and max
MATCH (p:Person)
RETURN min(p.age) AS youngest, max(p.age) AS oldest
```

Grouping (Implicit GROUP BY)

Cypher automatically groups by non-aggregated expressions:

```
// Count people by city
MATCH (p:Person)
RETURN p.city, count(p) AS residents
ORDER BY residents DESC
```

Multiple grouping keys:

```
MATCH (p:Person)-[:WORKS_AT]->(c:Company)
RETURN c.name, p.department, count(p) AS employees
ORDER BY c.name, employees DESC
```

Aggregating map projection: a map projection that contains an aggregate groups by its own variable, so it answers one row per value of that variable, as in Neo4j:

```
// One row per person, each with the names of the people they know
MATCH (p:Person)-[:KNOWS]->(f)
```

```
RETURN p{.name, friends: collect(f.name)} AS person
```

Ambiguous aggregation: any other expression that mixes an aggregate with a variable must read only grouping keys. A map literal, a list, a `CASE` or a `reduce()` reading a variable that is not one fails with `AmbiguousAggregationExpression`, the error arithmetic such as `p.age + count(*)` already raises:

```
// Fails: p is not a grouping key of the map literal
MATCH (p:Person)-[:KNOWS]->(f)
RETURN {name: p.name, friends: count(f)} AS person

// Works: p.name is projected as a grouping key
MATCH (p:Person)-[:KNOWS]->(f)
RETURN p.name AS name, {name: p.name, friends: count(f)} AS person
```

Since v26.12.1. Before this release, an aggregating map literal or map projection that read a variable that was not a grouping key folded every input row into a single row, and read that variable from an arbitrary row. The map projection now groups by its variable, and the map literal (or list, `CASE`, `reduce()`) is refused as ambiguous. An aggregate inside a pattern comprehension (`[(n)-→(m) | count(*)]`) is refused too, as one inside a list comprehension already was.

COLLECT

Collect values into a list:

```
// Collect all names
MATCH (p:Person)
RETURN collect(p.name) AS allNames

// Group and collect
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN m.title, collect(p.name) AS actors
```

15.2.6. Part 4: Data Modification

Updating with SET

```
// Update a single property
MATCH (p:Person {name: 'Alice'})
SET p.age = 31

// Update multiple properties
MATCH (p:Person {name: 'Alice'})
SET p.age = 31, p.city = 'Boston'
```

MERGE (Upsert)

MERGE finds or creates a pattern:

```
// Create if not exists
MERGE (p:Person {name: 'Grace'})

// With ON CREATE and ON MATCH actions
MERGE (p:Person {name: 'Grace'})
ON CREATE SET p.created = true, p.visits = 1
ON MATCH SET p.visits = p.visits + 1
```

MERGE relationships:

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MERGE (a)-[r:KNOWS]->(b)
ON CREATE SET r.since = 2024
ON MATCH SET r.lastContact = 2024
```

DELETE

Delete a relationship:

```
MATCH (a:Person {name: 'Alice'})-[r:KNOWS]->(b:Person {name: 'Bob'})
DELETE r
```

Delete a node (must have no relationships):

```
MATCH (p:Person {name: 'TestPerson'})
DELETE p
```

DETACH DELETE (delete node and its relationships):

```
MATCH (p:Person {name: 'TestPerson'})
DETACH DELETE p
```

15.2.7. Part 5: Advanced Queries

WITH Clause

Use `WITH` to chain query parts:

```
// Filter and project before continuing
MATCH (p:Person)
```

```
WITH p.name AS name, p.age AS age
WHERE age > 30
RETURN name ORDER BY name
```

Aggregation with WITH:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH m.title AS movie, count(p) AS actorCount
WHERE actorCount > 2
RETURN movie, actorCount
```

UNWIND

Expand a list into rows:

```
// Unwind a literal list
UNWIND [1, 2, 3] AS x
RETURN x

// Unwind a property array
MATCH (p:Person {name: 'Alice'})
UNWIND p.hobbies AS hobby
RETURN p.name, hobby

// Create multiple nodes from a list
UNWIND ['Alice', 'Bob', 'Charlie'] AS name
CREATE (p:Person {name: name})
```

Pattern Predicates

Check if patterns exist:

```
// Find people who know someone
MATCH (p:Person)
WHERE (p)-[:KNOWS]->()
RETURN p.name

// Find people who don't know anyone
MATCH (p:Person)
WHERE NOT (p)-[:KNOWS]->()
RETURN p.name

// Find people who know a specific person
MATCH (a:Person {name: 'Alice'}), (p:Person)
WHERE (p)-[:KNOWS]->(a)
RETURN p.name AS knowsAlice
```

A pattern predicate only tests existence, so it cannot be counted: `WHERE sizep)-[:KNOWS]→( > 5` is refused (since v26.12.1). Count with a `COUNT { }` subquery instead: `WHERE COUNT { (p)-[:KNOWS]→() } > 5`.

A pattern predicate that names a node or relationship variable whose value is null, as an `OPTIONAL MATCH` that found nothing leaves it, is false, and `NOT` of it is true: after `OPTIONAL MATCH (p)-[:MANAGES]→(m)`, `WHERE (m)-→()` drops the rows where `m` is null and `WHERE NOT (m)-→()` keeps them (since v26.12.1, as Neo4j does).

CASE Expressions

Conditional logic in queries:

```
MATCH (p:Person)
RETURN p.name,
       CASE
         WHEN p.age < 18 THEN 'minor'
         WHEN p.age < 65 THEN 'adult'
         ELSE 'senior'
       END AS ageGroup
```

15.2.8. Part 6: Using Functions

String Functions

```
MATCH (p:Person)
RETURN toUpper(p.name) AS upperName,
       toLower(p.name) AS lowerName,
       substring(p.name, 0, 3) AS initials
```

Math Functions

```
MATCH (p:Person)
RETURN p.name,
       abs(p.balance) AS absBalance,
       round(p.salary / 1000.0) AS salaryK
```

List Functions

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p.name AS actor, collect(m.title) AS movies
RETURN actor,
       size(movies) AS movieCount,
       head(movies) AS firstMovie,
       last(movies) AS lastMovie
```

Node/Relationship Functions

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN id(p) AS personId,
       labels(p) AS personLabels,
       type(r) AS relationshipType,
       keys(m) AS movieProperties
```

15.2.9. Part 7: Best Practices

1. Use Labels

Always specify labels in MATCH patterns for better performance:

```
// Good - uses index on Person
MATCH (p:Person {name: 'Alice'}) RETURN p

// Avoid - scans all nodes
MATCH (n {name: 'Alice'}) RETURN n
```

2. Create Indexes

Create indexes on frequently queried properties:

```
CREATE INDEX ON Person (name) NOTUNIQUE
CREATE INDEX ON Movie (title) NOTUNIQUE
```

3. Use Parameters

For repeated queries, use parameters instead of string concatenation:

```
MATCH (p:Person)
WHERE p.name = $name AND p.age >= $minAge
RETURN p
```

Since v26.10.2 a query that writes its values in the text (`WHERE p.name = 'Alice'`) is not parsed and planned again for every value: ArcadeDB extracts the literals into parameters itself, so all the queries of the same shape share one cached plan (see the `queryLiteralParameterization` setting). Parameters remain the better choice: they keep values out of the query text, which rules out injection, and they let every value share the plan, including the ones written where a literal is kept, such as `SKIP` and `LIMIT`.

4. Limit Results

Use LIMIT for exploratory queries:

```
MATCH (p:Person)-[:KNOWS*1..5]->(other)
RETURN p.name, other.name
LIMIT 100
```

5. Use WITH for Complex Queries

Break complex queries into steps with WITH:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, count(m) AS movieCount
WHERE movieCount > 5
MATCH (p)-[:DIRECTED]->(d:Movie)
RETURN p.name, movieCount, collect(d.title) AS directed
```

15.3. Cypher Clauses Reference

15.3.1. MATCH

The `MATCH` clause is used to search for patterns in the graph. It is the primary read operation in Cypher.

Syntax:

```
MATCH pattern [, pattern ...]
[WHERE condition]
```

Pattern Syntax:

- `(variable)` - Match any node
- `(variable:Label)` - Match nodes with specific label
- `(variable:Label {prop: value})` - Match nodes with properties
- `(a)-[r]→(b)` - Match directed relationship
- `(a)-[r]-(b)` - Match relationship in either direction
- `(a)-[r:TYPE]→(b)` - Match relationship with specific type
- `(a)-[r:TYPE1|TYPE2]→(b)` - Match relationship with multiple types
- `(a)-[*min..max]→(b)` - Match variable-length path

Examples:

```
// Match all Person nodes
MATCH (p:Person)
RETURN p
```

```
// Match with property filter
MATCH (p:Person {name: 'Alice', age: 30})
RETURN p

// Match relationships
MATCH (p:Person)-[r:KNOWS]->(friend:Person)
RETURN p.name, friend.name

// Match with relationship properties
MATCH (p:Person)-[r:WORKS_AT {since: 2020}]->(c:Company)
RETURN p.name, c.name

// Variable-length paths
MATCH (a:Person)-[:KNOWS*1..3]->(b:Person)
WHERE a.name = 'Alice'
RETURN b.name

// Named paths
MATCH path = (a:Person)-[:KNOWS*]->(b:Person)
WHERE a.name = 'Alice'
RETURN path

// Comma-separated patterns (Cartesian product)
MATCH (a:Person), (b:Company)
RETURN a.name, b.name

// Comma-separated patterns related in WHERE: joined, not crossed
MATCH (a:Person), (b:Company)
WHERE b.id = a.companyId AND b.country = 'IT'
RETURN a.name, b.name

// Multiple MATCH clauses
MATCH (a:Person {name: 'Alice'})
MATCH (b:Person {name: 'Bob'})
RETURN a, b
```

15.3.2. Path Mode (WALK, TRAIL, ACYCLIC)

Path modes control how variable-length traversals handle cycles. They are placed after the optional path variable assignment and before the pattern.

Syntax:

```
MATCH [variable =] [WALK | TRAIL | ACYCLIC] (a)-[*min..max]->(b)
```

Modes:

Mode	Behavior	Default?
TRAIL	Edges must be unique per path. Vertices **can** repeat.	Yes (default)
ACYCLIC	No vertex repeated per path (strongest restriction). Implies edge uniqueness.	
WALK	No restrictions — vertices and edges can be revisited. Requires an explicit max hop bound.	

TRAIL is the default path mode per the GQL/Cypher v25 standard. If no mode is specified, TRAIL semantics apply.

Examples:

```
// TRAIL (default): edges unique, vertices can repeat.
// On a cycle A→B→C→D→A, this can traverse the full cycle.
MATCH (a:Person {name: 'Alice'})-[:KNOWS*1..5]->(b)
RETURN b.name

// ACYCLIC: no vertex revisited — stops before completing a cycle.
MATCH ACYCLIC (a:Person {name: 'Alice'})-[:KNOWS*1..5]->(b)
RETURN b.name

// WALK: no restrictions. Edges and vertices can be reused.
// Requires explicit max hops to prevent infinite traversal.
MATCH WALK (a:Person {name: 'Alice'})-[:KNOWS*1..6]->(b)
RETURN b.name

// Path mode with named path variable
MATCH p = TRAIL (a:Person)-[:KNOWS*1..4]->(b)
RETURN p, length(p)
```

Comparison on a cyclic graph (A → B → C → D → A):

Mode	Reachable from A with *1..5 hops
TRAIL	B, C, D, **A** (vertex A revisited after full cycle — edges stay unique)
ACYCLIC	B, C, D (stops — cannot revisit A)
WALK	B, C, D, A, B, C, … (repeats edges and vertices up to max hops)

WALK mode without an explicit maximum hop bound will throw an error. Always specify an upper bound, e.g., [*1..10].

The path mode applies to a variable-length pattern predicate too: WHERE (a)-[*]->(a) is true when a cycle leads back to a, as MATCH (a)-[*]->(a) finds it, and WHERE (a)-[*0..]->(a) is true through the zero-length path.

15.3.3. Matching a Bound Relationship List

When the relationship variable of a variable-length pattern is already bound to a list of relationships by an earlier clause, the pattern matches exactly that list: it does not expand the graph, it checks that the relationships, in list order, form a path leaving the left-hand node in the pattern's direction.

```
MATCH (:A)-[r1]->(:B)-[r2]->(:C)
WITH [r1, r2] AS rs
MATCH (first)-[rs*]->(second)
RETURN first, second   // one row: (:A), (:C)
```

- The list is read in written order: its first relationship touches the left-hand node of the pattern. With the list above, `(second)<-[rs*]-(first)` matches nothing, while `[r2, r1]` would match it.
- Every other constraint of the pattern still applies: the hop range (`-[rs*1..1]->` refuses a two-element list), the relationship types, the inline property map, the inline `WHERE` and the path mode (under the default `TRAIL`, a list naming the same relationship twice matches nothing).
- A `null` list, or a list containing `null`, matches nothing. A value that is not a list of relationships raises a `TypeError`.
- The same rule applies to a pattern predicate (`WHERE (a)-[rs*]->(b)`), a pattern comprehension and an `EXISTS { }` or `COUNT { }` subquery.

Inside a pattern comprehension, the relationship variable of a variable-length hop is the list of relationships walked, as in `MATCH`: `[(a)-[r*]->(x) | size(r)]` returns the length of each path, and a zero-length hop binds the empty list.

A dangling entry is a relationship whose record is gone while the edge lists of its two endpoints still name it, left behind by a data-integrity anomaly (an incomplete replication, a partially rolled-back transaction). A healthy database holds none. A hop that reads the relationship (a relationship variable read downstream, `size(r)`) skips such an entry. A hop that answers from the edge lists without reading an edge follows it, as it follows a live relationship: the fixed-length hop (`-[:R]->`), `out()` and `in()`, everything that counts the entries of a vertex (its degree, `count(*)` over a hop), and, by default, the variable-length hop (`*`, `*2..4`, undirected, every path mode). The way to remove the anomaly is `CHECK DATABASE FIX`, which prunes the dangling entries; after it every spelling of a hop agrees.

On a database suspected to hold dangling entries, set `arcadedb.graph.ghostEdgeProbe` to `true` until it is repaired: the variable-length hops then ask the bucket, once per vertex reached and without reading a record, whether the relationship of the entry still exists, and skip the dangling entries and report them in the log, so `RETURN b.name` and `RETURN b.name, size(r)` agree. Another, live, relationship to the same vertex still reaches it. An undirected hop over an edge type declared unidirectional is probed too.

Since v26.12.1 `arcadedb.graph.ghostEdgeProbe` is `false` by default. In 26.11.1 it was `true`: a variable-length hop skipped dangling entries on every database and paid one edge-bucket page read per vertex reached, about a quarter to a third of a hop

> that reaches many vertices, for an anomaly only a corrupt database can hold. A database that holds dangling entries now answers a variable-length hop as it answers the fixed-length one, until `CHECK DATABASE FIX` prunes them. Set the setting to `true` to restore the 26.11.1 behavior.

15.3.4. Distinct End Vertices of a Variable-Length Pattern

A variable-length hop enumerates paths: a vertex that many paths lead to is reached once per path. When nothing in the query can tell two paths between the same pair of vertices apart, ArcadeDB does not enumerate them. It walks the graph breadth first with a visited set and reports each end vertex once per start vertex, so the time grows with the number of vertices reached and not with the number of paths. That is the "everything this depends on" or "everything that depends on this" question over a graph with many paths onto few vertices:

```
MATCH (a:Package)<-[:DEPENDS_ON*1..12]-(b:Package)
WHERE a.name = 'core'
RETURN count(DISTINCT b)
```

The answer holds the same rows and values as the one path enumeration gives, in every direction and every path mode; only their order can differ, see below. The shortcut applies when the statement is nothing but one non-optional `MATCH` with one hop, followed by `RETURN`, and:

- the lower bound of the hop is 0 or 1 (`*`, `*..n`, `*0..n` and `*1..n`);
- there is no named path and the relationship variable is not used anywhere (not in `WHERE`, `RETURN`, `ORDER BY` or an inline `WHERE`);
- the `RETURN` is either `DISTINCT` over the start or end vertices or their properties, or it groups by such values and aggregates them with `count(DISTINCT …)`, `collect(DISTINCT …)`, `sum(DISTINCT …)`, `avg(DISTINCT …)`, `min(…)` or `max(…)`. `ORDER BY` may use those values or the output names.

Anything else keeps enumerating paths: a plain `RETURN b`, `count(b)`, `size(r)`, a second `MATCH`, a `WITH`, `OPTIONAL MATCH`, and a lower bound of 2 or more. Use `EXPLAIN` to see which one a query gets: the hop is shown as `[DISTINCT ENDPOINTS]`. The traversal is lazy, so `LIMIT` stops it early. The order of the rows, and of the list built by `collect(DISTINCT …)`, is the breadth-first order of the walk, while path enumeration meets the vertices depth first: the same data can come back in another order once a statement stops being eligible (a named path or a second `MATCH` is enough). As in every `MATCH` without `ORDER BY`, no order is promised. A statement that reads the order of the collected list, such as `head(collect(DISTINCT b.name))` or `collect(DISTINCT b.name)[0]`, is not one of the shapes above and keeps the order of the path enumeration.

15.3.5. Named Paths

Assigning a variable to a pattern with `=` binds the whole matched path to that variable, which can then be passed to `length()`, `nodes()` and `relationships()` (see Expressions and Functions).

Syntax:

```
MATCH variable = pattern
```

A pattern with no relationship binds a **zero-length path**: one node, no relationships.

Examples:

```
// A one-hop path: length 1, two nodes, one relationship
MATCH p = (a:Person {name: 'Alice'})-[:KNOWS]->(b:Person)
RETURN length(p), nodes(p), relationships(p)

// A zero-length path: length 0, nodes(p) is [Alice], relationships(p) is []
MATCH p = (n:Person {name: 'Alice'})
RETURN length(p), nodes(p), relationships(p)
```

A named path can be one part of a comma-separated pattern, in any position, and the node it covers may already be bound by an earlier part or an earlier clause:

```
MATCH (x:Person)
OPTIONAL MATCH p = (x), (x)-[:KNOWS]->(z)
RETURN x.name, p, z.name
```

Inside an `OPTIONAL MATCH`, a path variable follows the same rule as every other variable the clause introduces: it is bound when the **whole** pattern matches and NULL when it does not. In the example above `p` is a zero-length path on the rows where `x` has an outgoing `KNOWS`, and NULL on the rows where it has none - even though `(x)` on its own would always match.

Returning a path. A path returned as it is reaches the client as one path value, whether a `MATCH` pattern, `shortestPath()`, `allShortestPaths()`, a path procedure or `apoc.path.create`/`slice`/`combine` built it. A Bolt driver receives a `Path`. Over HTTP, gRPC and the PostgreSQL protocol, which have no path type, it is the object `{"_type": "path", "nodes": [...], "relationships": [...], "length": n}`, with every node and relationship written whole, and Studio's graph view draws its nodes and relationships. *(Since v26.10.2)* See reference/graph-algorithms/path-finding.pdf.

15.3.6. OPTIONAL MATCH

`OPTIONAL MATCH` works like `MATCH`, but returns NULL values for missing parts of the pattern instead of filtering out rows. It behaves like a LEFT OUTER JOIN in SQL.

Syntax:

```
MATCH pattern
```

```
OPTIONAL MATCH optional_pattern
[WHERE condition]
RETURN ...
```

Examples:

```
// Find all people and their optional movies
MATCH (p:Person)
OPTIONAL MATCH (p)-[r:ACTED_IN]->(m:Movie)
RETURN p.name, m.title

// With WHERE clause scoped to OPTIONAL MATCH
MATCH (p:Person)
OPTIONAL MATCH (p)-[:KNOWS]->(friend:Person)
WHERE friend.age > 30
RETURN p.name, friend.name
```

The WHERE clause after OPTIONAL MATCH filters the optional match results, but preserves rows where the match failed (with NULL values).

Testing for absence. An `OPTIONAL MATCH` whose new variable is then tested for `IS NULL` keeps exactly the rows for which the pattern does not exist, which is what `WHERE NOT pattern` says. Since v26.10.2 it is planned that way: when the variables the `OPTIONAL MATCH` introduces are read by nothing but that test (a conjunct of the `WHERE` of the `WITH`, or `FILTER`, right after it), the nodes it shares with the earlier `MATCH` clauses are never null, and that `WITH` neither aggregates nor orders nor pages, the two spellings below run the same plan, including the anti-join count push-down (issue #9598):

```
MATCH (t1:Tag)<-[:HAS_TAG]-(m:Message)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(t2:Tag)
OPTIONAL MATCH (c)-[h:HAS_TAG]->(t1)
WITH t1, t2, h WHERE t1 <> t2 AND h IS NULL
RETURN count(*)

MATCH (t1:Tag)<-[:HAS_TAG]-(m:Message)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(t2:Tag)
WHERE NOT (c)-[:HAS_TAG]->(t1) AND t1 <> t2
RETURN count(*)
```

A variable read anywhere else (returned, counted, renamed, carried to a `RETURN *`), an `OPTIONAL MATCH` with its own `WHERE` or starting from a node an earlier `OPTIONAL MATCH` may have left null, and a statement that writes keep the `OPTIONAL MATCH` as written.

In a read-only query, when consecutive rows hold the same values for everything an `OPTIONAL MATCH` reads, ArcadeDB reuses the matches it found for the previous row instead of searching again. This matters after a clause that turns one node into many rows: an `OPTIONAL MATCH (r)-[:LOW]→(l)` that follows `OPTIONAL MATCH (r)-[:HIGH]→(h)` looks up the `LOW` relationships of `r` once, not once per `h`. `PROFILE` shows it as `[matches reused per …]` with `match runs` and `reused` counters. The result is the

same. *(Since v26.11.1)*

15.3.7. SEARCH (vector index)

(Since v26.11.1) The Cypher 25 `SEARCH` subclause of a `MATCH` runs a vector index search first and binds a node variable to the `k` nearest records, with their score:

```
MATCH (d:Doc) [WHERE w]
SEARCH d IN (VECTOR INDEX docEmbeddings FOR $queryVector [WHERE p] LIMIT 3) SCORE AS
score
RETURN d.title, score
```

- The pattern and the `WHERE` of the `MATCH` then filter those rows, as Neo4j applies them. The `WHERE` inside the parentheses is tested on each candidate during the search, and the search widens until it has `k` matching rows. Before v26.11.1 the subclause was parsed and dropped, so every `Doc` was answered and a `SCORE AS` variable was undefined.
- `FOR`, `LIMIT` and the index name (also given as a `$parameter`) are evaluated per input row, so they can read variables of earlier clauses.
- The binding variable must be a node variable of the `MATCH` pattern. `OPTIONAL MATCH ⋯ SEARCH` is refused. A binding variable already bound by an earlier clause is restricted to the `k` nearest, not rebound.
- The widening stops at 100,000 candidates per index, so an inner `WHERE` that keeps fewer than `k` of them returns fewer than `k` rows.

See CREATE VECTOR INDEX to create the index.

15.3.8. WHERE

The `WHERE` clause filters results based on conditions.

Syntax:

```
WHERE condition
```

Operators:

Type	Operators	Example
Comparison	`=`, `<>`, `<`, `>`, `⇐`, `>=`	`WHERE n.age >= 25`
Logical	`AND`, `OR`, `NOT`	`WHERE n.age > 25 AND n.city = 'NYC'`
Null check	`IS NULL`, `IS NOT NULL`	`WHERE n.email IS NOT NULL`
List	`IN`	`WHERE n.name IN ['Alice', 'Bob']`

Type	Operators	Example
String	`STARTS WITH`, `ENDS WITH`, `CONTAINS`	`WHERE n.name STARTS WITH 'A'`
Regex	`=~`	`WHERE n.email =~ '.*@gmail.com'`

Examples:

```
// Simple comparison
MATCH (p:Person)
WHERE p.age > 30
RETURN p.name

// Multiple conditions
MATCH (p:Person)
WHERE p.age >= 25 AND p.age <= 40 AND p.city = 'NYC'
RETURN p

// NULL checks
MATCH (p:Person)
WHERE p.email IS NOT NULL
RETURN p.name, p.email

// IN operator
MATCH (p:Person)
WHERE p.name IN ['Alice', 'Bob', 'Charlie']
RETURN p

// String matching
MATCH (p:Person)
WHERE p.name STARTS WITH 'A'
RETURN p

// Regular expression
MATCH (p:Person)
WHERE p.email =~ '.*@example.com'
RETURN p

// Parenthesized expressions
MATCH (p:Person)
WHERE (p.age < 25 OR p.age > 60) AND p.status = 'active'
RETURN p

// Pattern predicates
MATCH (p:Person)
WHERE (p)-[:KNOWS]->()
RETURN p.name AS hasConnections
```

15.3.9. RETURN

The `RETURN` clause specifies what to include in the query result.

Syntax:

```
RETURN expression [AS alias] [, expression [AS alias] ...]
```

Examples:

```
// Return nodes
MATCH (p:Person)
RETURN p

// Return properties
MATCH (p:Person)
RETURN p.name, p.age

// Return with aliases
MATCH (p:Person)
RETURN p.name AS personName, p.age AS personAge

// Return all: *
MATCH (a:Person)-[r:KNOWS]->(b:Person)
RETURN *

// Return expressions
MATCH (p:Person)
RETURN p.name, toUpper(p.city) AS upperCity

// Aggregations
MATCH (p:Person)
RETURN count(p) AS total, avg(p.age) AS avgAge

// RETURN DISTINCT - removes duplicate rows
MATCH (p:Person)
RETURN DISTINCT p.city AS city

// DISTINCT with multiple columns
UNWIND [1, 2, 2, 3, 3, 3] AS n
RETURN DISTINCT n

// Aggregations with DISTINCT
UNWIND [1, 1, 2, 3] AS n
RETURN count(DISTINCT n) AS uniqueCount, sum(DISTINCT n) AS uniqueSum

// Standalone expressions (no MATCH)
RETURN abs(-42), sqrt(16), range(1, 5)
```

In a read-only query with `RETURN DISTINCT`, the clauses after the last one that binds a returned variable (`MATCH`, `OPTIONAL MATCH`, `UNWIND`, a `WITH` that only passes variables on) can only keep or drop a row. ArcadeDB runs them as a filter: it stops at the first match for each row and skips rows whose returned values it already returned, instead of expanding every combination. `PROFILE` shows this as a `SEMI JOIN` step. The result is the same. A query keeps the full expansion when it aggregates, uses `RETURN *`, or returns a value that can change from call to call, such as `rand()`, `randomUUID()` or a user function. *(Since v26.11.1)*

A query has to conclude with a `RETURN`, a `FINISH`, an update clause (`CREATE`, `MERGE`, `SET`, `REMOVE`, `DELETE`, `FOREACH`), a unit `CALL { }` (one whose body returns nothing) or a procedure `CALL`. A query that ends with `MATCH`, `OPTIONAL MATCH`, `UNWIND`, `WITH`, `LOAD CSV` or a returning `CALL { }` is refused before it runs with `Query cannot conclude with …`, as Neo4j refuses it: write the `RETURN` that says what to answer (`MATCH (p:Person) RETURN p`, not `MATCH (p:Person)`). The body of a `CALL { }` concludes the same way, while the bodies of `EXISTS { }` and `COUNT { }` may end with a pattern. *(Since v26.12.1, issue #12354: such a query used to answer the variables in scope.)*

15.3.10. CREATE

The `CREATE` clause creates new nodes and relationships.

Syntax:

```
CREATE pattern [, pattern ...]
```

Examples:

```
// Create a node
CREATE (p:Person {name: 'Alice', age: 30})

// Create multiple nodes
CREATE (a:Person {name: 'Bob'}), (b:Person {name: 'Charlie'})

// Create a relationship between new nodes
CREATE (a:Person {name: 'Alice'})-[:KNOWS]->(b:Person {name: 'Bob'})

// Create a relationship with properties
CREATE (a:Person {name: 'Alice'})-[:WORKS_AT {since: 2020}]->(c:Company {name:
'ArcadeDB'})

// Create relationship between existing nodes
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2024}]->(b)

// Create chained paths
```

```
CREATE (a:Person {name: 'A'})-[:KNOWS]->(b:Person {name: 'B'})-[:KNOWS]->(c:Person
{name: 'C'})
```

A property value follows the same rule as `SET`: it must be a scalar, a list of scalars, or a `point()` value.

A relationship type name cannot contain `~`, since that character is reserved internally for multi-label vertex types.

A relationship endpoint bound by an earlier clause must be a node. A variable an `OPTIONAL MATCH` left null, or a value that is not a node, fails the statement with `Expected the value NULL to be of type NODE`, as on Neo4j *(since v26.11.1; before, `OPTIONAL MATCH (a:A {k: 99}) CREATE (a)-[:R]→(b)` created an anonymous node in place of `a`)*. A null property, on the other hand, is simply not set: `CREATE (n:B {k: null})` creates a node without `k`.

15.3.11. MERGE

The `MERGE` clause creates nodes or relationships if they don't exist, or matches them if they do (upsert operation).

Syntax:

```
[path_variable =] MERGE pattern
[ON CREATE SET ...]
[ON MATCH SET ...]
```

`ON CREATE SET` and `ON MATCH SET` accept every form the `SET` clause does.

A property of the `MERGE` pattern cannot be null, since nothing matches a null and the `MERGE` would create the element again on every run: `MERGE (n:A {k: null})`, `UNWIND [null] AS x MERGE (n:A {k: x})`, `MERGE (n:A {k: $p})` with `$p` null and `MATCH (a) MERGE (n:B {k: a.missing})` all fail with `'MERGE' cannot be used with a graph element property value that is null`, on nodes and relationships alike. A relationship endpoint bound to null, by an `OPTIONAL MATCH` that found nothing, fails the statement as in `CREATE`. Both match Neo4j *(since v26.11.1; before, only the literal `null` was refused, a null from a variable created a node without the property, and a null endpoint silently dropped the row)*.

Like `MATCH`, `MERGE` can name the path it matched or created. The path spans the whole pattern, so `length()`, `nodes()` and `relationships()` report what they would for the equivalent `MATCH`, whether or not the individual nodes and relationships were given names of their own:

```
// length(p) is 2, nodes(p) has 3 entries and relationships(p) has 2,
// even though only the first node is named
MERGE p = (a:Person {name: 'Alice'})-[:KNOWS]->(:Person {name: 'Bob'})-[:KNOWS]->
```

```
(:Person {name: 'Carol'})
RETURN length(p), nodes(p), relationships(p)
```

Examples:

```
// Find or create a node
MERGE (p:Person {name: 'Alice'})

// With ON CREATE action
MERGE (p:Person {name: 'Bob'})
ON CREATE SET p.created = true, p.timestamp = timestamp()

// With ON MATCH action
MERGE (p:Person {name: 'Alice'})
ON MATCH SET p.lastSeen = timestamp(), p.visits = p.visits + 1

// Both ON CREATE and ON MATCH
MERGE (p:Person {name: 'Charlie'})
ON CREATE SET p.created = true, p.count = 1
ON MATCH SET p.count = p.count + 1, p.updated = true

// Merge relationship between existing nodes
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MERGE (a)-[r:KNOWS]->(b)
ON CREATE SET r.since = 2024
ON MATCH SET r.lastContact = timestamp()

// Copy the defaults onto a node that was just created
MATCH (defaults:Settings {scope: 'person'})
MERGE (p:Person {name: 'Dave'})
ON CREATE SET p += defaults
```

A property value in the pattern follows the same rule as `SET`: it must be a scalar, a list of scalars, or a `point()` value.

(Since v26.10.2) `MERGE` finds exactly the records the identical `MATCH` pattern finds, with the same comparison, including for a declared property (see Comparing a property declared in the schema).

Concurrent `MERGE` of a relationship. *(Since v26.12.1)* When several transactions `MERGE` the same relationship between two existing nodes at the same time, exactly one creates it. A transaction whose `MERGE` created a relationship that another transaction committed after it began is refused at commit with a retryable `ConcurrentModificationException`; its retry (the server retries an auto-committed request by itself, a driver retries a managed transaction) matches the relationship the other one created. The relationship is identified by its two endpoints, its type and the properties the `MERGE` matched it by, so a `MERGE` matching by type alone (`MERGE (a)-[:KNOWS]→(b)`) also finds a relationship of that type created concurrently with properties. The same guarantee covers

`merge.relationship()` and SQL `CREATE EDGE … IF NOT EXISTS`. In v26.11.1 and earlier, concurrent writers could commit the same relationship more than once when its endpoints already had relationships.

The one shape this does not detect is a concurrent `MERGE` whose identifying property map is a strict superset of another one's (`{uuid: 1}` against `{uuid: 1, kind: 'a'}`). A unique constraint on the identifying property (`CREATE CONSTRAINT FOR ()-[r:KNOWS]-() REQUIRE r.uuid IS UNIQUE`) is exact in every case, and remains the declaration to use where a property identifies the relationship. Concurrent `MERGE` of a node still needs a unique constraint, as in other Cypher databases.

Bulk loads of relationships with several writers. *(Since v26.12.1)* When the relationship's type has an index (unique or not, but not full-text) on properties the `MERGE` pattern gives all of, with non-null values, a `MERGE` of a relationship between two existing nodes looks the relationship up in that index instead of reading the nodes' relationship lists. Writers that `MERGE` different relationships on shared nodes then no longer refuse each other at commit, which is what made a load like `UNWIND $rows AS r MATCH (a {uuid: r.src}), (b {uuid: r.dst}) MERGE (a)-[:RELATES_TO {uuid: r.uuid}]→(b)` slower with 8 writers than with one. With such an index, a concurrent `CREATE` of the same relationship (even one that sets the key with a later `SET`) is detected too, and the `MERGE` is refused and retried. Without an index, with only part of a composite index's properties in the pattern, or for a user who may not read the relationship type, the `MERGE` reads the relationship lists as before.

15.3.12. SET

The `SET` clause updates properties and labels on nodes and relationships.

Syntax:

```
SET variable.property = value [, ...]     // one property
SET variable[keyExpression] = value       // property name computed at runtime
SET variable = source                     // replace all properties
SET variable += source                    // add to the existing properties
SET variable:Label [, ...]                // add a label
```

For the `=` and `+=` forms, `source` is a map, a node or a relationship. Copying from a node or a relationship takes all of its properties, which is the usual way to clone one.

Examples:

```
// Update a single property
MATCH (p:Person {name: 'Alice'})
SET p.age = 31

// Update multiple properties
MATCH (p:Person {name: 'Alice'})
```

```
SET p.age = 31, p.city = 'Boston', p.updated = true

// Update relationship property
MATCH (a:Person)-[r:KNOWS]->(b:Person)
WHERE a.name = 'Alice' AND b.name = 'Bob'
SET r.strength = 10

// Replace every property with the ones in the map: 'age' and 'city' are dropped
MATCH (p:Person {name: 'Alice'})
SET p = {name: 'Alice', role: 'admin'}

// Add to the existing properties, keeping the ones not mentioned
MATCH (p:Person {name: 'Alice'})
SET p += {role: 'admin'}

// Copy every property of one node onto another
MATCH (source:Person {name: 'Alice'}), (target:Person {name: 'Alice2'})
SET target = source

// Add a label
MATCH (p:Person {name: 'Alice'})
SET p:Employee
```

`SET` is a simultaneous assignment: every value is read as it was before the clause ran, so `SET p.a = p.b, p.b = p.a` swaps the two rather than copying one onto the other.

A label change (`SET n:Label`, `REMOVE n:Label`, or `ON CREATE SET` / `ON MATCH SET` of a label in a `MERGE`) rewrites the node under its new type and re-creates its relationships, so the node and its relationships get new RIDs (see `opencypher.labelWriteDegreeWarning` in the settings). *(Since v26.12.1)* Every row the statement returns holds the records as they are after the change, including a row produced before a later row of the same statement changed a label of one of its nodes: its nodes, relationships and paths are the live ones, and a node another row relabelled reports its new labels. `DISTINCT` and `UNION` count a node before and after its label change as one. Up to 26.11.1 such a row still held the deleted originals: the HTTP API answered RIDs of records that no longer existed, and a path returned over Bolt failed with `Record #⋯ not found` (issue #10614).

A `SET` updates each matched node once, even when it writes the indexed property the `MATCH` reads its nodes by: `MATCH (n:A) WHERE n.k > 0 SET n.k = n.k + 1000` with an index on `A.k` reads the range of the index first, then updates. Only the record addresses are held, a few bytes per node. Before v26.11.1 such a statement met the nodes it had just updated again further on in the index and ran out of memory. *(Since v26.11.1)*

A property value must be a scalar, a list of scalars, or a `point()` value. A map value (`SET p.address = {city: 'Rome'}`), or a right-hand side of `=` / `+=` that is neither a map nor an entity (`SET p = 5`), is rejected with a `TypeError` and no part of the clause is written.

Concurrent updates. A `SET` (stand-alone, or in `MERGE ⋯ ON CREATE SET` / `ON MATCH SET`) computes the value it writes from the latest committed version of the record, even under the default

`READ_COMMITTED` isolation. So `SET n.count = n.count + 1` run by concurrent transactions never silently loses an increment: a change committed by another transaction after that point makes the commit fail with a retryable conflict. Two things follow:

- under contention the right-hand side can be evaluated more than once, so keep it free of side effects;
- a `MERGE ⋯ ON MATCH SET` that assigns a value the matched record already holds writes nothing, so read-mostly `MERGE` workloads do not conflict. It keeps a different value committed concurrently after the match.

15.3.13. DELETE

The `DELETE` clause removes nodes and relationships from the graph.

Syntax:

```
DELETE expression [, expression ...]
DETACH DELETE expression [, expression ...]
```

A target is usually a variable, but any expression that evaluates to a node, a relationship or a path works too: `DETACH DELETE [a][0]`, `DELETE l[i]`, `DETACH DELETE CASE WHEN a.k = 0 THEN a ELSE b END`. A target that evaluates to a list or a map deletes every element it holds, so `DETACH DELETE [a, b]` deletes both nodes (Neo4j raises a type error for a list target). A target that can never be an entity (`DELETE 1 + 1`, `DELETE n:Label`) is refused before the statement runs.

A node cannot be deleted if it has relationships. Use `DETACH DELETE` to delete a node and all its relationships.

The check runs once the whole statement is processed, so `MATCH (n)-[r]-() DELETE r, n` deletes a node that has several relationships. A node that still has a relationship at the end fails the statement and nothing is deleted.

(Since v26.12.1) These two failures are client errors, as in Neo4j, not server faults. A node that still has a relationship answers HTTP `409` and `Neo.ClientError.Schema.ConstraintValidationFailed` over Bolt. Reading, setting or linking a node deleted earlier in the same statement (`MATCH (n:A) DELETE n RETURN n.x`) answers HTTP `404` and `Neo.ClientError.Statement.EntityNotFound`. Before, both answered `500` and `Neo.DatabaseError.General.UnknownError`.

(Since v26.12.1) The same error is raised when the deleted node or relationship is reached through another variable, a list, a map or a path the row still holds, for example `MATCH (n:A) WITH n, collect(n) AS ns DELETE n UNWIND ns AS m RETURN m.x`, a relationship removed by `DETACH DELETE` and held in `collect(r)` or in a path, or a record deleted inside a `FOREACH` body. A property read, `keys()`, `properties()`, `labels()` or a map projection of it fails, while `size()` of the list and `type()` of a

deleted relationship keep working, and the elements that were not deleted are read as usual. Up to 26.11.1 such a read answered the values the entity had before the delete (issue #12548).

(Since v26.12.1) A `DETACH DELETE` that follows a relationship pattern (`MATCH (a)-[r]→(b) DETACH DELETE a, b`, directly, behind a `WITH` that neither aggregates nor sorts, or inside a `FOREACH` or `CALL { ⋯ }` body) reads every matched row into memory before the first deletion, so removing the relationships of `a` cannot cut short the walk of a□s relationships. The rows count against the query memory limits (`queryMaxHeapElementsAllowedPerOp` and `queryMaxHeapRAM`): past them the statement fails and nothing is deleted. A plain `DELETE`, a `DETACH DELETE` of the matched relationship variables only, and a `DETACH DELETE` after a node-only `MATCH` are not read first by this rule (a delete of a relationship can be, see the next note). Up to 26.11.1 such a statement could silently process only part of the rows once `a` had more than about 100 relationships, so it deleted, set or returned only some of the matched nodes (issue #12254).

(Since v26.12.1) A `DELETE r` or `DETACH DELETE r` of a relationship also reads every matched row into memory before the first deletion when a later expansion could reach the same relationship again: the part of the query since the last `WITH` holds more than one relationship pattern (two `MATCH` clauses, two patterns of one `MATCH`, or a path of two hops), a relationship without a direction (`-[r]-`), or rows that come ahead of the pattern from an `UNWIND`, `LOAD CSV`, `CALL`, `CALL { ⋯ }` or an earlier `WITH` (unless that `WITH` aggregates everything into one row). The same holds inside a `FOREACH` or `CALL { ⋯ }` body that deletes. The rows count against `queryMaxHeapElementsAllowedPerOp` and `queryMaxHeapRAM`, as above. A bulk delete over one relationship pattern with a direction, fed by the `MATCH` alone (`MATCH ()-[r:T]→() DELETE r`), reaches each relationship once and keeps streaming. Up to 26.11.1 the first rows deleted the relationship before the later rows expanded again, so `MATCH (a:V {k: 0}) UNWIND range(1, 500) AS i MATCH (a)-[r:F]→(b) DELETE r RETURN i` answered 100 rows instead of 500, and every clause after the `DELETE` ran for those rows only (issue #12521).

This includes the incoming edges of a unidirectional edge type, which the node itself does not store: `DELETE` refuses a node they still point to, and `DETACH DELETE` removes them. Deleting a vertex through the Java API or SQL `DELETE VERTEX` removes them too. The first vertex deleted in a transaction scans the unidirectional edge types to find them, so on a very large unidirectional type prefer deleting many vertices in one transaction.

(Since v26.12.1) Deleting a relationship that the statement already removed is a no-op, as in Neo4j. In `MATCH (a)-[r]→(b) DETACH DELETE a DELETE r`, the `DETACH DELETE` removes `r` together with `a`, and the `DELETE r` that follows does nothing more: `relationshipsDeleted` counts the relationship once, and `count()` on its type stays right. Up to 26.11.1 the relationship was deleted twice, so `relationshipsDeleted` counted it twice and `count()` on the type under-reported,

> also after a restart, until `CHECK DATABASE FIX` recomputed it (issue #12253).

Examples:

```
// Delete a relationship
MATCH (a:Person)-[r:KNOWS]->(b:Person)
WHERE a.name = 'Alice' AND b.name = 'Bob'
DELETE r

// Delete a node (must have no relationships)
MATCH (p:Person {name: 'TestNode'})
DELETE p

// Delete node with all its relationships
MATCH (p:Person {name: 'Alice'})
DETACH DELETE p

// Delete multiple elements
MATCH (a:Person)-[r:TEMP]->(b:Person)
DELETE a, r, b
```

15.3.14. Reads and Writes in the Same Query

A query never sees the vertices and edges it creates itself. Everything the pattern matched before the query wrote anything is what it returns - the reading clauses are evaluated against the graph as it was, not against the rows the writes behind them add.

This matters whenever a write can produce something an earlier pattern would match:

```
// [r] has no type, so it would match the SCORED edges this query creates.
// It does not: the row count is the number of edges that existed beforehand.
MATCH (a:Person)-[r]->(b:Person)
MERGE (a)-[:SCORED]->(b)
RETURN count(*)
```

To make that hold, ArcadeDB reads the whole result of the pattern into memory before applying the first write. You can see where this happens in `EXPLAIN`, as an `EAGER (read/write barrier)` step. Two things follow from it, and they only apply to the queries that actually get one:

- the matched rows are held in memory until the last write is applied, so a query over a very large match needs the memory for it;
- a `LIMIT` no longer stops the scan early - the write runs for every matched row.

Queries whose writes cannot feed a preceding pattern are left alone and keep streaming, which covers the usual bulk shapes:

```
// Nothing is read, so nothing can be re-read.
UNWIND range(1, 1000000) AS i CREATE (:LogEntry {seq: i})

// KNOWS is read, SCORED is written: no overlap.
MATCH (a:Person)-[:KNOWS]->(b:Person) CREATE (a)-[:SCORED]->(b)
```

A call to a procedure that writes (`merge.node`, `merge.relationship`, the `refactor.*` procedures) always counts as a conflicting write, because what it creates is only known once it runs.

A pattern is read wherever it appears, not only in a `MATCH`. *(Since v26.12.1)* A `MATCH` inside a `CALL { … }` body, an `EXISTS { … }`, `COUNT { … }` or `COLLECT { … }` subquery, a pattern predicate and a pattern comprehension are weighed against a later `CREATE` or `MERGE` too:

```
// With no Item nodes yet: 150 rows, because every row counts the Item nodes as they
were before the MERGE.
UNWIND range(1, 150) AS i
WITH i WHERE COUNT { MATCH (:Item) } = 0
MERGE (:Item {id: i})
RETURN count(*)
```

Up to 26.11.1 such a query could stop after the first 100 rows, because the rows after them already saw the nodes the first rows created (issue #11463).

A `MERGE` is a read too: it looks its pattern up once per row. A `SET` or `REMOVE` after it (directly, behind a `WITH`, or inside a `FOREACH`) that writes a property key or a label the `MERGE` pattern matches on therefore runs only after the `MERGE` has run for every row, exactly as if the two clauses ran one after the other:

```
// One node: both rows MERGE first (row 2 finds the node row 1 created), then both
rows remove key.
UNWIND [1, 2] AS x
MERGE (n:Item {id: 1, key: 'value'})
SET n.key = null
```

A `SET n = map` or `SET n += map` whose keys are only known when the query runs (`SET n += row`, `SET n = $props`) counts as writing every key, so `UNWIND $rows AS row MERGE (n:Person {id: row.id}) SET n += row` gets the barrier too. Writes of other keys keep streaming: `MERGE (n:Person {id: row.id}) SET n.name = row.name` has no barrier, nor does a `MERGE` that starts the query, which runs its lookup only once.

Up to 26.10.2 the `SET` ran right after the `MERGE` of each row, so the `MERGE` of a later row already saw it: the example above created two nodes, and adding a `WITH` between the clauses changed the result depending on how many rows were pulled at a time. Since 26.11.1 the answer is the same in every spelling, and the same as Neo4j's. A bulk load that used `SET n += row` after a `MERGE` now holds its rows in memory until the `MERGE` has run for all of them: to keep it streaming, set the

properties by name (`SET n.name = row.name, n.age = row.age`).

A value the query reads before a write keeps the value it had when it was read. When a `WITH`, an `UNWIND` or the value of a `SET` reads a property that a later `SET`, `REMOVE`, `MERGE` action, `FOREACH`, writing subquery or writing procedure changes, every row is read before the first write runs:

```
// One Counter node with k = 1: all 250 rows return 1, and k ends at 251.
MATCH (n:Counter)
UNWIND range(1, 250) AS i
WITH n, n.k AS before
SET n.k = n.k + 1
RETURN before
```

Reading and writing the same property through a variable that holds a different node on every row keeps streaming, because no row can see another row's write: `MATCH (n:P) WITH n, n.k AS old SET n.k = old + 1` has no barrier. Values read from `LOAD CSV` rows or from a literal list are not graph properties and never cause one.

Up to 26.10.2 the rows were projected in batches of 100, so the rows after the first 100 could see the write and return the new value; adding a `WITH *` changed the result (issue #11005).

A `MATCH` filter also sees the graph as it was before a later write of the property it filters on. *(Since v26.11.1)* A `SET`, `REMOVE`, `SET +=` with a map literal, `MERGE ⋯ ON CREATE` or `ON MATCH SET`, `FOREACH` or writing subquery that names the property read by an earlier `MATCH` pattern or `WHERE` plants an eager barrier before the write, so the query holds its input in memory. A query that read its own later write now returns the pre-write answer, and `UNWIND ⋯ MATCH (n {id: row.id}) SET n.id = ⋯` costs memory. An opaque map write that can rewrite a lookup key plants the barrier too: `MATCH (n {id: row.id}) SET n += <anything but row>` over several input rows used to answer duplicate lookups with the rewritten key (150 duplicate lookups answered 100 rows), and now holds the rows in memory, as Neo4j does. The exception is a map taken from the same variable the key is read from (an inline `{id: row.id}`, `WHERE n.id = row.id`, or a key projected by `WITH row, row.id AS id`), which writes back the value the node was found by, so `LOAD CSV` and `UNWIND` upserts keep streaming. A `WITH` rename drops that pairing, and a subquery body with an opaque write is weighed as well.

15.3.15. WITH

The `WITH` clause allows query chaining by passing results from one part of a query to another. It can project, filter, aggregate, and paginate intermediate results.

Syntax:

```
WITH expression [AS alias] [, expression [AS alias] ...]
[WHERE condition]
[ORDER BY expression [ASC|DESC]]
[SKIP n]
```

```
[LIMIT n]
```

Examples:

```
// Basic projection
MATCH (p:Person)
WITH p.name AS name, p.age AS age
RETURN name, age ORDER BY name

// Filter after projection
MATCH (p:Person)
WITH p.name AS name, p.age AS age
WHERE age > 30
RETURN name

// Aggregation with WITH
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH m.title AS movie, count(p) AS actorCount
WHERE actorCount > 2
RETURN movie, actorCount

// WITH DISTINCT
MATCH (p:Person)
WITH DISTINCT p.city AS city
RETURN city ORDER BY city

// Pagination with WITH
MATCH (p:Person)
WITH p.name AS name
ORDER BY name
SKIP 5
LIMIT 10
RETURN name

// Multiple WITH clauses
MATCH (p:Person)
WITH p.name AS name, p.age AS age
WHERE age > 25
WITH name, age
WHERE age < 50
RETURN name ORDER BY name

// WITH * (pass all variables)
MATCH (p:Person)
WITH *
WHERE p.age > 30
RETURN p.name
```

15.3.16. LET

`LET` (ISO GQL) adds new variables to every row without grouping or dropping the others. It is the same as `WITH *, expression AS variable`. Since v26.10.2.

```
MATCH (p:Person)
LET age = 2026 - p.birthYear, adult = age >= 18
RETURN p.name, age, adult
```

- A later item can read an earlier one (`adult` reads `age` above).
- `LET` declares a new variable: naming a variable already in scope is refused (`Variable 'p' already declared`). To replace a value, use `WITH`.
- `LET` cannot aggregate: use `WITH` for `count()`, `collect()` and the other aggregations.

15.3.17. UNWIND

The `UNWIND` clause expands a list into individual rows.

Syntax:

```
UNWIND list AS variable
```

Examples:

```
// Unwind a literal list
UNWIND [1, 2, 3] AS x
RETURN x

// Unwind with range
UNWIND range(1, 10) AS num
RETURN num

// Unwind property array
MATCH (p:Person {name: 'Alice'})
UNWIND p.hobbies AS hobby
RETURN p.name, hobby

// Create from list
UNWIND ['Alice', 'Bob', 'Charlie'] AS name
CREATE (p:Person {name: name})

// Nested unwind
UNWIND [[1, 2], [3, 4]] AS inner
UNWIND inner AS num
RETURN num

// Combine with MATCH
```

```
MATCH (p:Person)
UNWIND p.skills AS skill
RETURN p.name, skill
```

15.3.18. ORDER BY

The `ORDER BY` clause sorts the results.

Syntax:

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC] ...]
```

Examples:

```
// Ascending order (default)
MATCH (p:Person)
RETURN p.name
ORDER BY p.name

// Descending order
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC

// Multiple sort keys
MATCH (p:Person)
RETURN p.name, p.age, p.city
ORDER BY p.city ASC, p.age DESC, p.name ASC
```

Strings are sorted by Unicode code point, the same order an index keeps them in (and the order Neo4j uses). The same order applies to string comparisons (`<`, `>`, ...) and to `min()`/`max()`.

`BINARY` values are sorted by their bytes, compared unsigned and one by one, a shorter value before a longer one it is the start of: the order SQL's `ORDER BY` and an index give them. Before v26.11.1 the order of two binary values changed from one run of the query to the next. *(Since v26.11.1)*

The arrays of numbers (`ARRAY_OF_SHORTS`, `ARRAY_OF_INTEGERS`, `ARRAY_OF_LONGS`, `ARRAY_OF_FLOATS`, `ARRAY_OF_DOUBLES`) are sorted element by element, as a list is. A `BINARY` or array value is one value wherever Cypher compares values: two records holding the same bytes or elements are equal under `=`, one row under `DISTINCT`, `UNION` and `collect(DISTINCT ⋯)`, one group under an aggregation, and `min()` / `max()` (and `coll.min()` / `coll.max()`) order them like `ORDER BY` does. A `BINARY` value is not a list: it compares with another `BINARY` value by its unsigned bytes, and is never equal to a list. A `FLOAT` is keyed by its decimal form, as `=` reads it, so `0.1` stored as a `FLOAT` and as a `DOUBLE` are one group. Before v26.11.1 (issue #10380) they were distinct and `min()` / `max()` answered the first row met. *(Since v26.11.1)*

After a `DISTINCT` or aggregating `RETURN`/`WITH`, `ORDER BY` can still use any expression built only from the grouping keys, also through the variable a key was projected from. The expression is read from the projected column:

```
MATCH (n)
RETURN n AS node, count(*) AS c
ORDER BY n:Person DESC, n.name   // sorts on node:Person, node.name
```

This covers label checks (static and `$(…)`), `IN`, `IS NULL`, `STARTS WITH`, `=~`, `CASE`, list and map expressions, map projections, list comprehensions, `any()`/`all()` and `reduce()`. Before v26.12.1 most of these failed with `UndefinedVariable`. *(Since v26.12.1)*

15.3.19. SKIP and LIMIT

`SKIP` skips a number of results, and `LIMIT` restricts the maximum number of results returned. Used together for pagination.

Syntax:

```
SKIP number
LIMIT number
```

Examples:

```
// Limit results
MATCH (p:Person)
RETURN p.name
LIMIT 10

// Skip results
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 20

// Pagination (page 3, 10 items per page)
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 20
LIMIT 10
```

LIMIT after a write: a `LIMIT` that follows a write clause (`CREATE`, `MERGE`, `SET`, `REMOVE`, `DELETE`, `FOREACH`, a write procedure or a subquery that writes) never cuts the write short. The write runs for every matched row first and only then does `LIMIT` trim the rows returned, as in Neo4j. `MATCH (n:Person)`

`SET n.seen = true RETURN n LIMIT 10` therefore updates every `Person` and returns 10 of them. On a large input this means the first row comes back only after all the writes are done.

Keyset pagination with an index: when the `ORDER BY` sorts by the indexed property the `MATCH` reads its range from, and the query has a `LIMIT`, the rows are read directly from the index in the requested order: no sort is needed and the scan stops as soon as the page is full, however many nodes match the range. This makes keyset (cursor) pagination fast on large types:

```
// Next page: pass the last id of the previous page as $cursor
MATCH (p:Person) WHERE p.id > $cursor
RETURN p.id, p.name ORDER BY p.id LIMIT 10

// Previous page: the index is read backwards
MATCH (p:Person) WHERE p.id < $cursor
RETURN p.id, p.name ORDER BY p.id DESC LIMIT 10

// First page, with no WHERE: nodes without an id still come last
MATCH (p:Person)
RETURN p.id, p.name ORDER BY p.id LIMIT 10
```

It applies when the query only matches and returns (no `WITH`, `UNWIND`, aggregation or write clause in between), and also works with `SKIP`, `DISTINCT` and relationship expansions from the ordered node. `EXPLAIN` and `PROFILE` show `[index order]` on the `NodeIndexRangeScan` when it is used.

SKIP over an index-ordered scan: since v26.12.1 the index scan passes over the skipped rows from the index entries, without loading their nodes, when the query is a single `MATCH` and a `RETURN` with no `DISTINCT` or aggregation, has no `WHERE` left to evaluate on the rows, and every returned item is the matched variable, one of its properties or a literal. `MATCH (p:Person) RETURN p.id, p.name ORDER BY p.id SKIP 50000 LIMIT 10` therefore no longer loads 50,000 nodes to throw them away. The rows returned are the same. A range condition in `WHERE` (`p.id >= 0`) and a type with a read trigger or read event listener still load every skipped node. `EXPLAIN` and `PROFILE` show `SKIP n (passed over in the index scan)` when it is used. Keyset pagination stays the faster way to read deep pages.

15.3.20. EXPLAIN

The `EXPLAIN` clause shows the query execution plan without actually executing the query. This is useful for understanding how the query optimizer processes your query and for performance tuning.

Syntax:

```
EXPLAIN query
```

Examples:

```
// Show execution plan for a simple query
EXPLAIN MATCH (p:Person) RETURN p

// Show plan for a relationship traversal
EXPLAIN MATCH (a:Person)-[:KNOWS]->(b:Person) WHERE a.name = 'Alice' RETURN b

// Show plan for a query with aggregation
EXPLAIN MATCH (p:Person) RETURN p.city, count(p)
```

Output:

The EXPLAIN output includes:

- Whether the cost-based optimizer is being used
- Physical operators in the execution plan (for optimized queries)
- Estimated costs and row counts
- Index usage information
- An `EAGER (read/write barrier)` step where the query reads its match into memory before writing - see Reads and Writes in the Same Query

```
OpenCypher Native Execution Plan
================================

Using Cost-Based Query Optimizer

Physical Plan:
NodeIndexSeek (Person.name = 'Alice')
   ├──── Estimated Cost: 10.00
   └──── Estimated Rows: 1

Total Estimated Cost: 10.00
Total Estimated Rows: 1
```

For queries that don't use the optimizer, EXPLAIN shows:

```
OpenCypher Native Execution Plan
================================

Using Traditional Execution (Non-Optimized)

Reason: Query pattern not yet supported by optimizer
Execution will use step-by-step interpretation
```

Composite indexes. A node pattern that binds only the leading properties of a composite index, `(n:T {a: $x})` with an index on `T(a, b)`, reads the index entries starting with that value on both

paths, and so does MERGE *(since v26.12.1, issue #11576)*. Before, the non-optimized path (a MATCH after a WITH, OPTIONAL MATCH, UNWIND ⋯ MATCH, MERGE) scanned the whole type unless every property of the index was bound. PROFILE shows it as [index: T[a, b], prefix: a]. A composite index with a column indexed BY ITEM, BY KEY or BY VALUE is not read this way while that column is left unbound: a record with an empty collection has no entry there, so the statement scans the type.

When several indexes lead with the same property, the cost-based optimizer reads the one whose key the pattern binds furthest *(since v26.12.1, issue #12557)*. With indexes on T(tenantId, id) and T(tenantId, name), (n:T {tenantId: $t, id: $v}) seeks T(tenantId, id) on both properties, instead of reading every tenantId entry and filtering id. A UNIQUE index whose whole key is bound comes first, then the index with the most bound properties, then an exact lookup over a prefix range; the index name settles a tie, so the same schema always plans the same way. A label alternative such as (n:A|B {⋯}) seeks the best index of each type.

15.3.21. PROFILE

The PROFILE clause executes the query and returns performance metrics along with the actual results. Unlike EXPLAIN which only shows the plan, PROFILE runs the query and provides real execution statistics.

Because it runs the query, a PROFILE of a write (CREATE, MERGE, SET, DELETE, ...) is refused on the read-only path (query(), /api/v1/query) exactly like the bare write; profile it through command() or POST /api/v1/command. EXPLAIN never runs the query and is accepted on either path. *(Since v26.10.2)*

A query that fails fails under PROFILE too, with the same error, and a write that fails is rolled back exactly as it is without PROFILE. *(Since v26.11.1)*

Before v26.11.1 a failing query under PROFILE answered no rows and no error: the error only appeared as an Error: line of the profile text, and the writes the query made before failing were committed. A client that relied on that line to detect a failure now receives the error itself.

Syntax:

```
PROFILE query
```

Examples:

```
// Profile a simple query
PROFILE MATCH (p:Person) RETURN p.name

// Profile a relationship traversal
PROFILE MATCH (a:Person)-[:KNOWS]->(b:Person) WHERE a.name = 'Alice' RETURN b.name

// Profile a query with aggregation
```

```
PROFILE MATCH (p:Person) RETURN p.city, count(p) AS count
```

Output:

The PROFILE output includes:

- Execution time in milliseconds
- Number of rows returned
- Query execution plan details (when using optimizer)

```
OpenCypher Query Profile
========================

Execution Time: 15 ms
Rows Returned: 42

Physical Plan:
NodeTypeScan (Person)
   └──── Estimated Rows: 50

Actual Results Follow...
```

The profile information is returned as the first row in the result set, followed by the actual query results.

15.3.22. UNION and UNION ALL

The `UNION` clause combines the results of two or more queries into a single result set. `UNION` removes duplicate rows, while `UNION ALL` keeps all rows including duplicates.

Syntax:

```
query1
UNION [ALL]
query2
[UNION [ALL]
query3 ...]
```

All queries in a UNION must return the same column names.

Examples:

```
// UNION - removes duplicate results
MATCH (p:Person) RETURN p.name AS name
```

```
UNION
MATCH (c:Company) RETURN c.name AS name

// UNION ALL - keeps all results including duplicates
MATCH (p:Person) RETURN p.name AS name
UNION ALL
MATCH (c:Company) RETURN c.name AS name

// Multiple UNIONNs
MATCH (p:Person) RETURN p.name AS name
UNION
MATCH (c:Company) RETURN c.name AS name
UNION
MATCH (l:Location) RETURN l.name AS name

// UNION with WHERE clauses
MATCH (p:Person) WHERE p.age > 30 RETURN p.name AS name, 'person' AS type
UNION ALL
MATCH (c:Company) WHERE c.employees > 100 RETURN c.name AS name, 'company' AS type
```

Behavior:

- `UNION` removes duplicate rows from the combined result (like `SELECT DISTINCT` in SQL)
- `UNION ALL` preserves all rows, including duplicates (more efficient when duplicates are acceptable)
- Results are returned in the order they are produced by each query

---

15.3.23. Conditional query: WHEN ... THEN ... ELSE

A conditional query (Cypher 25) runs the first branch whose condition is true, the `ELSE` branch when none is, and returns no rows when none is true and there is no `ELSE`. Since v26.10.2.

```
WHEN $mode = 'all' THEN MATCH (p:Person) RETURN p.name AS name
WHEN $mode = 'adults' THEN MATCH (p:Person) WHERE p.age >= 18 RETURN p.name AS name
ELSE RETURN null AS name
```

- A conditional query is a whole statement: it cannot be nested in a subquery.
- Conditions read parameters only, not variables. A null condition is not taken; a condition that is not a boolean is an error.
- Every branch must return the same columns in the same order.
- Only the branch taken runs, so the writes of the other branches never happen. A query with any writing branch is still a write statement: run it as a command, not as a read-only query.
- Every parameter the query names must be bound, also the parameters of branches that are not taken.

15.3.24. CALL

The `CALL` clause invokes procedures and functions. This includes built-in database procedures as well as custom functions defined via SQL's `DEFINE FUNCTION`.

Syntax:

```
[OPTIONAL] CALL procedureName([arguments])
[YIELD field [AS alias] [, field [AS alias] ...]]
[WHERE condition]
```

Built-in Procedures:

Procedure	Description
`db.labels()`	Returns all vertex type names (labels)
`db.relationshipTypes()`	Returns all edge type names
`db.propertyKeys()`	Returns all property keys defined in the schema
`db.schema()`	Returns schema visualization data
`db.schema.visualization()`	Returns schema visualization data (alias)

These procedures only return the types the current user is allowed to read, like `SELECT FROM schema:types` in SQL.

The procedure arguments and the `WHERE` after `YIELD` can read every variable bound before the `CALL`, as in Neo4j: `MATCH (a)-[r:KNOWS]→(b) CALL my.proc(r) YIELD value WHERE value.weight > r.weight RETURN value`. *(Since v26.11.1: before, the `WHERE` saw only the yielded fields, and a relationship used only in the `CALL` was `null` there.)*

A name that is neither a procedure nor a function runs the SQL function of that full name (`CALL abs(-5)`, `CALL vector.neighbors(⋯)`), or the one after an explicit `sql.` prefix (`CALL sql.abs(-5)`, `CALL sql.geo.asText(⋯)`). Any other name fails as an unknown procedure (`Neo.ClientError.Procedure.ProcedureNotFound` over Bolt). *(Since v26.12.1: before, a namespaced name nothing answered ran the SQL function named by its last segment, so `CALL apoc.atomic.concat(n, 'name', 'zz')` ran SQL `concat()`, returned no row and wrote nothing, and `CALL nosuch.max(1, 2)` returned 2.)*

Examples:

```
// List all vertex types (labels)
CALL db.labels()

// List all relationship types
CALL db.relationshipTypes()

// List all property keys
CALL db.propertyKeys()
```

```
// Get schema information
CALL db.schema()
```

Vector Search Procedures:

ArcadeDB provides built-in procedures for performing approximate nearest neighbor (ANN) vector search using HNSW indexes. These are accessible from Cypher via `CALL` and use the same high-performance index as SQL's `vector.neighbors()`.

Procedure	Description
`db.index.vector.queryNodes(indexSpec, k, vector [, options])`	Neo4j-compatible vector search. Returns `node` (the document) and `score` (in [0, 1], higher is better, as in Neo4j: `(1 + cosine) / 2` for COSINE and DOT_PRODUCT, `1 / (1 + d^2)` for EUCLIDEAN). On a `PRODUCT` index it walks on the PQ codes and re-ranks `rerankOversample` (index setting, default 10) candidates per neighbor on the stored vectors since 26.11.1, so the score is exact. The optional `options` map, an ArcadeDB extension since 26.11.1, takes `where` and `filter` as `vector.neighbors()` does
`vector.neighbors(indexSpec, vector, k [, options])`	ArcadeDB-native vector search. Returns document properties and `distance` (lower is better). It takes the options of the SQL vector.neighbors(), among them `where` (since 26.11.1) and `filter`, which restrict the candidates during the search

The `indexSpec` parameter uses ArcadeDB's `'Type[property]'` format (e.g., `'Document[embedding]'`).

A `WHERE` after the `YIELD` runs on the `k` rows the search already returned. When most of the nearest neighbors fail the condition (the rows of another tenant, for example), fewer than `k` rows come back, or none. To filter during the search, pass the `where` or `filter` option: as the fourth argument of `db.index.vector.queryNodes()`, or in the options map of `vector.neighbors()` (see the examples below). The search then returns `k` matching rows whenever the index holds that many. Before 26.11.1, `db.index.vector.queryNodes()` takes no options and `vector.neighbors()` has `filter` but not `where`.

Examples:

```
// Neo4j-compatible syntax — returns node and score (similarity)
CALL db.index.vector.queryNodes('Document[embedding]', 10, $queryVector)
YIELD node, score
RETURN node.title AS title, score
ORDER BY score DESC
```

```
// ArcadeDB-native syntax — returns properties and distance
CALL vector.neighbors('Document[embedding]', $queryVector, 10)
YIELD name, distance
RETURN name, distance
ORDER BY distance

// Filter during the search: the 10 nearest rows of one tenant, however many rows the
other tenants hold
CALL db.index.vector.queryNodes('Document[embedding]', 10, $queryVector, { where: {
tenantId: 'acme' } })
YIELD node, score
RETURN node.title AS title, score
ORDER BY score DESC

// The same filter with the ArcadeDB-native procedure
CALL vector.neighbors('Document[embedding]', $queryVector, 10, { where: { tenantId:
'acme' } })
YIELD name, distance
RETURN name, distance
ORDER BY distance

// With parameters for k and vector
CALL db.index.vector.queryNodes('Document[embedding]', $k, $vec)
YIELD node, score
RETURN node.title AS title, score

// Search specific child type in an inheritance hierarchy
CALL db.index.vector.queryNodes('EMBEDDING_IMAGE[vector]', 5, $vec)
YIELD node, score
RETURN node.name, score
```

These procedures use the HNSW vector index for fast approximate search (O(log n)). Do **not** use `vector.cosinesimilarity()` inside a `MATCH` loop — that performs a brute-force O(n) scan over all documents. Always use `CALL` with a vector index for production workloads.

Full-Text Search Procedures:

ArcadeDB provides built-in procedures for searching a `FULL_TEXT` index (BM25-ranked) directly from Cypher, so text search and graph pattern matching can be combined in one statement. These are Neo4j-compatible equivalents of SQL's `SEARCH_INDEX()`; see the "Full-Text Index" how-to guide for how to create and configure one.

Procedure	Description
`db.index.fulltext.queryNodes(indexName, query [, options])`	Searches a full-text index on a vertex type. Returns `node` and `score` (BM25 relevance, higher is better)
`db.index.fulltext.queryRelationships(indexName, query [, options])`	Searches a full-text index on an edge type. Returns `relationship` and `score`

The `indexName` parameter uses ArcadeDB's `'Type[property]'` format (e.g., `'Article[content]'`), and `query` accepts Lucene query syntax.

Rows come back ordered by descending score, with ties broken by record id. The optional `options` map pages over that order with two keys, both non-negative integers: `skip` (rows to drop from the top, default `0`) and `limit` (rows to return, default unlimited). Neo4j also accepts an `analyzer` key; ArcadeDB refuses it by name, because the analyzer is fixed by the index metadata when the index is created and cannot be overridden per query. Any other key is refused too.

Examples:

```
// Search and rank by relevance
CALL db.index.fulltext.queryNodes('Article[content]', 'java AND database')
YIELD node, score
RETURN node.title AS title, score
ORDER BY score DESC

// Second page of ten results
CALL db.index.fulltext.queryNodes('Article[content]', 'java AND database', {skip: 10,
limit: 10})
YIELD node, score
RETURN node.title AS title, score

// Search relationships and continue graph pattern matching
CALL db.index.fulltext.queryRelationships('Cites[note]', 'java')
YIELD relationship, score
MATCH (a:Article)-[c:Cites]->(b:Article) WHERE c = relationship
RETURN a.title AS from, b.title AS to, score
ORDER BY score DESC
```

A `MATCH` may follow `CALL … YIELD` directly, as above: the rows the procedure yields feed straight into the pattern, and an intervening `WITH` is not required.

Calling User Functions:

User functions defined via SQL's `DEFINE FUNCTION` statement can be called using the `CALL` clause with their namespace. Functions can be defined in SQL, JavaScript, OpenCypher, or Java and are callable from any query language.

```
// Define user functions in different languages
DEFINE FUNCTION math.add "SELECT :a + :b AS result" PARAMETERS [a,b] LANGUAGE sql
DEFINE FUNCTION js.multiply "return x * y" PARAMETERS [x,y] LANGUAGE js
DEFINE FUNCTION cypher.double "RETURN $n * 2" PARAMETERS [n] LANGUAGE opencypher
```

```
// Call user functions from Cypher using CALL clause
CALL math.add(3, 5)
// Returns: {value: 8}
```

```
CALL js.multiply(4, 2)
// Returns: {value: 8}

CALL cypher.double(5)
// Returns: {value: 10}

// Or call directly in expressions
RETURN math.add(3, 5) AS sum,
       js.multiply(4, 2) AS product,
       cypher.double(5) AS doubled
// Returns: {sum: 8, product: 8, doubled: 10}
```

See User Functions for complete documentation on defining and using functions.

Using YIELD:

The `YIELD` clause filters and renames the columns returned by a procedure:

```
// Get only the label column
CALL db.labels() YIELD label

// Rename the output column
CALL db.labels() YIELD label AS vertexType

// Filter results with WHERE
CALL db.labels() YIELD label
WHERE label STARTS WITH 'P'
RETURN label

// Yield all columns
CALL db.propertyKeys() YIELD *
```

When calling a custom `DEFINE FUNCTION` (SQL, JavaScript, OpenCypher, or Java), the result is wrapped in a `value` property, regardless of the language it was defined in. Built-in procedures like `db.labels()` return results with specific property names like `label`, `relationshipType`, or `propertyKey`.

`YIELD *` on a call that has no declared output columns, such as a SQL function called as a procedure (`CALL vector.neighbors(…) YIELD *`) or a `DEFINE FUNCTION`, binds the columns of the rows the call returns, so `RETURN *` returns them and a column can be named after it (`YIELD * RETURN name, distance`). Those columns are known only when the call runs, so a name that no row carries reads as `null` instead of being refused when the query is parsed. Before 26.12.1 such a `YIELD *` bound nothing: `RETURN *` was refused with `NoVariablesInScope` and a named column with `UndefinedVariable`.

OPTIONAL CALL:

`OPTIONAL CALL` is to `CALL` what `OPTIONAL MATCH` is to `MATCH`: when the procedure yields no row for an

incoming row - none at all, or none that passes the `YIELD … WHERE` filter - the incoming row is kept and every yielded name is bound to `null`, instead of the row being dropped.

```
// Isolated nodes are returned too, with reached = null
MATCH (s:Person)
OPTIONAL CALL algo.bfs(s) YIELD node
RETURN s.name, node.name AS reached
```

Before v26.10.2 an `OPTIONAL CALL` whose procedure ran but yielded no rows dropped the incoming row, like a plain `CALL`; only a procedure that failed produced the `null` row. Queries such as `MATCH (s) OPTIONAL CALL algo.bfs(s) YIELD node RETURN count(*)` now also count the nodes that reach nothing.

A procedure that does not exist fails the query also under `OPTIONAL CALL`, as in Neo4j: `OPTIONAL` covers a procedure that yields nothing, not a misspelled name. Before v26.12.1 `OPTIONAL CALL db.nosuch()` returned a `null` row.

Subqueries (CALL { … }):

Cypher supports subqueries using the `CALL { … }` syntax. Subqueries are enclosed in curly braces and can import variables from the outer scope using `WITH`.

```
// Basic subquery with imported variable
UNWIND [1, 2, 3] AS x
CALL {
  WITH x
  RETURN x * 10 AS y
}
RETURN x, y

// Subquery with MATCH
MATCH (n:Item)
CALL {
  WITH n
  RETURN n.value * 2 AS doubled
}
RETURN n.name, doubled

// Multiple imported variables
UNWIND [1, 2] AS a
UNWIND [10, 20] AS b
CALL {
  WITH a, b
  RETURN a + b AS sum
}
RETURN a, b, sum
```

Subqueries allow you to isolate complex query logic, perform computations on imported variables, and return results back to the outer query.

In a read-only query the subquery hands its rows to the outer query as it finds them, and reads only as many as the rest of the query asks for: in `CALL { … UNION ALL … } RETURN x LIMIT 3`, a later `UNION` branch does not run when the earlier ones already gave 3 rows. A `UNION` inside `CALL { }` removes duplicates exactly as a top-level `UNION` does, so `1` and `1.0` count as the same value. *(Since v26.11.1: before, the whole subquery ran before its first row was returned, and a `UNION` inside it compared values by their text.)*

In a statement that does not write, the body of a `CALL { }`, `EXISTS { }`, `COUNT { }` or `COLLECT { }` is planned once per execution of the statement and run again for each outer row, instead of being planned again for every row. Correlated subqueries over many rows run up to about twice as fast; the rows are the same. `PROFILE` shows it as `Subquery step chains: <built> built, <reused> reused`. A statement that writes still plans its subqueries for every row, so a write made for one row is seen by the subquery of the next. *(Since v26.12.1, issue #12136)*

Subqueries IN TRANSACTIONS:

When importing large datasets (e.g., from LOAD CSV), you can use `CALL { … } IN TRANSACTIONS` to commit changes in batches rather than accumulating all changes in a single transaction. This prevents running out of memory and allows progress to be committed incrementally.

```
// Commit every 1000 rows (default batch size)
LOAD CSV WITH HEADERS FROM 'file:///data.csv' AS row
CALL {
  WITH row
  CREATE (n:Person {name: row.name, age: toInteger(row.age)})
} IN TRANSACTIONS

// Custom batch size
LOAD CSV WITH HEADERS FROM 'file:///data.csv' AS row
CALL {
  WITH row
  CREATE (n:Person {name: row.name})
} IN TRANSACTIONS OF 500 ROWS
```

`IN TRANSACTIONS` cannot be used inside a transaction you opened (HTTP `/begin`, Bolt `BEGIN` or a managed transaction, an embedded `begin()`): the subquery commits its own batches, which your rollback could not undo. The statement is refused with an `ImplicitTransactionRequiredException`, answered as HTTP `409 Conflict`, Bolt `Neo.DatabaseError.Transaction.TransactionStartFailed` (as Neo4j does) and PostgreSQL SQLSTATE `25001`. Run it as an auto-commit statement instead.

(Since v26.12.1) The rows a batch receives see what the earlier batches committed, even when those

rows were produced before the commits (for example by a `WITH … ORDER BY` in front of the `CALL`): a record an earlier batch wrote is read again before it is used, so writing it again no longer fails with a `ConcurrentModificationException`. To do this, a body that writes keeps the identities of the records it wrote, about 16 bytes each, until the statement ends.

15.3.25. FOREACH

The `FOREACH` clause iterates over a list and executes write operations (CREATE, SET, DELETE, MERGE) for each element. Unlike other clauses, FOREACH does not modify the result set - it passes through the input rows unchanged.

Syntax:

```
FOREACH (variable IN list |
  update_clause [update_clause ...]
)
```

Examples:

```
// Create multiple nodes from a list
FOREACH (name IN ['Alice', 'Bob', 'Charlie'] |
  CREATE (:Person {name: name})
)

// Create relationships in a loop
CREATE (root:Root {name: 'Main'})
FOREACH (i IN [1, 2, 3] |
  CREATE (root)-[:HAS_ITEM]->(:Item {id: i})
)

// Use with MATCH context
MATCH (p:Person)
FOREACH (tag IN ['developer', 'tester'] |
  CREATE (p)-[:HAS_TAG]->(:Tag {value: tag})
)

// Update properties with SET
MATCH (item:Item)
WITH collect(item) AS items
FOREACH (item IN items |
  SET item.status = 'processed', item.timestamp = timestamp()
)

// FOREACH passes through input rows
CREATE (root:Root {name: 'test'})
FOREACH (i IN [1, 2] |
  CREATE (:Child {id: i})
```

```
)
RETURN root.name AS name
// Returns: {name: 'test'}
```

Important Notes:

- FOREACH can only contain write operations (CREATE, SET, DELETE, MERGE)
- FOREACH does not modify the result set - it passes input rows through unchanged, and an empty list therefore leaves the query's rows exactly as they were
- Variables defined inside FOREACH are not visible outside the clause
- FOREACH is useful for batch operations and creating multiple elements from lists
- What a FOREACH body writes is subject to Reads and Writes in the Same Query like any other write

---

15.3.26. LOAD CSV

The `LOAD CSV` clause reads data from a CSV file (local or remote) and makes each row available as a variable for further query processing. This is the primary mechanism for bulk-importing data into ArcadeDB from CSV files.

Syntax:

```
LOAD CSV FROM <url> AS <variable>
LOAD CSV WITH HEADERS FROM <url> AS <variable>
LOAD CSV WITH HEADERS FROM <url> AS <variable> FIELDTERMINATOR <delimiter>
```

- **Without `WITH HEADERS`**: Each row is bound as a list of strings. Access fields by index: `row[0]`, `row[1]`, etc.
- **With `WITH HEADERS`**: The first row is treated as column names. Each subsequent row is bound as a map. Access fields by name: `row.name`, `row['age']`, etc.
- **FIELDTERMINATOR**: Specify a custom field delimiter (default is `,`). Example: `FIELDTERMINATOR ';'` for semicolon-separated files.

Basic Examples:

```
// Load CSV without headers (list access)
LOAD CSV FROM 'file:///people.csv' AS row
CREATE (:Person {name: row[0], age: toInteger(row[1])})

// Load CSV with headers (map access)
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
CREATE (:Person {name: row.name, age: toInteger(row.age)})

// Custom field delimiter
```

```
LOAD CSV WITH HEADERS FROM 'file:///data.tsv' AS row FIELDTERMINATOR '\t'
RETURN row.name, row.value

// Load from HTTP URL
LOAD CSV WITH HEADERS FROM 'https://example.com/data.csv' AS row
CREATE (:Item {id: row.id, label: row.label})
```

Bulk Import with CALL IN TRANSACTIONS:

For large files, combine `LOAD CSV` with `CALL { ... } IN TRANSACTIONS` to commit in batches and avoid memory issues:

```
LOAD CSV WITH HEADERS FROM 'file:///large-dataset.csv' AS row
CALL {
  WITH row
  CREATE (p:Person {name: row.name, email: row.email})
} IN TRANSACTIONS OF 1000 ROWS
```

Compressed Files:

LOAD CSV transparently decompresses `.gz` (gzip) and `.zip` files:

```
// Gzip compressed
LOAD CSV WITH HEADERS FROM 'file:///data.csv.gz' AS row
RETURN row.name

// ZIP compressed (reads the first entry in the archive)
LOAD CSV WITH HEADERS FROM 'file:///data.csv.zip' AS row
RETURN row.name
```

CSV Quoting:

Quoted fields follow RFC 4180 and also support backslash escaping:

- Fields containing the delimiter, newlines, or quotes must be enclosed in double quotes
- Embedded quotes can be escaped as `""` (RFC 4180) or `\"` (backslash escaping)
- Quoted fields can span multiple lines

Security Configuration:

LOAD CSV file access is controlled by two configuration settings:

Setting	Default	Description
`arcadedb.opencypher.loadCsv.allowFileUrls`	`true`	Allow access to local files via `file:///` URLs and bare file paths. Set to `false` in multi-tenant server deployments.

Setting	Default	Description
`arcadedb.opencypher.loadCsv.importDirectory`	(empty)	Root directory for file URLs. When set, file paths are resolved relative to this directory and path traversal (`../`) is blocked. Empty means no restriction.

HTTP/HTTPS URLs are always allowed regardless of these settings.

(Since v26.12.1) A URL that cannot be opened or read (a file that does not exist, a host that does not answer) is a client error, as in Neo4j: HTTP `400` and `Neo.ClientError.Statement.ExternalResourceFailed` over Bolt. Before, it answered `500` and `Neo.DatabaseError.General.UnknownError`.

Example — Restrict to import directory:

```
// In server configuration
arcadedb.opencypher.loadCsv.importDirectory=/var/arcadedb/import
```

```
// This works (resolved to /var/arcadedb/import/data.csv)
LOAD CSV FROM 'file:///data.csv' AS row RETURN row

// This is blocked (path traversal)
LOAD CSV FROM 'file:///../etc/passwd' AS row RETURN row
```

Built-in Helper Functions:

Two special functions are available within LOAD CSV queries:

- `file()` — Returns the URL of the currently loaded CSV file
- `linenumber()` — Returns the current line number being processed (1-based)

Both functions work anywhere in the rest of the query following `LOAD CSV`, including inside a `WHERE` predicate — not only in `RETURN`/`WITH` projections — and they remain valid even after a `WITH` that does not re-project the CSV row variable, and after a `WITH ⋯ ORDER BY`. This makes `WITH row WHERE linenumber() > 1 ⋯`, the common idiom for skipping a header row, work as expected.

Both functions describe the CSV row currently being processed, so an aggregation ends them: after a clause such as `WITH collect(row) AS rows` or `WITH count(*) AS total`, many rows have been folded into one and there is no single line for `linenumber()` to report, so `file()` and `linenumber()` return `null` for the rest of the query — including after an `UNWIND` that expands the collected list back into rows. This matches Neo4j. If you need the file or line further downstream, capture it into a variable before the aggregation, for example `WITH row, file() AS csvFile, linenumber() AS csvLine`.

15.3.27. CREATE CONSTRAINT

The `CREATE CONSTRAINT` clause defines schema constraints on node or relationship properties. Constraints enforce data integrity rules such as uniqueness, mandatory (NOT NULL), and node keys.

ArcadeDB maps Cypher constraints to its native schema features:

- **IS UNIQUE** → Creates a unique LSM-Tree index on the specified properties
- **IS NOT NULL** → Sets the property as mandatory in the schema
- **IS NODE KEY** / **IS KEY** → Creates a unique index and sets the property as mandatory
- **IS TYPED `<type>`** → Declares the property's data type in the schema (see IS TYPED data types)

Only `IS TYPED` declares a type you choose. `IS UNIQUE` leaves an undeclared property undeclared, like `CREATE INDEX`, so every value keeps the type it is written with. Its index keys such a property by the text of each value, so `2` and `2.0` are one key as Cypher wants, but `2` and `'2'`, or `true` and `'true'`, are one key too: the second of them is refused as a duplicate, where Neo4j accepts both. Declare the type first (`IS TYPED`) when a property must hold values of several kinds that read the same. `IS NOT NULL` and `IS NODE KEY` need a declared property to make it mandatory: on an undeclared property they declare it with the type every stored value shares (for example `LONG` over integers, the widest precision over datetimes), or `STRING` when no record has the property yet. A declared property converts every value written to it, so declare the type yourself first (`IS TYPED`, or SQL `CREATE PROPERTY`) when the inferred one is not the one you want. *(Since v26.12.1)* Before 26.12.1 `IS UNIQUE` declared the property from the stored values too, so on a label of integers `2.5` was stored as `2` and then refused as a duplicate of the `2` already there.

Syntax:

```
CREATE CONSTRAINT [name] [IF NOT EXISTS]
FOR (variable:Label)
REQUIRE variable.property IS UNIQUE

CREATE CONSTRAINT [name] [IF NOT EXISTS]
FOR (variable:Label)
REQUIRE variable.property IS NOT NULL

CREATE CONSTRAINT [name] [IF NOT EXISTS]
FOR (variable:Label)
REQUIRE variable.property IS NODE KEY

CREATE CONSTRAINT [name] [IF NOT EXISTS]
FOR (variable:Label)
REQUIRE variable.property IS TYPED <type>
```

`IS

<type>` is a synonym of `IS TYPED <type>`.

Examples:

```
// Unique constraint on a single property
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.email IS UNIQUE

// Named constraint with IF NOT EXISTS (idempotent)
CREATE CONSTRAINT unique_person_email IF NOT EXISTS
FOR (p:Person) REQUIRE p.email IS UNIQUE

// Composite unique constraint (multiple properties)
CREATE CONSTRAINT FOR (p:Person) REQUIRE (p.firstName, p.lastName) IS UNIQUE

// NOT NULL constraint (mandatory property)
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.name IS NOT NULL

// Node key constraint (unique + mandatory)
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.id IS NODE KEY

// Relationship constraint
CREATE CONSTRAINT FOR ()-[r:KNOWS]-() REQUIRE r.since IS NOT NULL

// Typed constraint: declare the property's data type
CREATE CONSTRAINT FOR (m:Measure) REQUIRE m.temperature IS TYPED FLOAT32
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.age IS TYPED INT16

// Typed constraint on a list: declares the property as LIST OF STRING
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.tags IS :: LIST<STRING NOT NULL>
```

If the property does not already exist in the schema, ArcadeDB will automatically create it as a STRING type. For other types, define the property first using SQL (`ALTER TYPE Person CREATE PROPERTY email STRING`).

A constraint name is unique across the whole database. *(Since v26.10.2)* `CREATE CONSTRAINT <name>` with a name already in use raises an error before creating anything; with `IF NOT EXISTS` it is a no-op. Running the same statement with `IF NOT EXISTS` several times therefore never raises.

Constraint names:

(Since v26.10.2) A constraint created without a name gets a generated one, `constraint_<Label>_<property>[<property>…]<kind>`, where the kind is `unique`, `key`, `existence` (for `IS NOT NULL`) or `type` (for `IS TYPED`):

Statement	Generated name
`CREATE CONSTRAINT FOR (p:Person) REQUIRE p.email IS UNIQUE`	`constraint_Person_email_unique`
`CREATE CONSTRAINT FOR (p:Person) REQUIRE (p.firstName, p.lastName) IS UNIQUE`	`constraint_Person_firstName_lastName_unique`
`CREATE CONSTRAINT FOR (p:Person) REQUIRE p.id IS NODE KEY`	`constraint_Person_id_key`

Statement	Generated name
`CREATE CONSTRAINT FOR (p:Person) REQUIRE p.name IS NOT NULL`	`constraint_Person_name_existence`
`CREATE CONSTRAINT FOR (p:Person) REQUIRE p.age IS TYPED INT16`	`constraint_Person_age_type`

The name is derived from the statement, so repeating the same unnamed statement does not create a second constraint. It is the name `SHOW CONSTRAINTS` lists and `DROP CONSTRAINT` takes.

(Since v26.10.2) A property carries one property type constraint at most: a second `IS TYPED` constraint that requires a different type of the same property is refused, as in Neo4j. Drop the first one before creating the other.

When existence constraints are checked:

A property declared mandatory with `IS NOT NULL` (or `IS NODE KEY`) is checked when the statement finishes, not at the instant the node or relationship is created. This is what makes the standard upsert work, since the `SET` supplies the property after the pattern has created the node:

```
CREATE CONSTRAINT rec_org_exists IF NOT EXISTS FOR (n:Record) REQUIRE n.orgId IS NOT
NULL;

MERGE (n:Record {id: $id}) SET n.orgId = $orgId RETURN n
```

Any clause of the same statement may supply the property. A statement that ends without supplying it fails, and the incomplete record is removed.

Only `CREATE`/`MERGE` pattern writes are deferred this way. Every other constraint (`IS TYPED`, `min`, `max`, regular expressions, read-only) is still checked at the moment of the write, as are writes issued from SQL, the Java API and `CALL` procedures.

Between the pattern creating the record and the statement ending, another connection can briefly read a record that does not yet satisfy its existence constraints. If the server stops in that window, the incomplete record is left behind.

A record left behind that way is not lost track of: it is, by definition, a record that does not satisfy its own type's existence constraints, and `CHECK DATABASE` asks exactly that question of every record it scans. The findings come back under `constraintViolatingRecords`, and `CHECK DATABASE FIX DELETE INVALID RECORDS` removes them; a record that holds data worth keeping is instead completed by a single `UPDATE` supplying every missing property. See Existence constraints.

IS TYPED data types:

`IS TYPED <type>` declares the property's data type in the schema. ArcadeDB maps the ISO/IEC 39075 (GQL) type names onto its native types. Numeric values are stored and reloaded at the declared width, so a value written to a property declared `IS TYPED INT16` round-trips as a 16-bit integer and continues to satisfy the `x IS TYPED INT16` value-type predicate.

GQL / Cypher type	ArcadeDB type	Notes
`BOOLEAN`, `BOOL`	`BOOLEAN`	
`STRING`, `VARCHAR`	`STRING`	
`INT8`, `INTEGER8`	`BYTE`	8-bit signed integer
`INT16`, `INTEGER16`	`SHORT`	16-bit signed integer
`INT`, `INT32`, `INTEGER32`	`INTEGER`	32-bit signed integer (`INT` is the alias for `INT32`)
`INTEGER`, `INT64`, `INTEGER64`, `SIGNED INTEGER`	`LONG`	64-bit signed integer (`INTEGER` is the generic integer type)
`FLOAT32`	`FLOAT`	32-bit IEEE 754 floating point
`FLOAT`, `FLOAT64`	`DOUBLE`	64-bit IEEE 754 floating point
`DATE`	`DATE`	
`LOCAL DATETIME`, `TIMESTAMP WITHOUT TIME ZONE`	`DATETIME_NANOS`	Nanosecond precision. A property already declared `DATETIME`, `DATETIME_SECOND` or `DATETIME_MICROS` is left at its precision
`ZONED DATETIME`, `TIMESTAMP WITH TIME ZONE`	`ZONED_DATETIME`	Keeps the zone
`LOCAL TIME`, `TIME WITHOUT TIME ZONE`	`LOCAL_TIME`	
`ZONED TIME`, `TIME WITH TIME ZONE`	`OFFSET_TIME`	Keeps the offset
`DURATION`	`DURATION`	
`POINT`	`MAP`	The map `point()` returns
`LIST<T>`, `T LIST`, `ARRAY<T>`, `LIST<T NOT NULL>`	`LIST OF` the type `T` maps to	For example `LIST<INTEGER>` is `LIST OF LONG`, `LIST<FLOAT>` is `LIST OF DOUBLE`, `LIST<INT32>` is `LIST OF INTEGER`, `LIST<DATE>` is `LIST OF DATE`
`LIST`, `LIST<ANY>`	`LIST`	A list with no element type
`MAP`	`MAP`	

The `LOCAL DATETIME`, `ZONED DATETIME`, `LOCAL TIME`, `ZONED TIME` and `DURATION` types map onto the native ArcadeDB types that hold the value without loss (see Managing Dates). A list of lists (`LIST<LIST<STRING>>`) and a union type (`STRING | INTEGER`) cannot be declared and are refused.

A `NOT NULL` on the type itself, as in `REQUIRE n.p IS`

STRING NOT NULL`, is refused, as in Neo4j: existence is a constraint of its own. Declare the type and add an existence constraint:

```
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.name IS :: STRING
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.name IS NOT NULL
```

A `NOT NULL` on the element type of a list (`LIST<STRING NOT NULL>`) is accepted.

Re-typing a declared property:

When the property is already declared with another type, `IS TYPED <type>` (or `IS <type>`) re-declares it with the new type. *(Since v26.10.2)* The attributes that do not depend on the type are always kept: `MANDATORY`, `NOTNULL`, `READONLY`, `HIDDEN`, `EXTERNAL`, `COMPRESSION` and custom values. The type-dependent ones are kept only when the new type reads them the same way:

- `OF` when both types are a `LIST` or a `MAP`;
- `REGEXP` when the new type is `STRING`;
- `MIN` and `MAX` when both types measure the same thing (characters, bytes, items, a numeric value or a date) and the new type can hold the bound, so `max 4` moves from `INTEGER` to `LONG` but not from `STRING` to `LONG`, and `max 2.5` does not move onto an `INTEGER`;
- `DEFAULT` when it converts to the new type without changing its value (a `DOUBLE` default `3.0` moves onto a `LONG`, `3.7` does not).

Otherwise the statement is refused with an error naming each attribute that blocks it, and the declaration is left unchanged: remove or change the attribute with `ALTER PROPERTY` first. The stored records are then checked against the new declaration, as `CREATE PROPERTY` checks them: if a record holds a value of another type, or breaks a kept `NOT NULL`, `MANDATORY`, `REGEXP`, `MIN` or `MAX`, the constraint is refused and the previous declaration is put back.

Following Cypher 25 / GQL, `INTEGER` is the 64-bit generic integer type while `INT` is the alias for the 32-bit `INT32`. A property declared `IS TYPED INT` therefore holds 32-bit values; use `IS TYPED INTEGER` (or `INT64`) for 64-bit values.

When upgrading to 26.10.2: before, a property type constraint refused a list with an element type (`LIST<STRING>`), `LOCAL DATETIME` and `ZONED DATETIME` declared a `DATETIME` (millisecond precision, which truncated the sub-millisecond digits and dropped the zone), `LOCAL TIME` and `ZONED TIME` declared a `STRING`, `DURATION` declared a `LONG` (which refused every duration value) and `POINT` declared a `STRING`. A constraint created before 26.10.2 keeps the declaration it made then.

IS :: VECTOR property type constraint

***(Since v26.11.1)* `CREATE CONSTRAINT ... REQUIRE n.e IS VECTOR<T>(n)`, also spelled `VECTOR(n, T)`, declares the property as the array type of the coordinate type and enforces the dimension on every write:**

```
CREATE CONSTRAINT embedding_vector FOR (d:Doc) REQUIRE d.embedding IS :: VECTOR
<FLOAT32>(1536)
```

Coordinate type	Property type
`FLOAT32`	`ARRAY_OF_FLOATS`
`FLOAT`	`ARRAY_OF_DOUBLES`
`INTEGER`	`ARRAY_OF_LONGS`

Coordinate type	Property type
INT32	ARRAY_OF_INTEGERS
INT16	ARRAY_OF_SHORTS
INT8	BINARY

The dimension is enforced through the property's MIN and MAX, which count array items. `IS

VECTOR` matches a number array of the right type and length (a list does not), so a constraint the stored data already breaks is refused and rolled back. DROP CONSTRAINT hands the bounds to an equivalent constraint with the same dimension, or clears them. A type constraint can now retype a property to an array type, and dropping one of two equivalent type constraints no longer puts the old declaration back while the other still requires the new type. A fixed-size array declared outside Cypher shows in SHOW CONSTRAINTS under its ArcadeDB name.

The Cypher vector(v, n, T) function builds the array of its coordinate type: INT8 gives a byte[], INT16 a short[], INT32 an int[], INTEGER a long[], FLOAT32 a float[] and FLOAT a double[], so `vector([1,2,3], 3, INT8) IS

VECTOR<INT8>(3)` is true. An integer coordinate must be a whole number inside the range of the type, otherwise the call fails and nothing wraps. A null type still means FLOAT32. A byte[] that reaches the Cypher vector functions and the vector.* SQL functions is read as plain INT8 coordinates, not a quantized vector, except by vector.dequantizeInt8.

15.3.28. DROP CONSTRAINT

The DROP CONSTRAINT clause removes a previously created constraint by its name: the name given in CREATE CONSTRAINT, the generated name of a constraint created without one, or any name SHOW CONSTRAINTS lists.

Syntax:

```
DROP CONSTRAINT constraintName [IF EXISTS]
```

Examples:

```
// Drop a constraint by name
DROP CONSTRAINT constraint_Person_email_unique

// Drop the constraint that created the unique index Person[email]
DROP CONSTRAINT `Person[email]`

// Drop only if it exists (no error if missing)
DROP CONSTRAINT `Person[email]` IF EXISTS
```

The name of the unique index a UNIQUE or NODE KEY constraint created is accepted too, and drops the constraint that created it: an unnamed NODE KEY dropped by its

index name is dropped whole, index and existence requirement. Use backtick escaping for names containing special characters like brackets.

(Since v26.10.2) dropping a name that no constraint or index carries raises an error, unless `IF EXISTS` is given, which makes the statement a no-op.

What dropping a constraint undoes:

(Since v26.10.2) `DROP CONSTRAINT` undoes what the constraint created:

- A `UNIQUE` constraint drops its unique index, a `NOT NULL` constraint clears the mandatory flag it set, and a `NODE KEY` / `KEY` constraint does both.
- Only what the constraint itself created is undone. A `MANDATORY` flag or a unique index declared before the constraint, for example with SQL `CREATE PROPERTY ⋯ (mandatory true)` or `CREATE INDEX`, is kept when the constraint is dropped, and so is an index or a flag another named constraint still needs.
- A property type constraint (`IS TYPED`) lifts the type restriction. If the constraint declared the property, the declaration is removed. If it replaced an earlier declaration, for example one made with SQL `CREATE PROPERTY`, that declaration is put back; when records written since cannot be read as the old type, the drop is refused and the constraint stays in place.
- While another constraint still needs the declaration, the type stays until that one is dropped too: a `NOT NULL` constraint needs a declared property to carry its mandatory flag, and a `UNIQUE` index is built over the declared type.
- A requirement the schema enforces without a Cypher constraint, which `SHOW CONSTRAINTS` lists under a generated name, is dropped the same way: `constraint_<Type>_<property>_existence` clears the property's `MANDATORY` and `NOT NULL` flags, `constraint_<Type>_<property>_type` removes the property declaration (refused while the declaration also carries an index or other attributes, such as a default or a min/max, that would go with it), and the name of a unique index drops the index.

When upgrading to 26.10.2: before, dropping a property type constraint left the property declared with the constraint's type, so the type restriction stayed in force after the constraint was gone.

15.3.29. SHOW CONSTRAINTS

`SHOW CONSTRAINTS` lists the constraints of the database with Neo4j's syntax and columns, so the tools and drivers written for Neo4j read it unchanged. *(Since v26.10.2)* It is answered by the openCypher engine, so it works the same over every protocol that runs Cypher: HTTP, Studio, Bolt and the embedded Java API.

Syntax:

```
SHOW [ALL] CONSTRAINTS
SHOW [NODE | RELATIONSHIP] {UNIQUE | UNIQUENESS | EXIST | EXISTENCE | KEY | PROPERTY
```

```
TYPE} CONSTRAINTS
```

The keyword before `CONSTRAINTS` restricts the list to one kind of constraint, and a preceding `NODE` or `RELATIONSHIP` to the constraints of vertex types or of edge types. The command accepts `YIELD`, `WHERE`, `ORDER BY`, `SKIP`, `LIMIT` and `RETURN` like every `SHOW` command, see Filtering the output of a SHOW command.

Result columns:

Column	Description
`id`	A sequence number of the row
`name`	The constraint name, the one `DROP CONSTRAINT` takes
`type`	`UNIQUENESS`, `RELATIONSHIP_UNIQUENESS`, `NODE_KEY`, `RELATIONSHIP_KEY`, `NODE_PROPERTY_EXISTENCE`, `RELATIONSHIP_PROPERTY_EXISTENCE`, `NODE_PROPERTY_TYPE` or `RELATIONSHIP_PROPERTY_TYPE`
`entityType`	`NODE` or `RELATIONSHIP`
`labelsOrTypes`	The label or relationship type, as a list
`properties`	The constrained properties, as a list
`ownedIndex`	The unique index that enforces a uniqueness or key constraint, otherwise `null`
`propertyType`	The Cypher type a property type constraint requires, otherwise `null`

`YIELD *` adds the columns `enforcedLabel`, `options` and `createStatement`, the `CREATE CONSTRAINT` statement that recreates the constraint.

The list shows what the schema enforces, whether it was created with Cypher, SQL or the Java API. A requirement that no Cypher constraint created is listed as follows:

- a unique index, as a uniqueness constraint named after the index;
- a property declared `MANDATORY` or `NOT NULL`, as an existence constraint named `constraint_<Type>_<property>_existence`;
- a declared property type, as a property type constraint named `constraint_<Type>_<property>_type`.

Every name the list shows can be dropped with `DROP CONSTRAINT`. The types the user is not allowed to read are left out.

Examples:

```
// Every constraint
SHOW CONSTRAINTS

// The uniqueness constraints of vertex types
SHOW NODE UNIQUENESS CONSTRAINTS
```

```
// The property type constraints of one label
SHOW PROPERTY TYPE CONSTRAINTS YIELD name, labelsOrTypes, properties, propertyType
WHERE 'Person' IN labelsOrTypes

// The statements that recreate them
SHOW CONSTRAINTS YIELD name, createStatement ORDER BY name
```

15.3.30. CREATE INDEX / DROP INDEX

`CREATE INDEX` creates a (non-unique) `LSM_TREE` index on one or more properties of a label or relationship type, with Neo4j's syntax. `RANGE` is accepted as a synonym of the plain form.

```
CREATE INDEX person_name FOR (p:Person) ON (p.name)
CREATE INDEX IF NOT EXISTS FOR (p:Person) ON (p.firstName, p.lastName)
CREATE INDEX follows_since FOR ()-[r:FOLLOWS]-() ON (r.since)

DROP INDEX person_name
DROP INDEX person_name IF EXISTS
```

(Since v26.10.2) The index is created under the name the statement gives it, which is the name `SHOW INDEXES` lists, `DROP INDEX` takes, and SQL sees. An index created without a name is named `<Type>[<property>,⋯]`, for example `Person[name]`: use backtick escaping to drop it, `DROP INDEX` `` `Person[name]` ``. Before 26.10.2 the name was ignored, so `DROP INDEX person_name` dropped nothing.

Index and constraint names share one namespace, as in Neo4j:

- a name already taken by an index or a constraint is refused, and with `IF NOT EXISTS` the statement does nothing;
- an index already defined on the same properties is refused under another name, and with `IF NOT EXISTS` the statement does nothing;
- `DROP INDEX` of a name no index has raises an error unless `IF EXISTS` is given;
- the index a `UNIQUE` or `NODE KEY` constraint created is dropped with `DROP CONSTRAINT`: `DROP INDEX` refuses it, since the constraint would stay listed with nothing enforcing it;
- *(Since v26.12.1)* `DROP INDEX` of the bucket index of an index (for example `S_0_4559193709411875`, the part of `S[k]` on one bucket) is refused, as in SQL: the index would miss every record of that bucket. Drop the index itself, which drops all its bucket indexes.

Property types:

As in Neo4j, an index does not constrain the values of a property: `CREATE INDEX` on a property the schema does not declare leaves it undeclared, whether the label already has data or not, and every value keeps the type it is written with (`2.5`, `7`, `'7'`, `date('2024-05-06')` on the same property). The index keys such a property by the text of each value, with one key per number however it is written (`3`, `3.0`, `3.00`), and the query re-checks every row, so `n.x = 7` and `n.x = '7'` each find only their own node. Equality, `IN` and string ranges (`STARTS WITH`, `n.x > 'm'`) go through the index; a numeric or

temporal range (`n.x > 10`, `n.ts > datetime(⋯)`) reads the whole index and filters it, with the right answer but without narrowing the scan.

(Since v26.12.1) An index of this kind written by an older version can hold floating point numbers and lists keyed by an older text (`1.0`, `[1.5, 2.0]` instead of `1`, `[1.5, 2]`). It still finds, removes and refuses copies of them, at the cost of an extra read on such values until `REBUILD INDEX` rewrites its keys. See the upgrade note.

To keep typed keys, and with them indexed numeric and temporal ranges, declare the type before creating the index:

```
CREATE CONSTRAINT FOR (p:Person) REQUIRE p.age IS :: INTEGER
CREATE INDEX FOR (p:Person) ON (p.age)
```

A declared property converts every value written to it to its type. *(Since v26.12.1)* Before 26.12.1, `CREATE INDEX` on a label that already held the property declared it with the type of the stored values, so every later write of another kind was converted silently: on a label of integers `2.5` was stored as `2`, `'7'` as the number `7` and a date as epoch milliseconds. A property declared that way by an earlier version stays declared: drop the declaration with SQL `DROP PROPERTY` (after dropping the index) if the values should keep their own types.

Nodes with no label, or several:

(Since v26.12.1) A node pattern that names no single label is answered by the indexes too. `MATCH (n {uuid: $u})`, `MATCH (n) WHERE n.uuid = $u`, `MATCH (n:A:B {uuid: $u})` and a list of values, `MATCH (n) WHERE n.uuid IN $keys`, look the values up in the index of every type that has one on the property, and scan only the types that have none. Before 26.12.1 such a pattern read every node of the database, so it got slower as the database grew.

The index is used when the value has the type the index is keyed by (text for a text property, a whole number for a whole number property, a boolean, a decimal number for a decimal property). Any other value is compared by scanning, as before. A list is looked up in an index only when the index is on that one property alone and every value of the list has the type it is keyed by: one value of another type (`2.5` against a whole number property, `'7'` against a number property) makes that index scan its types for the whole list. Writing the label on the node, as in `MATCH (n:Entity {uuid: $u})`, still names the index directly.

CREATE VECTOR INDEX / CREATE FULLTEXT INDEX

(Since v26.11.1) Neo4j's `CREATE VECTOR INDEX` creates an `LSM_VECTOR` index and `CREATE FULLTEXT INDEX` a `FULL_TEXT` index, so a Neo4j application can create the indexes `SEARCH`, `db.index.vector.queryNodes` and `db.index.fulltext.queryNodes` query. Both keep the given name and honour `IF NOT EXISTS`, on a label or on a relationship type. A vector index over existing records is built before the statement returns, so it is `ONLINE` at once and needs no `db.awaitIndex`.

```
CREATE VECTOR INDEX moviePlots IF NOT EXISTS FOR (m:Movie) ON m.embedding
OPTIONS { indexConfig: { `vector.dimensions`: 1536, `vector.similarity_function`:
'cosine' } }
```

```
CREATE VECTOR INDEX moviePlots FOR (m:Movie) ON m.embedding
OPTIONS { indexConfig: { `vector.dimensions`: 1536, `vector.quantization.type`:
'scalar', efSearch: 200 } }

CREATE FULLTEXT INDEX titles FOR (n:Movie) ON EACH [n.title, n.tagline]
OPTIONS { indexConfig: { `fulltext.analyzer`: 'english', defaultOperator: 'AND' } }
```

An undeclared vector property is declared `ARRAY_OF_FLOATS`. The Neo4j settings of `indexConfig` map onto the index settings:

Neo4j setting	ArcadeDB setting
`vector.dimensions` (required)	`dimensions`
`vector.similarity_function`: `cosine` (default), `euclidean`	`similarity`: `COSINE`, `EUCLIDEAN` (`dot_product` is accepted too)
`vector.quantization.type`: `none`, `scalar`, `binary`	`quantization`: `NONE`, `INT8`, `BINARY`
`vector.quantization.enabled` (deprecated by Neo4j)	`quantization`: `INT8` when `true`, `NONE` when `false`. It must agree with `vector.quantization.type` when both are given
`vector.hnsw.m`	`maxConnections`, doubled: JVector does not double the degree at the base layer as an HNSW index does
`vector.hnsw.ef_construction`	`beamWidth`
`fulltext.analyzer`	`analyzer`: a Neo4j analyzer name (`standard-no-stop-words`, `english`, `simple`, ...) or the class name of a Lucene analyzer
`fulltext.eventually_consistent`	accepted with either value: the index is updated by the transaction that writes the record

With no quantization setting the index keeps the exact vectors (`NONE`), while Neo4j quantizes by default (`binary` since Neo4j 2026.08, `scalar` before): a quantized index has a lower recall, which a script must ask for. `vector.default_search_expansion_factor` is refused: the search beam of an `LSM_VECTOR` index is `efSearch`.

Every other setting of the index is given in `indexConfig` under its own name, the one the `METADATA` of SQL `CREATE INDEX` takes (`efSearch`, `quantization: 'PRODUCT'`, `pqSubspaces`, `bm25_k1`, `<property>_boost`, ...), and is checked the same way. A setting named both ways (`vector.hnsw.m` and `maxConnections`) or not known is refused, and the statement then creates nothing.

The `createStatement` column of `SHOW INDEXES` names every setting of a vector or full-text index that differs from its default, and always the quantization, so it creates the same index again (an odd `maxConnections` has no `vector.hnsw.m` spelling, so it is listed under its own name); the `options` column holds the same `indexConfig` with the `indexProvider`.

Further behavior, all since v26.11.1:

- **Several labels or relationship types.** `CREATE FULLTEXT INDEX docs FOR (n:Article|Post) ON EACH [n.title, n.body]` and `CREATE VECTOR INDEX emb FOR ()-[r:LIKES|RATES]-() ON r.embedding`

build one index per type and record a logical name on every labelled type, persisted and replicated with the schema. `db.index.fulltext.queryNodes`, `queryRelationships`, `db.index.vector.queryNodes` and the `SEARCH` clause resolve the name, `SHOW INDEXES` lists one row with every label, `DROP INDEX <name>` drops only the indexes the statement built, and `IF NOT EXISTS` and name uniqueness apply. A failed statement drops the indexes and properties it created. A vector index over several types needs the same `vector.dimensions` and `vector.similarity_function` on each. Full-text scores are merged by raw score: BM25 statistics are per per-type index, so a score is not exactly the one a single Neo4j index over all the labels would give. From SQL, each per-type index is an ordinary index (`SEARCH_INDEX('Post[title,body]', …)`, `vector.neighbors('Post[e]', …)`); the logical name itself is not known to SQL.

- **Filter properties.** `WITH […]` filter properties of a vector index are accepted and build the same index as without them: the `SEARCH` clause filters on the record itself, so any property can be filtered on, declared or not. `SHOW INDEXES` does not list them.
- **No `vector.dimensions`.** The dimension is taken from the first vector the indexed property already holds. With no vector yet, the index is created unsized (the `LSM_VECTOR` metadata key `sizeOnFirstVector`, also usable from SQL) and takes its dimension from the first vector it receives, with no upper bound; a vector of another length is then refused like with any sized index. `SHOW INDEXES` lists `vector.dimensions` once decided. SQL without that flag is still refused.
- **Analyzer.** `fulltext.analyzer: 'standard'` maps to a standard tokenizer with lower-casing and the English stop words, without stemming, and `SHOW INDEXES` lists it back as `'standard'`.

15.3.31. SHOW INDEXES

`SHOW INDEXES` lists the indexes of the database with Neo4j's syntax and columns. *(Since v26.10.2)* Like `SHOW CONSTRAINTS`, it is answered by the openCypher engine over every protocol (HTTP, Studio, Bolt, embedded).

Syntax:

```
SHOW [ALL | RANGE | TEXT | POINT | VECTOR | FULLTEXT | LOOKUP] INDEXES
```

The keyword after `SHOW` restricts the list to one type of index, matched against the `type` column. ArcadeDB has no `TEXT` or `LOOKUP` index, so those two list nothing. `YIELD`, `WHERE`, `ORDER BY`, `SKIP`, `LIMIT` and `RETURN` work as on every `SHOW` command.

Result columns:

Column	Description
`id`	A sequence number of the row
`name`	The index name

Column	Description
`state`	`ONLINE`, `POPULATING` while the index is being built, or `FAILED` when the index lacks the part of one of the buckets it covers (left by an interrupted rebuild of an earlier version): no query uses it until `REBUILD INDEX` repairs it. *(`FAILED` since v26.12.1; before, such an index was listed as `POPULATING` or `ONLINE`.)*
`populationPercent`	`100.0` when the index is online, otherwise `0.0`
`type`	`RANGE` (an `LSM_TREE` index), `HASH`, `FULLTEXT`, `VECTOR`, `SPARSE_VECTOR` or `POINT` (a geospatial index)
`entityType`	`NODE` or `RELATIONSHIP`
`labelsOrTypes`	The label or relationship type, as a list
`properties`	The indexed properties, as a list
`indexProvider`	The index implementation, for example `range-1.0`
`owningConstraint`	The name of the constraint that created the index (for a unique index created without a constraint, the index name, which is the name `SHOW CONSTRAINTS` lists it under), otherwise `null`
`lastRead`, `readCount`	Always `null`: ArcadeDB does not track index reads

`YIELD *` adds the columns `trackedSince`, `options`, `failureMessage` and `createStatement`.

Examples:

```
// Every index
SHOW INDEXES

// The indexes no constraint owns
SHOW INDEXES YIELD name, type, labelsOrTypes, properties, owningConstraint
WHERE owningConstraint IS NULL

// Count the vector indexes
SHOW VECTOR INDEXES YIELD name RETURN count(name) AS total
```

15.3.32. SHOW USERS

The `SHOW USERS` command lists all users configured on the server, along with the databases each user has access to.

This command requires ArcadeDB to be running in **server mode** (it does not work in embedded mode). See Security - Users for more information about the ArcadeDB security model.

All user management commands are non-idempotent and require a POST request (command endpoint) when executed via the HTTP API.

Syntax:

```
SHOW USERS
```

Result columns:

Column	Description
`user`	The user name
`databases`	The set of databases the user has access to (`*` means all databases)

Examples:

```
// List all users
SHOW USERS

// Only the rows that match
SHOW USERS WHERE user = $name

// Pick the columns to return
SHOW USERS YIELD user WHERE user STARTS WITH 'app_'
```

Filtering the output of a SHOW command:

Every `SHOW` command accepts an optional trailing `WHERE`, or a `YIELD` that selects columns and may itself be followed by `ORDER BY`, `SKIP`, `LIMIT`, `WHERE` and `RETURN`:

```
SHOW DATABASES WHERE name = $dbName
SHOW DATABASES YIELD name, currentStatus WHERE name STARTS WITH 'prod'
SHOW DATABASES YIELD name ORDER BY name SKIP 1 LIMIT 5
SHOW INDEXES YIELD name, type RETURN count(name) AS total
```

The filter is evaluated by the Cypher engine, so operators, functions and query parameters behave exactly as they do in any other query. Checking whether a database exists is therefore a matter of asking for it by name and seeing whether a row comes back:

```
SHOW DATABASES WHERE name = $dbName
```

A trailing `;` is accepted. A `UNION` of several commands is answered branch by branch, so a trailing clause there applies only within its own branch.

15.3.33. SHOW CURRENT USER

The `SHOW CURRENT USER` command returns information about the currently authenticated user executing the query.

Syntax:

```
SHOW CURRENT USER
```

Result columns:

Column	Description
`user`	The current user name
`databases`	The set of databases the current user has access to

Over the Bolt protocol, `SHOW CURRENT USER` and `CALL dbms.showCurrentUser()` answer the columns Neo4j drivers and tools expect instead: `user`, `roles`, `passwordChangeRequired`, `suspended` and `home`. `roles` lists the groups the user holds on the database the session is bound to, sorted, so ArcadeDB's `admin` group is Neo4j's `admin` role; `passwordChangeRequired` and `suspended` are always `false`, since ArcadeDB has neither. *(Since v26.11.1: `roles` was always `["admin"]`, whatever the user was granted.)*

Examples:

```
// Show the currently authenticated user
SHOW CURRENT USER
```

15.3.34. CREATE USER

The `CREATE USER` command creates a new user on the server with the specified password. The new user is assigned to the `admin` group on all databases (`*`) by default.

Syntax:

```
CREATE USER username SET PASSWORD 'password'

CREATE USER username IF NOT EXISTS SET PASSWORD 'password'
```

Examples:

```
// Create a new user
CREATE USER bob SET PASSWORD 'SecurePass123!'

// Create a user only if it doesn't already exist (idempotent)
```

```
CREATE USER bob IF NOT EXISTS SET PASSWORD 'SecurePass123!'
```

The password is stored hashed using PBKDF2 — it is never stored in plain text. See Security - Users for details on password storage.

`IF NOT EXISTS` makes the operation safe to run multiple times without error if the user already exists.

15.3.35. DROP USER

The `DROP USER` command removes a user from the server.

Syntax:

```
DROP USER username

DROP USER username IF EXISTS
```

Examples:

```
// Drop a user (error if user doesn't exist)
DROP USER bob

// Drop a user only if it exists (no error if missing)
DROP USER bob IF EXISTS
```

`IF EXISTS` makes the operation safe to run even when the user does not exist.

15.3.36. ALTER USER

The `ALTER USER` command changes the password of an existing user.

Syntax:

```
ALTER USER username SET PASSWORD 'newPassword'
```

Examples:

```
// Change a user's password
ALTER USER bob SET PASSWORD 'NewSecurePass456!'
```

The user must already exist. If the user does not exist, an error is returned.

15.3.37. SHOW TRANSACTIONS and TERMINATE TRANSACTIONS

`SHOW TRANSACTIONS` lists the statements the server is running and `TERMINATE TRANSACTIONS` stops them, with Neo4j's syntax, so the tools and scripts written for Neo4j work unchanged. Available since 26.10.2.

Both commands read the same registry as the HTTP `list queries` and `terminate query` server commands: a "transaction" here is one running statement, whatever protocol it came from (HTTP, PostgreSQL, Bolt, gRPC, MongoDB, Redis, Gremlin), and its `transactionId` is the id `list queries` shows (`q<n>-<hex>`). The same rule decides what you see: **root** sees and stops every statement, any other user only their own.

These commands require ArcadeDB to be running in **server mode**. They work over Bolt and every other protocol that runs Cypher; in embedded mode they fail with "SHOW TRANSACTIONS and TERMINATE TRANSACTIONS require a server". On an HA cluster each server answers for its own statements: the commands run on the server that receives them, a replica included, and are not forwarded to the leader.

Syntax:

```
SHOW TRANSACTIONS [ 'id' [, 'id' ...] | $ids ]
  [ WHERE ... | YIELD ... [ORDER BY ...] [SKIP ...] [LIMIT ...] [WHERE ...] [RETURN
...] ]
  [ TERMINATE TRANSACTIONS ... ]

TERMINATE TRANSACTIONS 'id' [, 'id' ...] | $ids
  [ WHERE ... | YIELD ... [WHERE ...] [RETURN ...] ]
```

`$ids` is a parameter holding one id or a list of ids. The `WHERE` and `YIELD` tail works as for every other `SHOW` command, see Filtering the output of a SHOW command. A `SHOW TRANSACTIONS` can be followed by a `TERMINATE TRANSACTIONS` that names the ids by a column the `SHOW` yields, which stops the statements the `SHOW` rows select (see the last example).

Result columns of `SHOW TRANSACTIONS`:

A bare `SHOW TRANSACTIONS` returns Neo4j's default columns:

Column	Description
`database`	The database the statement runs on
`transactionId`	The statement's id, as `list queries` shows it
`currentQueryId`	The same id

Column	Description
`connectionId`	The client connection, when the protocol has one (a PostgreSQL backend process id, `bolt-<n>`, …), otherwise an empty string
`clientAddress`	The client's address, when known, otherwise an empty string
`username`	The user running the statement
`currentQuery`	The statement text, truncated and with credentials masked as in `list queries`
`startTime`	When the statement started, as an ISO-8601 string
`status`	`Running`, or `Terminating` once it was asked to stop
`elapsedTime`	How long it has been running, as a `Duration`

`YIELD` also reaches these columns:

Column	Description
`protocol`	How the statement reached the server: `http`, `postgres`, `bolt`, `grpc`, `mongodb`, `redis` or `gremlin`
`language`	The query language of the statement
`metaData`	A map; its `tag` is the label the client sent with the `X-ArcadeDB-Query-Tag` HTTP header
`sessionId`	The HTTP transaction session the statement runs in, if any
`statusDetails`	Who asked the statement to stop, once it is terminating

Result columns of `TERMINATE TRANSACTIONS`:

Column	Description
`transactionId`	The id that was asked for
`username`	The user running the statement
`message`	`Transaction terminated.`, or `Transaction not found.` when no such statement is running or it is another user's

Like Neo4j, `TERMINATE TRANSACTIONS` does not wait for the statement to stop: it asks it to stop and answers. The statement stops at its next check and its client receives the error of its protocol (for example `Neo.ClientError.Transaction.Terminated` over Bolt, SQLSTATE `57014` over PostgreSQL). Use `SHOW TRANSACTIONS`, or `terminate query` with its `wait`, to know when it is gone.

Examples:

```
// Every statement you may see, with Neo4j's default columns
SHOW TRANSACTIONS

// Some statements by id
SHOW TRANSACTIONS 'q118-5e1f0a9c', 'q121-5e1f0a9c'
SHOW TRANSACTIONS $ids

// The statements labelled by a benchmark run, with where they came from
SHOW TRANSACTIONS YIELD transactionId, username, protocol, language, metaData,
elapsedTime
WHERE metaData.tag = 'bench-run-42'

// Stop statements by id
TERMINATE TRANSACTIONS 'q118-5e1f0a9c', 'q121-5e1f0a9c'
TERMINATE TRANSACTIONS $ids YIELD transactionId, message

// Find and stop in one command every statement carrying a label
SHOW TRANSACTIONS YIELD transactionId AS txId, metaData WHERE metaData.tag = 'bench-
run-42'
TERMINATE TRANSACTIONS txId
YIELD transactionId, message
RETURN transactionId, message
```

15.3.38. Transaction Control (START TRANSACTION, COMMIT, ROLLBACK)

ISO/IEC 39075 (GQL) transaction control statements manage a transaction from within the query language, in addition to the driver/HTTP API. A transaction opened with `START TRANSACTION` stays open across subsequent commands and is finalized only by an explicit `COMMIT` or `ROLLBACK`. They mirror the SQL `BEGIN` / `COMMIT` / `ROLLBACK` statements.

Syntax:

```
START TRANSACTION [ISOLATION <READ_COMMITTED | REPEATABLE_READ>]
COMMIT
ROLLBACK
```

Examples:

```
START TRANSACTION
CREATE (p:Person {name: 'Alice'})
CREATE (p:Person {name: 'Bob'})
COMMIT

// Discard the work instead
START TRANSACTION
CREATE (p:Person {name: 'Charlie'})
```

```
ROLLBACK
```

The optional `ISOLATION` clause is an ArcadeDB extension — GQL does not define isolation levels for `START TRANSACTION`. It accepts `READ_COMMITTED` (the default) or `REPEATABLE_READ`, matching the SQL `BEGIN ISOLATION` statement.

To span multiple requests the transaction needs a stateful session. Over HTTP, pass a session id (the `arcadedb-session-id` header); over Bolt, the connection provides the session. Over stateless HTTP without a session each request runs in its own transaction.

`COMMIT` with no active transaction reports an error; `ROLLBACK` with no active transaction is a no-op. A second `START TRANSACTION` while one is already active opens a nested transaction (as with SQL `BEGIN`). The GQL access modes `READ ONLY` / `READ WRITE` are not implemented (ArcadeDB has no read-only transaction mode); `START TRANSACTION READ ONLY` reports a parse error.

15.3.39. Session Management (SESSION SET, SESSION RESET, SESSION CLOSE)

ISO/IEC 39075 (GQL) session management statements operate on the current server session:

- **SESSION SET** `$name = <value>` binds a named query parameter on the session. Subsequent commands in the same session see it as `$name` (a value supplied with the request wins on a name clash). The value may reference an earlier session parameter.
- **SESSION RESET** clears the session parameters.
- **SESSION CLOSE** releases the session.

Syntax:

```
SESSION SET $parameter = <value>
SESSION RESET
SESSION CLOSE
```

Examples:

```
SESSION SET $threshold = 21
MATCH (p:Person) WHERE p.age >= $threshold RETURN p.name
SESSION RESET
SESSION CLOSE
```

Session statements require a server session — over HTTP the `arcadedb-session-id` header, over Bolt the connection. In embedded mode they report an actionable

error. Session parameters are resolved by the OpenCypher engine, so they apply to both the `/command` and `/query` endpoints.

`SESSION CLOSE` is transport-specific: over HTTP it also rolls back the session transaction and invalidates the session id, whereas over Bolt it clears the session parameters and leaves the connection (and its transaction) live.

Session state is node-local and is not replicated across an HA cluster. In an HA setup without session affinity, a `SESSION SET` on one node followed by a query routed to another node will not see the parameter.

15.4. Expressions and Functions

This section describes the expressions and functions available in ArcadeDB's OpenCypher implementation.

15.4.1. Literals

Type	Syntax	Example
Integer	`123`, `-456`	`RETURN 42`
Float	`3.14`, `-2.5`	`RETURN 3.14159`
String	`'text'`, `"text"`	`RETURN 'Hello'`
Boolean	`true`, `false`	`RETURN true`
Null	`null`	`RETURN null`
List	`[1, 2, 3]`	`RETURN [1, 'a', true]`
Map	`{key: value}`	`RETURN {name: 'Alice', age: 30}`

15.4.2. Property Access

Access properties using dot notation:

```
MATCH (p:Person)
RETURN p.name, p.age, p.address.city
```

A property can also be read with the subscript form `p['name']`, through `properties(p)` (`properties(p).name`, `properties(p)['name']`), and across a `LINK` property (`a.link.name`, `a.link['name']`). *(Since v26.10.2)* `a.link['name']` on a `LINK` property reads through the link like `a.link.name`; before, it returned `null`.

Comparing a property declared in the schema:

(Since v26.10.2) A value written into a declared property is converted to the declared type, and the declaration also decides how a text is read. A property declared `STRING` stores the text of what is

written into it (writing `1` stores `'1'`), and a property declared with any other scalar type (`INTEGER`, `LONG`, `FLOAT`, `DOUBLE`, `DECIMAL`, `BOOLEAN`, `DATE`, `DATETIME`, ...) stores a text written into it as that type (writing `'5'` into an `INTEGER` stores `5`). To find it again, a pattern (`MATCH (n:Product {code: 1})`, `MERGE`) or a comparison against such a property compares by what the other operand would be stored as: `n.code = 1` finds the `'1'` that writing `1` stored, and `n.qty = '5'` finds the `5` that writing `'5'` stored, so a repeated `MERGE` does not create a duplicate. A property that is not declared keeps Cypher's strict typing, so `'1' = 1` is `false`.

Only the text form is converted: two values of the same kind keep Cypher's comparison, so `n.qty = 1.5` stays `false` for a stored `1`. A text the declared type cannot read (`'abc'` or `'5.7'` for an `INTEGER`) matches nothing.

The comparison applies to every spelling of the same property read: `n.code`, `n['code']`, `properties(n).code`, `properties(n)['code']`, a chained read across a link (`a.link.code`, `a.link['code']`), `IN` (`n.code IN [1, 2]`) and the simple `CASE n.code WHEN 1 THEN ⋯ END`. A value projected first is a plain value and keeps Cypher's strict typing:

```
// Product.code is declared STRING, Product.qty is declared INTEGER
CREATE (:Product {code: 1, qty: '5'})                                  // stores '1' and 5

MATCH (n:Product {qty: '5'}) RETURN n                                  // found

MATCH (n:Product) WHERE n['code'] = 1 RETURN n                         // found
MATCH (n:Product) WHERE n.code IN [1, 2] RETURN n                      // found
MATCH (n:Product) WITH n, n.code AS code WHERE code = 1 RETURN n       // not found: '1' =
1 is false
```

15.4.3. CASE Expression

Conditional logic in expressions.

Simple CASE:

```
MATCH (p:Person)
RETURN p.name,
       CASE
         WHEN p.age < 18 THEN 'minor'
         WHEN p.age < 65 THEN 'adult'
         ELSE 'senior'
       END AS ageGroup
```

Extended CASE:

```
MATCH (p:Person)
RETURN p.name,
       CASE p.status
         WHEN 'A' THEN 'Active'
```

```
        WHEN 'I' THEN 'Inactive'
        ELSE 'Unknown'
      END AS statusName
```

A `WHEN` can hold several comma-separated values, a list or map literal, or a comparison applied to the tested value:

```
RETURN CASE x WHEN 1, 2 THEN 'low' WHEN > 5 THEN 'high' WHEN IS NULL THEN 'none' ELSE
'mid' END
RETURN CASE n.tags WHEN [] THEN 'empty' WHEN ['a'] THEN 'just a' ELSE 'other' END
```

15.4.4. Arithmetic Expressions

Perform arithmetic operations on numeric values.

Operator	Description	Example
`+`	Addition	`RETURN 5 + 3` (returns 8)
`-`	Subtraction	`RETURN 10 - 4` (returns 6)
`*`	Multiplication	`RETURN 6 * 7` (returns 42)
`/`	Division	`RETURN 15 / 3` (returns 5.0)
`%`	Modulo	`RETURN 17 % 5` (returns 2)
`^`	Power	`RETURN 2 ^ 8` (returns 256.0)

Examples with properties:

```
// Double a person's age
MATCH (p:Person)
RETURN p.name, p.age * 2 AS doubleAge

// Calculate monthly salary
MATCH (p:Person)
RETURN p.name, p.salary / 12 AS monthlySalary

// Complex expression
MATCH (p:Person)
RETURN p.name, (p.age * 2) + 10 AS result
```

The `+` operator also works for string concatenation: `RETURN 'Hello' + ' ' + 'World'` returns `'Hello World'`.

15.4.5. Map Literals

Create inline map (object) structures.

Syntax: `{key1: value1, key2: value2, ⋯}`

```
// Simple map literal
RETURN {name: 'Alice', age: 30}

// Empty map
RETURN {}

// Map with expressions
MATCH (p:Person)
WHERE p.name = 'Alice'
RETURN {
  personName: p.name,
  doubled: p.age * 2,
  nextYearAge: p.age + 1
} AS info
```

15.4.6. List Comprehensions

Transform and filter lists using comprehension syntax.

Syntax: `[variable IN list WHERE condition | expression]`

- `variable` - iteration variable
- `list` - source list or range
- `WHERE condition` - optional filter (elements that don't match are excluded)
- `| expression` - optional mapping expression (transforms each element)

Basic transformation:

```
// Double each number
RETURN [x IN [1, 2, 3] | x * 2]
// Returns [2, 4, 6]

// Square numbers using range
RETURN [x IN range(1, 5) | x * x]
// Returns [1, 4, 9, 16, 25]
```

With filter:

```
// Filter and transform
RETURN [x IN [1, 2, 3, 4, 5] WHERE x > 2 | x * 10]
// Returns [30, 40, 50]

// Filter only (no transformation)
RETURN [x IN [1, 2, 3, 4, 5] WHERE x > 3]
```

```
// Returns [4, 5]
```

15.4.7. Map Projections

Extract specific properties from nodes or maps into a new map structure.

Syntax: `variable{.property1, .property2, key: expression, .*}`

- `.property` - include this property from the source
- `key: expression` - add a computed value with the given key
- `.*` - include all properties from the source

Property selection:

```
// Select specific properties
MATCH (p:Person)
WHERE p.name = 'Alice'
RETURN p{.name, .age} AS person
// Returns {name: 'Alice', age: 30}
```

With computed values:

```
// Add computed properties
MATCH (p:Person)
WHERE p.name = 'Alice'
RETURN p{.name, doubleAge: p.age * 2} AS person
// Returns {name: 'Alice', doubleAge: 60}
```

All properties:

```
// Include all properties
MATCH (p:Person)
WHERE p.name = 'Alice'
RETURN p{.*} AS person
// Returns {name: 'Alice', age: 30, salary: 50000}
```

With aggregates: a map projection that contains an aggregate groups by its own variable, one row per value of it.

```
MATCH (p:Person)-[:KNOWS]->(f)
RETURN p{.name, friends: count(f)} AS person
// Returns one row per p, e.g. {name: 'Alice', friends: 2} and {name: 'Bob', friends:
1}
```

15.4.8. Aggregation Functions

Aggregation functions process multiple values and return a single result.

count()

Returns the number of values or rows.

```
// Count nodes
MATCH (p:Person)
RETURN count(p)

// Count all rows
MATCH (p:Person)
RETURN count(*)

// Count non-null values
MATCH (p:Person)
RETURN count(p.email)
```

sum()

Returns the sum of numeric values.

```
MATCH (p:Person)
RETURN sum(p.salary)
```

avg()

Returns the average of numeric values.

```
MATCH (p:Person)
RETURN avg(p.age)
```

min()

Returns the minimum value.

```
MATCH (p:Person)
RETURN min(p.age) AS youngest
```

max()

Returns the maximum value.

```
MATCH (p:Person)
```

```
RETURN max(p.age) AS oldest
```

collect()

Collects values into a list.

```
// Collect all names
MATCH (p:Person)
RETURN collect(p.name) AS allNames

// Group and collect
MATCH (p:Person)-[:LIVES_IN]->(c:City)
RETURN c.name, collect(p.name) AS residents
```

15.4.9. String Functions

Every STRING-typed argument of these functions (not just the first one) is validated: passing a non-STRING value - for example a list or a number - raises a client error instead of silently converting it to text. If you need to pass a non-string value on purpose, convert it explicitly first with `toString()`, e.g. `split('a,1', toString(1))`.

toUpper()

Converts string to uppercase.

```
RETURN toUpper('hello')  // Returns 'HELLO'
```

toLower()

Converts string to lowercase.

```
RETURN toLower('HELLO')  // Returns 'hello'
```

trim()

Removes leading and trailing whitespace.

```
RETURN trim('  hello  ')  // Returns 'hello'
```

substring()

Extracts a substring.

Syntax: `substring(string, start [, length])`

```
RETURN substring('hello world', 0, 5)  // Returns 'hello'
RETURN substring('hello world', 6)      // Returns 'world'
```

replace()

Replaces occurrences of a substring.

Syntax: `replace(string, search, replacement)`

```
RETURN replace('hello', 'l', 'L')  // Returns 'heLLo'
```

split()

Splits a string into a list.

Syntax: `split(string, delimiter)`

```
RETURN split('a,b,c', ',')  // Returns ['a', 'b', 'c']
```

left()

Returns the leftmost characters.

Syntax: `left(string, length)`

```
RETURN left('hello', 3)  // Returns 'hel'
```

right()

Returns the rightmost characters.

Syntax: `right(string, length)`

```
RETURN right('hello', 3)  // Returns 'llo'
```

reverse()

Reverses a string.

```
RETURN reverse('hello')  // Returns 'olleh'
```

toString()

Converts a value to string.

```
RETURN toString(123)      // Returns '123'
RETURN toString(3.14)     // Returns '3.14'
RETURN toString(true)     // Returns 'true'
```

(Since v26.12.1.) A `localtime()`, `time()`, `localdatetime()` or `datetime()` is written as Neo4j 5 writes it: the seconds are always printed, and the fraction of a second has its trailing zeros removed (up to 9 digits). A date, an offset, a zone name and a duration are written as before.

```
RETURN toString(localtime('10:00'))                        // Returns '10:00:00'
RETURN toString(time('10:00Z'))                            // Returns '10:00:00Z'
RETURN toString(localdatetime('2020-01-01T10:00'))         // Returns '2020-01-
01T10:00:00'
RETURN toString(datetime('2020-01-01T10:00:00.500Z'))      // Returns '2020-01-
01T10:00:00.5Z'
```

Only the text made by `toString()` (and `toStringOrNull()`, `toStringList()` and `convert.toString()`) follows this rule. A temporal value that a query returns is still serialized as before, for example `"10:00"` for `localtime('10:00')` over HTTP.

15.4.10. Math Functions

abs()

Returns absolute value.

```
RETURN abs(-42)  // Returns 42
```

sqrt()

Returns square root.

```
RETURN sqrt(16)  // Returns 4.0
```

ceil()

Rounds up to nearest integer.

```
RETURN ceil(2.3)  // Returns 3.0
```

floor()

Rounds down to nearest integer.

```
RETURN floor(2.7)  // Returns 2.0
```

round()

Rounds to nearest integer.

```
RETURN round(2.5)  // Returns 3.0
RETURN round(2.4)  // Returns 2.0
```

rand()

Returns random float between 0 and 1.

```
RETURN rand()  // Returns e.g. 0.7324...
```

In an inline property map, `rand()` is drawn once per input row, and that one value both finds the node and checks it: `UNWIND range(1, 2000) AS j MATCH (a:N {k: toInteger(rand() * 10)})` over keys 0 to 9 matches one node per row. In `WHERE a.k = toInteger(rand() * 10)` the expression is drawn again for every candidate node, as the predicate is written, so an index on `k` is not used to answer it. *(Since v26.12.1, issue #12179: a value drawn to look the node up was drawn again to check it, so the match kept about one row in ten.)*

15.4.11. List Functions

size()

Returns the size of a list or string. A string is measured in Unicode characters, so an emoji counts as one (`left()`, `right()`, `substring()` and `char_length()` count the same way).

```
RETURN size([1, 2, 3])      // Returns 3
RETURN size('hello')        // Returns 5
```

head()

Returns the first element of a list.

```
RETURN head([1, 2, 3])  // Returns 1
```

tail()

Returns all but the first element.

```
RETURN tail([1, 2, 3])  // Returns [2, 3]
```

last()

Returns the last element of a list.

```
RETURN last([1, 2, 3])  // Returns 3
```

range()

Creates a list of integers.

Syntax: `range(start, end [, step])`

```
RETURN range(1, 5)     // Returns [1, 2, 3, 4, 5]
RETURN range(0, 10, 2) // Returns [0, 2, 4, 6, 8, 10]
```

reverse() (list)

Reverses a list.

```
RETURN reverse([1, 2, 3])  // Returns [3, 2, 1]
```

15.4.12. Node and Relationship Functions

id()

Returns the internal ID of a node or relationship as a numeric (`Long`) value, matching the Neo4j semantics where `id()` returns an `INTEGER`. The two components of an ArcadeDB RID (`bucketId`, `position`) are packed into a single non-negative number so it can be used in numeric predicates such as `WHERE id(n) >= 0`.

```
MATCH (p:Person)
RETURN id(p)  // Returns e.g. 786432
```

The packing uses 63 of the 64 bits of the `Long` (the sign bit is kept clear so the value is never negative): the upper bits hold the `bucketId` and the lower bits the record `position`. The split is controlled by the global setting `arcadedb.opencypher.idBucketBits` (see Settings). The default of `16` reserves 16 bits for the bucketId (up to 65536 buckets) and 47 bits for the position (~1.4e14 records

per bucket), which covers the vast majority of use cases:

- Increase it (e.g. `20`) for databases that allocate a very large number of buckets across many types.
- Decrease it for buckets that hold an extremely high number of records.

```
# Reserve 20 bits for the bucketId (up to ~1M buckets)
java ... -Darcadedb.opencypher.idBucketBits=20
```

To put the position field in perspective with the default `16`: it reserves 47 bits, enough to address `2^47` ≈ 1.4e14 positions per bucket. A single bucket is internally limited to ~2.1 billion pages (the page index is a 32-bit integer), so with a standard 64KB page holding on average ~500 records that is about `2.1e9 x 500` ≈ **1 trillion records (~137 TB) in one bucket** - and the 47-bit position field covers that roughly 30x over. In other words, the numeric `id()` never runs out of range before the bucket is physically full, so for almost every database the only setting worth tuning is the bucketId width (the bucket **count**), not the position width.

`id()` fails with an error if a RID cannot be packed with the configured split (for example a `bucketId` beyond the reserved bits), so an out-of-range value is reported instead of silently colliding with another id. Because the value is computed from the split, changing `arcadedb.opencypher.idBucketBits` alters the numeric output: encode and decode must use the same setting. For a stable, opaque identifier that does not depend on this setting, use `elementId()` instead.

`id()` of a lightweight edge is `null`: a lightweight edge has no record, so there is no position to pack (*since v26.10.2*; earlier versions raised an error). Use `elementId()` instead, which names the edge (*since v26.12.1*, see Lightweight Edges).

From SQL, the numeric id can be resolved back to a record in O(1) with the `cypherRID()` function, e.g. `SELECT FROM cypherRID(:id)` (the inverse of `id()` / `.asCypherRID()`).

labels()

Returns the labels of a node as a list. A node created with no label at all (`CREATE (n {name: 'x'})`) returns an empty list, exactly like Neo4j — including when the label a node carries happens to be literally `V` or `Vertex`, which are ordinary, writable labels like any other and are not treated specially.

```
MATCH (p:Person)
RETURN labels(p)  // Returns ['Person']

CREATE (n {name: 'x'})
RETURN labels(n)  // Returns []

CREATE (n:V {name: 'y'})
RETURN labels(n)  // Returns ['V']
```

A database written by an earlier version can hold unlabelled nodes under the type

name `V` (or `Vertex`), which that version used internally for "no labels". Those vertices report `labels(n)` as `['V']`/`['Vertex']` instead of `[]`.

There is no automatic per-record migration, because the two cases are indistinguishable in such a database: an existing `V`-typed vertex may be one that was genuinely created unlabelled, or one that was deliberately labeled `:V`. If a database's entire `V` (or `Vertex`) type was only ever used for unlabelled nodes, its schema type can be renamed so that the whole type reports no labels again; consult ArcadeDB support before doing this on a database where the distinction is not already known.

type()

Returns the type of a relationship. Calling it on a node is always an error.

```
MATCH (a)-[r]->(b)
RETURN type(r)  // Returns e.g. 'KNOWS'
```

This check (and the equivalent one for `labels()`, `size()` and `length()`) is applied more precisely when the same variable name is reused later in the query with `WITH`. A query that called `type()` on a node and relied on a later `WITH` clause to mask the mistake now correctly reports the error instead of silently passing.

keys()

Returns the property keys of a node or relationship.

```
MATCH (p:Person {name: 'Alice', age: 30})
RETURN keys(p)  // Returns ['name', 'age']
```

properties()

Returns all properties as a map.

```
MATCH (p:Person {name: 'Alice', age: 30})
RETURN properties(p)  // Returns {name: 'Alice', age: 30}
```

startNode()

Returns the start node of a relationship.

```
MATCH (a)-[r:KNOWS]->(b)
RETURN startNode(r)  // Returns node a
```

endNode()

Returns the end node of a relationship.

```
MATCH (a)-[r:KNOWS]->(b)
RETURN endNode(r)  // Returns node b
```

15.4.13. Path Functions

The path functions read any path: one bound by a `MATCH` pattern or `shortestPath()`, the `path` a procedure yields (see reference/graph-algorithms/path-finding.pdf), or one a `path.*` function returns. `null` returns `null`, and any other value is a type error rather than an empty answer (`length()` also accepts a string, and returns its length in Unicode characters).

length()

Returns the length of a path (number of relationships).

```
MATCH path = (a:Person)-[:KNOWS*]->(b:Person)
RETURN length(path)
```

`length()` also accepts a string and counts it as `size()` and `char_length()` do: in Unicode characters, so a character outside the Basic Multilingual Plane (an emoji, a mathematical letter such as 𝐗) counts as one.

```
RETURN length('𝐗')   // Returns 2
```

Up to v26.10.2, `length()` of a string counted UTF-16 code units, so the example above answered 3 while `size('𝐗')` answered 2. From v26.11.1 the three agree. Strings made only of characters of the Basic Multilingual Plane (letters, digits and most scripts) are not affected.

nodes()

Returns the nodes in a path as a list.

```
MATCH path = (a:Person)-[:KNOWS*]->(b:Person)
RETURN nodes(path)
```

relationships()

Returns the relationships in a path as a list.

```
MATCH path = (a:Person)-[:KNOWS*]->(b:Person)
RETURN relationships(path)
```

15.4.14. Type Conversion Functions

toInteger()

Converts to integer.

```
RETURN toInteger('42')      // Returns 42
RETURN toInteger(3.7)       // Returns 3
```

toFloat()

Converts to float.

```
RETURN toFloat('3.14')      // Returns 3.14
RETURN toFloat(42)          // Returns 42.0
```

toBoolean()

Converts to boolean.

```
RETURN toBoolean('true')    // Returns true
RETURN toBoolean(1)         // Returns true
RETURN toBoolean(0)         // Returns false
```

15.4.15. SQL Function Bridge

ArcadeDB's OpenCypher implementation includes a bridge to the ArcadeDB SQL functions. This provides access to 100+ additional functions. Some of them (such as `sqrt()`, `date()` and `distance()`) can be called by their bare name; others (such as `pow()`, `uuid()`, `ifnull()` and `sysdate()`) are not found unless you prefix them with `sql.`, e.g. `sql.pow(2, 8)`. When in doubt, use the `sql.` prefix.

Usage:

Call SQL functions directly in Cypher expressions:

```
// Date functions
MATCH (p:Person)
RETURN p.name, date() AS today
```

```
// Math functions
RETURN sql.pow(2, 8) AS result  // Returns 256

// String functions
MATCH (p:Person)
RETURN normalize(p.name) AS normalizedName

// Geometric functions
RETURN distance(point(0, 0), point(3, 4)) AS dist  // Returns ~555811.94 (meters)
```

`point(0, 0)` creates a WGS-84 point, so `distance()` returns a geodesic distance in meters, not the cartesian distance of 5.0.

The Cypher `point()` function returns a map with the coordinates, the `crs` name and the `srid`, as Neo4j does:

Call	crs	srid
`point({x: 1, y: 2})`	cartesian	7203
`point({x: 1, y: 2, z: 3})`	cartesian-3d	9157
`point({longitude: 1, latitude: 2})`	wgs-84	4326
`point({longitude: 1, latitude: 2, height: 3})`	wgs-84-3d	4979

`crs` and `srid` can be given explicitly and are derived from each other. A `crs` name is accepted in any letter case and returned in the spelling above. A pair that disagrees, or 2D/3D coordinates that do not match the `crs`/`srid`, raise an error.

`x`/`y` (and `z`) with a geographic `srid` or `crs` (4326, 4979, `wgs-84`, `wgs-84-3d`) build the same point as `longitude`/`latitude` (and `height`): `point({x: 1, y: 2, srid: 4326}).longitude` is `1.0`. A map that has both `x`/`y` and `longitude`/`latitude` is built from `x`/`y`, as in Neo4j, and `z` wins over `height`. A coordinate must be finite: `Infinity` or `NaN` raises "Cannot create a point with non-finite coordinate values". A map with only one coordinate of its pair, such as `point({x: 1})` or `point({latitude: 1})`, raises "A map with keys [x] is not describing a valid point", while a coordinate given as `null` (`point({x: null, y: 2})`) still returns `null`. Unlike Neo4j, an unknown `crs` name or `srid` is kept as given.

Since v26.12.1. Before, `point({x: 1, y: 2, srid: 4326})` had a `null` `longitude` and `latitude`, a map with both spellings built the WGS-84 point from `longitude`/`latitude`, non-finite coordinates were accepted and an incomplete map returned `null` (issue #12276).

`ORDER BY`, `min()` and `max()` order points as Neo4j does: by coordinate reference system (srid 4326, 4979, 7203, 9157), then by x (`longitude`), y (`latitude`) and z (`height`). A point sorts after an ordinary map. Points are not comparable with `<` and `>` (the answer is `null`); `⇐` and `>=` are `true` for two equal points and `null` otherwise.

Since v26.12.1. Before, points were ordered as maps: a WGS-84 point by `latitude` first, a WGS-84-3D point by `height` first, and points of different systems by their

number of keys. `p ⇐ p` and `p >= p` were `null` (issue #12274).

`point.distance(p1, p2)` returns the Euclidean distance between two Cartesian points and the distance in meters between two WGS-84 points, on a sphere of radius 6,378,140 m as Neo4j does. For two WGS-84-3D points it includes the `height`: the surface arc is taken at the mean height of the two points and the height difference is added as the other side of a right triangle, so two points 1,000 m apart vertically are 1,000 m apart. `point.withinBBox(p, lowerLeft, upperRight)` also checks `z` (`height`) when all three points are 3D. Both functions return `null` when a 2D point is mixed with a 3D point, or a geographic point with a Cartesian one.

Since v26.12.1. Before, both functions ignored the `height` of WGS-84-3D points and `point.withinBBox()` ignored `z` too, a mix of 2D and 3D points gave a result instead of `null`, and `point.distance()` used an Earth radius of 6,371,000 m, so every geographic distance is now about 0.11% longer. The SQL `geo.*` functions and the bare `distance()` shown above keep the 6,371 km radius (issue #12273).

Up to 26.10.2 the `crs` names were `WGS-84`, `WGS-84-3D` and `cartesian-3D`. They are now the lower-case names Neo4j uses (`wgs-84`, `wgs-84-3d`, `cartesian-3d`), so a test like `p.crs ENDS WITH '-3d'` works as it does on Neo4j. A point stored in a property before the upgrade keeps the old spelling: `point.distance()`, `point.withinBBox()` and Bolt still recognise it, but a case-sensitive comparison of its `crs` with a new point's does not match, so compare `toLower(p.crs)` or the `srid` when old points can be involved.

For a complete list of available SQL functions, see the SQL Functions reference.

15.4.16. Extended Functions (APOC Compatible)

ArcadeDB provides a comprehensive set of extended functions that are fully compatible with Neo4j's APOC library. These functions work in both Cypher and SQL queries.

APOC Prefix Compatibility

ArcadeDB automatically supports the `apoc.` prefix. Your existing APOC queries work without modification:

```
// Neo4j/APOC style (works in ArcadeDB)
RETURN apoc.text.join(["a", "b", "c"], ",") AS result

// ArcadeDB native style (also works)
RETURN text.join(["a", "b", "c"], ",") AS result
```

Available Function Namespaces

Namespace	Description
`agg.*`	Aggregation functions (median, statistics, etc.)
`convert.*`	Type conversion (toJson, fromJsonMap, etc.)
`create.*`	Creation functions (uuid, virtual nodes/relationships)
`date.*`	Date/time operations (format, parse, add, etc.)
`map.*`	Map operations (merge, flatten, groupBy, etc.)
`math.*`	Mathematical functions (sigmoid, tanh, etc.)
`node.*`	Node operations (degree, labels, etc.)
`path.*`	Path operations (create, combine, slice, expand)
`rel.*`	Relationship operations (type, startNode, etc.)
`text.*`	String manipulation (join, split, replace, camelCase, etc.)
`util.*`	Utility functions (md5, sha256, compress, etc.)
`vector.*`	Vector/embedding operations (similarity, distance, quantization, neighbors)

Available Procedures

Procedure	Description
`merge.relationship`	Create or match relationships
`merge.node`	Create or match nodes
`algo.dijkstra`	Weighted shortest path
`algo.astar`	A* pathfinding with heuristics
`algo.allsimplepaths`	Find all simple paths
`path.expand`	Expand paths from a node
`path.subgraphnodes`	Get all reachable nodes
`meta.graph`	Schema as virtual graph
`meta.schema`	Detailed schema information
`meta.stats`	Database statistics

Examples:

```
// Text functions
RETURN text.camelCase("hello world") AS result
// Returns: "helloWorld"

// Map functions
RETURN map.merge({a: 1}, {b: 2, a: 3}) AS result
// Returns: {a: 3, b: 2}
```

```
// JSON conversion
RETURN convert.toJson({name: "Alice", age: 30}) AS json
// Returns: '{"name":"Alice","age":30}'

// Hashing
RETURN util.sha256("password") AS hash

// Statistics
RETURN agg.statistics([1, 2, 3, 4, 5]) AS stats
// Returns: {count: 5, min: 1.0, max: 5.0, sum: 15.0, mean: 3.0, ...}

// Graph algorithms
MATCH (a:City {name: 'NYC'}), (b:City {name: 'LA'})
CALL algo.dijkstra(a, b, 'ROAD', 'distance') YIELD path, weight
RETURN path, weight
```

For the complete function reference, see reference/extended-functions.pdf.

15.4.17. Operators

Comparison Operators

Operator	Description	Example
=	Equal	WHERE n.age = 30
<> or !=	Not equal	WHERE n.age <> 30
<	Less than	WHERE n.age < 30
>	Greater than	WHERE n.age > 30
<=	Less than or equal	WHERE n.age <= 30
>=	Greater than or equal	WHERE n.age >= 30

Logical Operators

Operator	Description	Example
AND	Logical AND	WHERE n.age > 20 AND n.age < 40
OR	Logical OR	WHERE n.city = 'NYC' OR n.city = 'LA'
NOT	Logical NOT	WHERE NOT n.active = false

String Operators

Operator	Description	Example
`STARTS WITH`	String prefix match. Uses an ordered index on the property, except a case insensitive one	`WHERE n.name STARTS WITH 'A'`
`ENDS WITH`	String suffix match	`WHERE n.email ENDS WITH '@gmail.com'`
`CONTAINS`	Substring match	`WHERE n.name CONTAINS 'son'`
`=~`	Regular expression	`WHERE n.phone =~ '[0-9]{3}-[0-9]{4}'`

`MATCH (n:Label) RETURN min(n.prop)` (and `max`) reads one end of the index when `prop` has an ordered index on `Label` and the query has no other clause. A `WHERE` that is only a range of the same property (`WHERE n.prop > 500000`, `<`, `>=`, ⇐) reads the first matching entry of the range instead of the whole range. SQL does the same for `SELECT min(prop) FROM Type WHERE prop > 500000`. A range bound that is `null` matches nothing, with or without an index.

List Operators

Operator	Description	Example
`IN`	Element in list	`WHERE n.name IN ['Alice', 'Bob']`
`+`	List concatenation	`RETURN [1,2] + [3,4]` (returns [1,2,3,4])

Null Operators

Operator	Description	Example
`IS NULL`	Check for null	`WHERE n.email IS NULL`
`IS NOT NULL`	Check for not null	`WHERE n.email IS NOT NULL`

15.4.18. Parameters

Parameters allow you to pass values into queries without string concatenation, providing security and performance benefits.

Syntax: `$parameterName`

```
MATCH (p:Person)
WHERE p.name = $name AND p.age >= $minAge
RETURN p
```

Java Example:

```
Map<String, Object> params = Map.of(
    "name", "Alice",
    "minAge", 25
);
ResultSet result = database.query("opencypher",
    "MATCH (p:Person) WHERE p.name = $name AND p.age >= $minAge RETURN p",
    params);
```

Parameters are particularly useful for:

- Preventing injection attacks
- Query plan caching (same plan reused for different parameter values)
- Handling special characters in values

15.5. Cypher Compatibility Matrix

This section provides a comprehensive overview of Cypher language features and their support status in ArcadeDB's native OpenCypher implementation.

15.5.1. OpenCypher TCK Compliance

ArcadeDB's OpenCypher implementation has been validated against the official OpenCypher Technology Compatibility Kit (TCK) version 9, developed by Neo4j and released under the Apache License 2.0.

Metric	Value	Status
TCK Pass Rate	**97.8%**	□ Production Ready
Total Scenarios	3,897	Official OpenCypher v9 test suite
Passed	3,813	Full compatibility with core features
Failed	0	All executable tests pass
Skipped	84	Documented limitations (2.2%)

Verify TCK Compliance Yourself

To run the OpenCypher TCK tests on your system:

```
mvn test -pl engine -am -Dsurefire.includes
='**/OpenCypherTCKSuite.java' -Dsurefire.failIfNoSpecifiedTests=false
```

The TCK test suite is located at: `engine/src/test/resources/opencypher/tck/features/`

For more information about the OpenCypher TCK, visit: https://github.com/opencypher/openCypher/tree/master/tck

15.5.2. Known Limitations

The 84 skipped scenarios (2.2%) represent either architectural design choices or edge cases that rarely occur in production use. The following limitations affect 84 TCK scenarios (2.2%) and represent either design choices or rare edge cases.

15.5.3. 1. Temporal Range Limitation (Java Platform Constraint)

TCK Impact: 4 scenarios (temporal sorting with extreme date ranges)

Reason: ArcadeDB uses Java's `LocalDateTime` with nanosecond precision stored as a 64-bit long, limiting the date range to approximately 1677-2262.

Examples that don't work:

```
// Year 0001 with nanoseconds - exceeds range
CREATE (:Event {datetime: localdatetime({year: 1, month: 1, day: 1,
                                         hour: 1, minute: 1, second: 1,
                                         nanosecond: 1})})

// Year 9999 with nanoseconds - exceeds range
CREATE (:FutureEvent {datetime: localdatetime({year: 9999, month: 9, day: 9,
                                               hour: 9, minute: 59, second: 59,
                                               nanosecond: 999999999})})
```

Workaround: Use dates within the 1677-2262 range (covers virtually all real-world data), or store historical dates as strings.

15.5.4. 2. Case-Insensitive Identifiers (By Design)

TCK Impact: ~40 scenarios (case-sensitive type/property names)

Reason: ArcadeDB follows SQL conventions where identifiers are case-insensitive, providing consistency across query languages.

Examples that don't work:

```
// These create a collision - T2 and t2 are considered the same type
CREATE ()-[:T2]->()
CREATE ()-[:t2]->()  // ❌ Conflicts with :T2
```

Workaround: Use distinct type names (e.g., `:Type2` and `:Type2Alt`).

15.5.5. 3. Cross-Type Comparison Strictness (Semantic Edge Case)

TCK Impact: ~20 scenarios (incompatible type comparisons)

Reason: Comparing fundamentally different types (nodes vs strings, paths vs primitives) in

production code typically indicates a query error.

Affected: Strict null-return semantics for cross-type comparisons.

15.5.6. 4. User Functions Support

TCK Impact: 0 scenarios (extension feature)

ArcadeDB enhances OpenCypher with user-defined functions in 4 languages:

```
// Define in SQL
DEFINE FUNCTION math.sum "SELECT :a + :b" PARAMETERS [a,b] LANGUAGE sql

// Define in JavaScript
DEFINE FUNCTION js.greet "return 'Hello ' + name" PARAMETERS [name] LANGUAGE js

// Define in OpenCypher
DEFINE FUNCTION cypher.double "RETURN $x * 2" PARAMETERS [x] LANGUAGE opencypher

// Call from any query language
RETURN math.sum(3, 5)  // Works in Cypher, SQL, Gremlin, etc.
```

See User Functions for complete documentation.

15.5.7. 6. Map-Valued Properties (Multi-Model Boundary)

A property cannot hold a map in openCypher, the same rule Neo4j applies ("Property values can only be of primitive types or arrays thereof"). `CREATE`, `MERGE` and `SET` all refuse one, whether it is written as a literal or bound as a parameter:

```
CREATE (n:R {config: {debug: true}})           // refused
MATCH (n:R) SET n += $p                        // refused when $p holds a map value
```

ArcadeDB's SQL is not bound by that rule, because `MAP` is a first-class schema type. So a record written through SQL can carry a map property that openCypher can **read** but never **write**:

```
-- SQL: allowed
CREATE VERTEX R SET id = 1, config = {'debug': true}
```

```
// Cypher: all of these work
MATCH (n:R) RETURN n.config                    // returns the map
MATCH (n:R) RETURN n.config.debug              // returns true
MATCH (n:R) WHERE n.config.debug RETURN n
MATCH (n:R) SET n.other = 1                    // untouched map is left alone

// but copying it is refused, because that would make openCypher store one
```

```
MATCH (n:R), (t:T) SET t = n
MATCH (n:R), (t:T) SET t.config2 = n.config
```

The refusal names the property and where the value came from, so the two copy forms above are diagnosable without reading the server log:

```
TypeError: InvalidPropertyType - Property values can only be of primitive types or
arrays thereof. Encountered a
map [debug] assigned to property 'config', copied from record #1:0. ArcadeDB SQL can
store a map in a property and
openCypher can not, so such a property can be read and returned but never copied into
another one.
```

Over Bolt it is reported as `Neo.ClientError.Statement.TypeError`, Neo4j's own code for this refusal, and over HTTP as `400`. To move such a property through openCypher, flatten it into scalar properties, or do the write in SQL.

15.5.8. The Variable of a Variable-Length Relationship Is a List

In `MATCH (a)-[r:R*1..3]→(b)` the variable `r` holds the **list** of relationships the path went through, not one relationship, so `r.weight` reads no property. As in Neo4j, the statement is refused before it runs, whatever it would match, and the kind follows the variable through a `WITH`:

```
TypeError: Type mismatch: expected Map, Node, Relationship, Point, Duration, Date,
Time, LocalTime, LocalDateTime or
DateTime but was List<Relationship>. 'r' is a variable-length relationship and holds a
list of relationships, so
'r.weight' reads no property: read it per relationship, e.g. [rel IN r | rel.weight]
```

Read the property per relationship instead: `[rel IN r | rel.weight]`, `UNWIND r AS rel RETURN rel.weight`, `r[0].weight`, or `all(rel IN r WHERE rel.weight > 1)`. The inline predicate of the pattern itself, `[r:R* WHERE r.weight > 1]`, tests one relationship at a time, so there `r.weight` is valid.

Over HTTP the refusal is a `400`. A type error that can only be found while the query runs, such as `UNWIND [[1, 2]] AS l RETURN l.x`, is a `400` too, with `com.arcadedb.exception.TypeMismatchException` as the reported exception, and `Neo.ClientError.Statement.TypeError` over Bolt. Before v26.11.1 both answered `500`.

15.5.9. The WHERE of a WITH

The `WHERE` of a `WITH` filters the rows the `WITH` produces. After a plain `WITH` it sees the projected names and the variables that came in. After an aggregating or `DISTINCT WITH` the incoming rows are gone, so, as in Neo4j, it sees only the projected columns, and an expression that repeats a grouping key reads that key's column:

```
MATCH (n:Node) WITH n.b AS v, count(*) AS c WHERE n.b IS NOT NULL RETURN v, c  // n.b
```

```
reads the column v
MATCH (n:Node) WITH n.b AS v, count(*) AS c WHERE n.id > 1 RETURN v, c        //
refused: n is not defined here
```

Since v26.12.1. Before, a name the `WHERE` could not see evaluated to `null`, which silently filtered every row out.

15.5.10. Invalid Inputs Are Refused

(Since v26.12.1.) As in Neo4j, these inputs raise an error instead of being accepted silently:

- an invalid regular expression on the right of `=~`, such as `'a' =~ '['`, raises `Invalid Regex` (it evaluated to `false`);
- `DELETE` of a value that is not a node, a relationship or a path, such as `WITH 1 AS x DELETE x`, raises a type mismatch (it did nothing);
- `YIELD` of a column the procedure does not produce raises `Unknown procedure output`, with the list of the columns it produces (it returned `null`).

Over Bolt they are reported as `Neo.ClientError.Statement.SemanticError`, `SyntaxError` and `TypeError` respectively.

15.5.11. A Pattern Expression Is a Boolean Test

A pattern expression such as `(p)-->()` tests whether the pattern exists, so in a `WHERE` it can only be used where a boolean is expected: as a predicate, as an operand of `NOT` / `AND` / `OR` / `XOR`, in a `CASE WHEN` condition, in the `WHERE` of a list predicate (`any()`, `all()`, `none()`, `single()`), of a list comprehension or of a pattern comprehension, and as the argument of `exists()`. Anywhere else it is refused with a syntax error (HTTP 400), as Neo4j 5 does and as `RETURN` already did: inside `size()`, `toString()` or `coalesce()`, and as an operand of a comparison, `IS NULL`, `IN` or arithmetic. To count the matches, use a `COUNT { }` subquery or a pattern comprehension:

```
MATCH (p:Person) WHERE size((p)-->()) > 5 RETURN p          // refused: use COUNT { }
instead
MATCH (p:Person) WHERE (p)-->() = true RETURN p             // refused
MATCH (p:Person) WHERE COUNT { (p)-->() } > 5 RETURN p      // more than 5 outgoing
relationships
MATCH (p:Person) WHERE size([(p)-->(q) | q]) > 5 RETURN p   // the same, with a
pattern comprehension
MATCH (p:Person) WHERE NOT (p)-->() RETURN p                // a boolean position:
fine
```

Since v26.12.1. Before, these statements ran without an error and gave a wrong answer: `sizep)-->(` was `null` for every node, so `WHERE sizep)-->( > 5` matched nothing. The same release fixes a `CASE WHEN (p)-->() ⋯` and a list predicate with a pattern in its `WHERE`, which were read as the bare pattern. A pattern expression in `UNWIND`, `ORDER BY`, `SKIP`, `LIMIT`, `FOREACH`, `DELETE` and `CALL` arguments is still read as a

> boolean and not refused (issue #12657).

15.5.12. Not Implemented Features

The following Cypher features are not currently implemented:

- **Index hints**: `USING INDEX n:Person(name)` - Query planning is automatic via cost-based optimizer

15.5.13. Cypher 25 / GQL Strictness

Following ISO/IEC 39075 (GQL) and Cypher 25, ArcadeDB rejects removed legacy syntax with an actionable error that points at the supported replacement:

- **`PERIODIC COMMIT` is removed** — use `CALL { … } IN TRANSACTIONS [OF n ROWS]` for batched writes. `USING PERIODIC COMMIT` reports an error with this hint.
- **Legacy `{param}` parameters are removed** — use `$param`. The old curly-brace parameter form reports an error with this hint (map projections such as `n{.name}` and quantified path quantifiers such as `{1,5}` are unaffected).
- **Ambiguous aggregation grouping is rejected** — a `RETURN` (or `WITH`) that mixes an aggregating expression with a non-grouping reference raises `AmbiguousAggregationExpression`. Aggregations are not allowed in `WHERE`.

15.5.14. Three-Valued Logic (UNKNOWN)

ArcadeDB follows the GQL three-valued logic: the `UNKNOWN` truth value is represented as `null` in boolean context (no separate `UNKNOWN` value type). `AND` / `OR` / `NOT` / `XOR` propagate `null`, comparisons against `null` evaluate to `null`, and a `null` condition in `WHERE` is treated as non-matching, while `RETURN` preserves the `null`.

```
RETURN true AND null AS a    // null
RETURN false AND null AS b   // false
RETURN true OR null AS c     // true
RETURN null = null AS d      // null
```

A predicate must be a `BOOLEAN` (or `null`), wherever it is written: `WHERE`, the operands of `AND` / `OR` / `XOR` / `NOT`, `CASE WHEN`, and the `WHERE` of a list predicate or comprehension. As in Neo4j, a `LIST` is the one exception, read as `true` when it is not empty, and any other value (`INTEGER`, `FLOAT`, `STRING`, `MAP`, a node, …) is a type error (`TypeError: Type mismatch: expected Boolean but was INTEGER`, a client error on every protocol). A literal operand (`RETURN 1 OR false`, `WHERE false AND 1`) is refused before the query runs, wherever it sits. `AND` and `OR` still skip an operand once the other one decides the result, so a skipped property is not checked: `WHERE true OR n.v` matches every node, as in Neo4j.

```
MATCH (n:Node) WHERE n.active RETURN n   // a BOOLEAN property: fine
MATCH (n:Node) WHERE n.tags RETURN n     // a LIST property: the nodes with at least
one tag
MATCH (n:Node) WHERE n.v RETURN n        // an INTEGER property: type error
```

```
MATCH (n:Node) WHERE n.v <> 0 RETURN n   // what the previous line usually means
```

Before 26.12.1 a non-BOOLEAN value was not refused, and its truth value depended on where it was written: any non-null value was `true` under AND / OR / NOT / XOR and in CASE WHEN, while WHERE read every non-BOOLEAN value, a non-empty list included, as `false`. A query relying on either reading now fails with the type error above; compare the value explicitly instead (`n.v <> 0`, `size(n.name) > 0`). The same release fixes `COUNT { ⋯ } > 0` written as an operand of AND / OR without parentheses, which was read as the bare count, so a count of zero passed as `true`.

The list predicates `all()`, `any()`, `none()` and `single()` follow the same rule: if the inner condition is `null` for one or more elements and no element decides the result outright (a `false` for `all()`, a `true` for `any()`/`none()`), the predicate itself evaluates to `null` rather than `true`/`false`. A definite mismatch always wins over an unknown one, so a single element that decides the result short-circuits regardless of any `null`s elsewhere in the list.

```
WITH [1, 2, null] AS list
RETURN any(x IN list WHERE x > 5) AS a          // null: no match, but one candidate
was unknown
RETURN none(x IN list WHERE x > 5) AS b         // null: same reasoning
RETURN all(x IN list WHERE x IS NOT NULL) AS c  // false: a definite mismatch always
wins over unknown
```

Evaluation order is not specified for AND / OR, so ArcadeDB evaluates the operand that cannot run a subquery (`COUNT { }`, `EXISTS { }`, `COLLECT { }`, a pattern predicate) first, and skips the subquery when the other operand already decides the result. Do not rely on left-to-right order to guard an operand that can raise an error. `coalesce()` stops at its first non-null argument: later arguments are not evaluated.

15.5.15. Zoned DateTime Equality and Ordering

Two DATETIME values are ordered by the instant they denote; at the same instant, by their offset from west to east, then by time zone name. Two values are equal only when they denote the same instant in the same time zone, so `datetime('2026-01-01T12:00:00+01:00') = datetime('2026-01-01T11:00:00Z')` is `false`, and DISTINCT and grouping keep the two apart as well. Compare `d.epochMillis` (or test with ⇐ and >=) to check for the same instant regardless of zone. A `java.util.Date` or `java.time.Instant` query parameter carries no time zone and is equal to any DATETIME at its instant.

15.5.16. Current Date and Time in a Query

As in Neo4j, `datetime()`, `date()`, `time()`, `localdatetime()` and `localtime()` without a value return the same instant for the whole statement, however many rows it produces, and so does the time zone form, such as `datetime('UTC')` or `date('Europe/Rome')`. Use `datetime.realtime()` (and the other `.realtime()` functions) to read the clock again at each call.

Since v26.11.1 for the time zone form. Before, `datetime('UTC')` and the same form

> of the other functions read the clock at each call, so the value changed from row to row.

15.5.17. Comparing and Ordering Temporal Values

Values of two different temporal types are never equal and are not comparable, as in Neo4j: `=` is `false`, `<>` is `true`, and `<`, `⇐`, `>`, `>=` are `null`. That holds for a `localdatetime()` against a `datetime()` too, so `localdatetime('2024-01-01T12:00') = datetime('2024-01-01T12:00Z')` is `false`.

The one exception is a `DATETIME` property read from the database. ArcadeDB stores a `DATETIME` as an instant and reads it back in Cypher as a local datetime in UTC, so against a zoned `datetime()` it compares as that instant: `MATCH (e:Event) WHERE e.createdAt > datetime('2024-01-01T00:00Z')` filters on the stored instant, whether the bound is a literal or a parameter.

Durations are equal when every component is (`duration('P1D') <> duration('PT24H')`), and are not comparable, since a month or a day has no fixed length: `duration('P1D') < duration('P2D')` is `null`, and `⇐` / `>=` are `true` only for two equal durations. A range filter on a duration therefore keeps no row; filter on a number instead, such as the average length in seconds, `d.months * 2629746 + d.days * 86400 + d.seconds`, or a single component when the durations hold nothing else (`d.days` for durations of whole days).

`ORDER BY`, `min()` and `max()` order every value, temporal ones included:

- values of different temporal types by type: `DATETIME`, `LOCAL DATETIME`, `DATE`, `TIME`, `LOCAL TIME`, `DURATION`, all of them after paths and before strings;
- durations by average length, a month counting 365.2425 / 12 days, then by months and then by days, so `PT24H` sorts before `P1D`, and `P1M` before `P31D`.

> Since v26.11.1. Before, a `localdatetime()` equalled the `datetime()` at the same UTC instant, durations compared with `<` component by component (so `P31D` was less than `P1M`), `ORDER BY` put `PT25H` before `P1D`, temporal values sorted after numbers, and `ORDER BY` over two temporal types failed with "Cannot compare DateTime with CypherDate".

15.5.18. Duration Arithmetic and Aggregation

Multiplying or dividing a duration is exact down to the nanosecond: a fraction of a month carries into days at the average month of 30.436875 days, a fraction of a day into seconds, and only the last digit, the nanosecond, is truncated toward zero. So `duration('P1D') / 24` is `PT1H` and `duration('P1M') / 3` is `P10DT3H29M42S`. An `INTEGER` operand is used as the integer it is, and a `FLOAT` as the decimal it is written as, so `duration('P1M') * 0.1` is exactly a tenth of a month. Neo4j computes the same cascade in double precision, and where the two answers differ it is by Neo4j's rounding error (it answers `PT7H18M17.459999999S` for `duration('P1M') / 100`).

`sum()` adds durations component by component, and `avg()` divides that sum by their count, the way `/` divides: `avg()` of `P1M` and `P1D` is `P15DT17H14M33S`. Numbers and durations cannot be mixed in one aggregation.

`duration.between()`, `duration.inMonths()` and `duration.inDays()` count months and days on the calendar of the first argument's time zone, and only the rest in elapsed time, so the day of a daylight saving change is one day: `duration.between(datetime('2024-03-30T12:00[Europe/Paris]'), datetime('2024-03-31T12:00[Europe/Paris]'))` is `P1D`, and `duration.inSeconds()` of the same pair is `PT23H`. The second argument is read in the time zone of the first, and a value with no time zone in the time zone of the other.

Since v26.11.1. Before, `duration('P1D') / 24` answered `PT59M59.999999999S`, an `INTEGER` factor past 2^53 was rounded, `duration.between()` and `inDays()` counted elapsed time across a daylight saving change (`PT23H`), and `sum()` / `avg()` refused durations.

15.5.19. Stored TIME, LOCAL TIME, Zoned DATETIME and DURATION Values

(Since v26.10.2.) ArcadeDB stores `time()`, `localtime()`, a zoned `datetime()` and `duration()` values natively, in the property types `OFFSET_TIME`, `LOCAL_TIME`, `ZONED_DATETIME` and `DURATION` (see Managing Dates). A value reads back as what it was written as, with no guessing from its text, so a String such as an opening-hours range `'09:00-17:00'` or a code `'P100D'` stays a String and compares, filters and sorts as one.

A database written by an earlier version holds these values as their ISO-8601 text (for example `10:35-08:00` or `PT1M30.5S`), turned back into a temporal value by their shape when an undeclared property is read. It keeps that reading after the upgrade until `CALL db.migrateLegacyTemporals()` converts the stored text to the native types (see Migrating Cypher Temporals Stored as Strings). Until then, declare a property `STRING` when its values may look like a time or a duration but are plain text: a declared `STRING` is never converted.

The `millisecond`, `microsecond` and `nanosecond` fields of `time()`, `localtime()`, `datetime()`, `localdatetime()` and the `*.truncate()` functions must each fit in their own range, as in Neo4j: `millisecond` 0-999, `microsecond` 0-999 when `millisecond` is also given (else 0-999999), `nanosecond` 0-999 when a coarser field is given (else up to 999999999). A value out of range, such as `time({hour: 12, millisecond: 5000})`, is an error rather than being carried into the seconds. Given with a base value (`time:`, `datetime:`, ...), these fields replace its whole fraction of second: `localtime({time: localtime('12:30:45.123456789'), nanosecond: 5})` is `12:30:45.000000005`. The adjustment map of a `*.truncate()` keeps the units the truncation left: `localtime.truncate('millisecond', t, {nanosecond: 5})` keeps the milliseconds of `t`.

`datetime({epochSeconds: ⋯})` and `datetime({epochMillis: ⋯})` take a `timezone` like any other `datetime()` map, and read the instant in it. A `dayOfWeek` without a `week` moves a base date within its own ISO week, and a `dayOfQuarter` without a `quarter` within its own quarter: `date({date: date('2024-02-29'), dayOfWeek: 1})` is `2024-02-26`. `datetime()` reads any year `date()` and `localdatetime()` read, such as `-0001-12-31T23:59:59Z` or `+10000-01-01T00:00Z`. The `timezone` of a `time()` can be a zone name: a time has no date, so `'Asia/Tokyo'` stands for the offset the zone has now (`+09:00`), as in Neo4j.

`duration.between(a, b)` splits the interval into whole months, then whole days, then the remaining time, as Neo4j does, so `a + duration.between(a, b)` is always `b`.

15.5.20. Building Temporal Values from a Map or a String

(Since v26.12.1.) The maps given to `date()`, `localtime()`, `time()`, `localdatetime()`, `datetime()` and their `*.truncate()` functions follow Neo4j:

- a date is given in one of four ways: calendar (`year`, `month`, `day`), week (`week`, `dayOfWeek`), quarter (`quarter`, `dayOfQuarter`) or ordinal (`ordinalDay`). Fields of two ways, such as `month` with `week`, are an error;
- without a base value, `year` is required (and `hour` for a time) and the fields cannot skip a level: `date({year: 2024, day: 5})` is an error;
- with a base value (`date:`, `datetime:`, ...), the fields replace its own in order, each keeping the day valid: `date({date: date('2024-02-29'), year: 2023, month: 3})` is `2023-03-28`;
- with a `timezone`, the other fields are applied first and the value is then converted to that zone. A map holding only a `timezone` is the current time in that zone;
- a field holding `null` takes its smallest value; a String field, an unknown key, or a base value of the wrong kind is an error. A whole number written as a float, such as `3.0`, is accepted.

Strings are parsed with the ISO-8601 forms Neo4j accepts, among them `2015-Q3`, `2015-W30-2`, `2015-202`, `+2015-07-21`, `2015-7-21`, `21:40:32,142`, `T21:40`, and zone names in any case (`[europe/london]`). A string whose offset does not match its zone, such as `2015-07-21T21:40:32-04:00[Europe/London]`, is refused with `Timezone and offset do not match`, in Cypher and when stored in a zoned SQL datetime property; a zone name on a time, such as `time('21:40[Europe/London]')`, is refused too.

Before v26.12.1, a missing `year` or a base value of the wrong kind answered January 1 of the current year, conflicting fields were dropped silently, and an offset that contradicted the zone was replaced by the zone's, which changed the instant.

15.5.21. Strict Numeric Types

The `IS TYPED` value-type predicate and `CREATE CONSTRAINT ⋯ IS TYPED <type>` honor the GQL numeric width hierarchy (`INT8`/`INT16`/`INT32`/`INT64`, `FLOAT32`/`FLOAT64`), mapping each onto a native ArcadeDB width so a declared property persists and reloads at that width. See IS TYPED data types for the full mapping.

15.5.22. Temporal and Spatial Value Types

The `IS TYPED` / `IS ::` value-type predicate and the `valueType()` function recognise temporal values and points by Neo4j's type names: `DATE`, `LOCAL DATETIME`, `ZONED DATETIME`, `LOCAL TIME`, `ZONED TIME`, `DURATION` and `POINT`. This holds whether Cypher built the value (`date()`, `datetime()`, ...) or it was read from a property written with SQL or the Java API. A point is a `POINT` and not a `MAP`, although it is a map at runtime. A list type with a non-nullable element type checks every element:

```
RETURN date() IS :: DATE                          // true
RETURN valueType(date())                          // 'DATE NOT NULL'
RETURN valueType(localdatetime())                 // 'LOCAL DATETIME NOT NULL'
RETURN valueType(duration('P1D'))                 // 'DURATION NOT NULL'
```

```
RETURN point({x: 1, y: 2}) IS :: MAP                    // false
RETURN [1, null] IS :: LIST<INTEGER NOT NULL>           // false
```

When upgrading to 26.10.2: before, `date() IS

DATE` and the same predicate on the other temporal types answered `false`, and `valueType()` returned the name of the internal Java class, such as `'CYPHERDATE NOT NULL'` for `valueType(date())`. A point was a `MAP`, and `LIST<INTEGER NOT NULL>` did not check the elements for `null`.

15.5.23. Relationship Types and Sub-types

A relationship type matches its sub-types, both in a pattern and in a predicate: with `CREATE EDGE TYPE Friend EXTENDS Knows`, `MATCH ()-[r:Knows]→()` and `MATCH ()-[r]→() WHERE r:Knows` both return the `Friend` relationships.

15.5.24. shortestPath() and Hop Bounds

`shortestPath()` and `allShortestPaths()` honor the `*min..max` quantifier written on the relationship: a shortest path outside that range is not an answer, and no longer path is substituted for it.

```
MATCH (a:City {name:'Rome'}), (b:City {name:'Oslo'}), p = shortestPath((a)-[:ROAD*..2]
-(b))
RETURN length(p)    // no row when the shortest Rome..Oslo path is longer than 2 hops
```

A relationship written without a quantifier, `-[:ROAD]-`, declares exactly one hop and is bounded accordingly.

When both endpoints resolve to the **same** vertex, the answer is the zero-length path only when the pattern declares a minimum of zero hops (`0..`, `*0..2`). Any other pattern, including `*1..` and a relationship without a quantifier, has no row for that pair: in the expression form `shortestPath(…)` returns `null` and `allShortestPaths(…)` an empty list.

```
MATCH (a:City {name:'Rome'}), p = shortestPath((a)-[:ROAD*0..]-(a)) RETURN length(p)
// 0
MATCH (a:City {name:'Rome'}), p = shortestPath((a)-[:ROAD*]-(a))    RETURN length(p)
// no row
```

Such a pattern is not reinterpreted as a request for the shortest cycle back to the start. Neo4j 5 refuses a shortest path whose start and end are the same node with an error; ArcadeDB drops the row instead, so a shortest path over a cartesian product (`MATCH (a), (b) MATCH p = shortestPatha)-[*]→(b`) answers every pair of distinct nodes.

When upgrading to 26.12.1: before, `*`, `*1..` and a relationship without a quantifier answered the zero-length path when both endpoints were the same node (issue #11913).

15.5.25. Self-Loops in Undirected Patterns

An undirected pattern follows a self-loop (a relationship from a node to itself) once, whether the pattern is written hop by hop or with a variable-length quantifier, as in Neo4j:

```
MATCH (n)-[*2]-() RETURN count(*)       // same count as the fixed-length form below
MATCH (n)-[]-()-[]-() RETURN count(*)
```

15.5.26. Indexes Inherited from a Parent Type

An index declared on a parent type serves queries on every child type, exactly as it does in SQL, so a single index at the top of a hierarchy is enough:

```
CREATE VERTEX TYPE Node;
CREATE VERTEX TYPE Entity EXTENDS Node;
CREATE PROPERTY Node.id STRING;
CREATE INDEX ON Node (id) UNIQUE;
```

```
MATCH (e:Entity) WHERE e.id = 'foo-1' RETURN e   // uses the inherited Node[id] index
```

A child type cannot declare a second index on a property its parent already indexes, and does not need one: the parent's index already covers the child's records.

15.5.27. Architectural Differences from Neo4j

While implementing the OpenCypher standard, ArcadeDB differs from Neo4j in these intentional design choices:

1. **Case-Insensitive Identifiers**: Type names and property keys follow SQL conventions
2. **Multi-Model Database**: Supports Document, Key-Value, and Graph models in one database
3. **Native Indexing**: Uses LSM-Tree indexes optimized for write-heavy workloads
4. **Query Language Flexibility**: Supports SQL, Cypher, Gremlin, GraphQL, and MongoDB Query Language on the same data
5. **User Functions**: Dynamic functions in 4 languages (SQL, JavaScript, OpenCypher, Java) vs Neo4j's Java-only compiled plugins

15.6. ISO GQL Coverage Statement

This page states which features of ISO/IEC 39075:2024 (GQL) ArcadeDB supports, supports in part, or does not support. Rows marked "since 26.10.2" describe a change in that release.

ArcadeDB has no separate GQL engine. GQL-aligned syntax runs in the native OpenCypher engine (query language `opencypher`), which implements the Cypher 25 grammar, the GQL-aligned revision of Cypher. GQL and Cypher 25 statements can be mixed freely in the same query, and no setting is

needed to enable them.

Status values used below:

- **Supported**: the feature runs with GQL semantics; any deviation is listed in the Notes column.
- **Partial**: part of the feature runs; the Notes column says which part.
- **Not supported**: the feature is not available.

15.6.1. Conformance testing

No public conformance test suite exists for ISO GQL yet, so ArcadeDB publishes no GQL suite results. The OpenCypher engine is validated against the official openCypher Technology Compatibility Kit on every build: 3,897 scenarios, 97.8% passing, 0 failing, 84 skipped. See Cypher Compatibility Matrix for the breakdown.

15.6.2. Graph pattern matching

GQL feature	Status	Notes
Node and edge patterns, directed and undirected (`-->`, `<--`, `--`)	Supported	The GQL tilde forms for undirected edges (`~[ ]~`) are not supported; use `-[ ]-`.
Label expressions: `&`, `|`, `!`, `%`, parentheses, `IS Label`	Supported	For example `(n:(Person&!Admin)|Admin)`, `(n IS Person & Employee)`, `-[r:!KNOWS]->`. On variable-length relationships since 26.10.2. A conjunction of relationship types (`r:A&B`) holds when the relationship's type is, or extends, every one of them, in a pattern and in a `WHERE` alike (since 26.12.1; the `WHERE` form was always false before).
Dynamic labels and types (Cypher 25): `$(expr)`, `$all(expr)`, `$any(expr)`	Supported	For example `MATCH (n:$($label))`, `-[r:$any(['KNOWS', 'LIKES'])]->`, `WHERE n:$($label)`. Since 26.12.1, as in Neo4j, the expression must evaluate to a string or a list of strings: a `null` (a missing parameter, for instance), a number, a map or a list with a `null` element is a type error. Before, a `null` matched every node. An empty list `$([])` matches every element. `$any` matches elements with at least one of the labels (it behaved like `$all` before 26.12.1). Since 26.12.1 a dynamic label can be combined with `!`, `%` and a mix of `&` and `|` (`(n:!$($label))`). Not allowed in `CREATE` and `MERGE` with those operators or with `$any`.
Element pattern `WHERE` (inline predicates in node and edge patterns)	Supported	`(n:Person WHERE n.age > 35)`, `-[r WHERE r.weight > 1]->`.
Quantified edges: `{m,n}`, `{n}`, `{,n}`, `+`, `*`	Supported	`-[:KNOWS]->{1,3}`. The quantifier `{0}` is refused.

GQL feature	Status	Notes
Quantified path patterns: `( ... ){m,n}`	Partial	Groups with fixed-length inner patterns and an inner `WHERE` are supported; inner variables bind as lists. The inner `WHERE` is checked as the `WHERE` of a clause is: an undefined variable or an aggregate function in it is refused, as in Neo4j (since 26.11.1). It can read the variables of the group and the one of the group's right boundary, which stands for the vertex the whole group ends on (`((x)-[:R]→(y) WHERE y.id ⇐ b.id){1,3} (b)`); it read null before 26.11.1, so the group matched nothing. An undirected hop inside a group matches a self-loop once, as the plain and the variable-length patterns do (since 26.11.1; it matched it twice before). Not supported inside a group: variable-length relationships, path variables, nested groups, parameterized property maps and dynamic labels.
Parenthesized path pattern with `WHERE` and no quantifier	Supported	Since 26.10.2, for example `MATCH ((a)-[:KNOWS]→(b) WHERE a.name = 'A')`.
Path variables (`p = ⋯`)	Supported	A path variable over a quantified group binds one path, as in Neo4j, since 26.10.2.
Multiple path patterns and multiple `MATCH` statements	Supported	
`OPTIONAL MATCH`	Supported	

15.6.3. Path modes, path search prefixes and match modes

GQL feature	Status	Notes
Path mode `TRAIL`	Supported	The default, as in GQL. See Path Mode.
Path mode `WALK`	Supported	The pattern needs an upper bound.
Path mode `ACYCLIC`	Supported	
Path mode `SIMPLE`	Not supported	
Path search prefix `ALL` / `ALL PATHS`	Supported	The default behavior.
Path search prefixes `ANY`, `ANY k`, `ANY SHORTEST`, `ALL SHORTEST`, `SHORTEST k`, `SHORTEST k GROUPS`	Not supported	Use the `shortestPath()` and `allShortestPaths()` functions. Since 26.10.2 these prefixes are refused with an error.

GQL feature	Status	Notes
Match mode `DIFFERENT EDGES`	Supported	The default, as in GQL (written `DIFFERENT RELATIONSHIPS` in Cypher 25). The GQL keyword `DIFFERENT EDGES` is not accepted.
Match mode `REPEATABLE ELEMENTS`	Not supported	Refused with an error since 26.10.2.

15.6.4. Query composition

GQL feature	Status	Notes
`RETURN`, `RETURN DISTINCT`	Supported	Grouping is implicit (Cypher style); the GQL `GROUP BY` clause and `SELECT` statement are not supported.
`FILTER [WHERE]`	Supported	
`FOR x IN list`	Supported	Equivalent to `UNWIND`. `WITH ORDINALITY` / `WITH OFFSET` are not supported.
`LET`	Supported	Since 26.10.2. See LET.
`ORDER BY`, `OFFSET` / `SKIP`, `LIMIT`	Supported	`NULLS FIRST` / `NULLS LAST` are not supported.
`FINISH`	Supported	
`NEXT` (statement chaining)	Not supported	Refused with an error. Use `WITH`.
`UNION`, `UNION ALL`, `UNION DISTINCT`	Supported	See UNION.
`EXCEPT`, `INTERSECT`, `OTHERWISE`	Not supported	
Conditional query `WHEN ⋯ THEN ⋯ ELSE` (Cypher 25)	Supported	Since 26.10.2, as a whole statement. See Conditional query.
`USE graph`	Partial	One ArcadeDB database is one graph: since 26.10.2 `USE` accepts the current database and refuses any other name.

15.6.5. Data modification

GQL feature	Status	Notes
`INSERT`	Supported	Nodes and edges, with `&` and `IS` labels, also after `MATCH`. Variable-length edges are refused.
`SET` (properties, `+=`, labels, `SET n IS Label`)	Supported	
`REMOVE`	Supported	
`DELETE`, `DETACH DELETE`, `NODETACH DELETE`	Supported	`DELETE` and `NODETACH DELETE` refuse a node that still has edges.
`CREATE`, `MERGE`	Supported	Cypher extensions, not part of GQL.

15.6.6. Predicates and expressions

GQL feature	Status	Notes
`IS [NOT] TYPED <type>`, `IS ::`, `::`	Partial	Supported for the types marked Supported under Value types. Union types, `RECORD`, `DECIMAL` and unsigned types are not supported. See IS TYPED data types.
`IS [NOT] NULL`	Supported	
`IS [NOT] LABELED`	Not supported	Use `n:Label` or `n IS Label`.
`IS [NOT] DIRECTED`, `IS SOURCE OF`, `IS DESTINATION OF`	Not supported	`IS DIRECTED` is refused with an error since 26.10.2 (every relationship is directed).
`IS [NOT] [NFC|NFD|NFKC|NFKD] NORMALIZED`	Supported	Since 26.10.2; NFC when no form is written, null for a value that is not a string.
`ALL_DIFFERENT()`, `SAME()`, `PROPERTY_EXISTS()`	Not supported	
`EXISTS { ... }`, `COUNT { ... }`, `COLLECT { ... }` subqueries	Supported	
`CASE` (simple and searched), `NULLIF`, `COALESCE`	Supported	
Comparison, `STARTS WITH`, `ENDS WITH`, `CONTAINS`, string concatenation `||`	Supported	
Three-valued logic	Supported	`UNKNOWN` is represented as `null`. See Three-valued logic.

GQL feature	Status	Notes
`CAST(x AS type)`	Not supported	Use `toInteger()`, `toFloat()`, `toString()`, `toBoolean()` and their `⋯OrNull()` variants.

15.6.7. Value types

GQL type	Status	Notes
`STRING`, `BOOLEAN`	Supported	
Signed integers: `INT8`, `INT16`, `INT32`, `INT64`, `INTEGER`	Supported	Stored at the declared width. An integer literal is `INT64`.
Unsigned integers (`UINT8` … `UINT64`)	Not supported	
`FLOAT32`, `FLOAT64`, `FLOAT`	Supported	
`DECIMAL`	Not supported	
`DATE`, `LOCAL TIME`, `ZONED TIME`, `LOCAL DATETIME`, `ZONED DATETIME`, `DURATION`	Supported	Including the `TIMESTAMP` / `TIME WITH[OUT] TIME ZONE` spellings. Typed literals such as `DATE '2026-01-01'` are not supported; use `date('2026-01-01')`.
`LIST<T>`, `NODE`, `EDGE` / `RELATIONSHIP`, `ANY`, `NOTHING`, `NULL`	Supported	
`PATH`	Supported	`IS TYPED PATH` and `valueType()` recognise paths, `shortestPath()` ones included, since 26.10.2.
`RECORD`	Not supported	Use maps.
`BYTES`	Not supported	

15.6.8. Functions

GQL functions	Status	Notes
String: `UPPER`, `LOWER`, `TRIM` (with `LEADING` / `TRAILING` / `BOTH`), `BTRIM`, `LTRIM`, `RTRIM`, `SUBSTRING`, `LEFT`, `RIGHT`, `NORMALIZE`, `CHAR_LENGTH`, `CHARACTER_LENGTH`	Supported	

GQL functions	Status	Notes
String: `OCTET_LENGTH`, `BYTE_LENGTH`	Not supported	
Numeric: `ABS`, `FLOOR`, `CEIL` / `CEILING`, `SQRT`, `EXP`, `LN`, `LOG`, `LOG10`, trigonometric and hyperbolic functions	Supported	
Numeric: `MOD`, `POWER`	Not supported	Use the `%` and `^` operators.
Aggregates: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX` (with `DISTINCT` / `ALL`), `COLLECT_LIST`, `PERCENTILE_CONT`, `PERCENTILE_DISC`	Supported	
Aggregates: `STDDEV_SAMP`, `STDDEV_POP`	Not supported	Use `stDev()` and `stDevP()`.
Graph: `ELEMENT_ID`, `PATH_LENGTH`	Partial	`elementId()` and `path_length()` are supported; the `element_id()` spelling is not.
List: `CARDINALITY`, `TRIM_ARRAY`, `ELEMENTS`	Not supported	Use `size()` and list slicing.
Temporal constructors: `date()`, `local_time()`, `zoned_time()`, `local_datetime()`, `zoned_datetime()`, `duration()`, `duration_between()`	Supported	
`CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`	Not supported	Use `date()`, `time()` and `datetime()` without arguments.

15.6.9. Transactions and sessions

GQL feature	Status	Notes
`START TRANSACTION`, `COMMIT`, `ROLLBACK`	Supported	The access modes `READ ONLY` / `READ WRITE` are not supported. The optional `ISOLATION` clause is an ArcadeDB extension. See Transaction control.
`SESSION SET $param = value`, `SESSION RESET`, `SESSION CLOSE`	Partial	Available with a server session (HTTP session or Bolt), not in embedded mode. `SESSION SET SCHEMA`, `SESSION SET GRAPH` and `SESSION SET TIME ZONE` are not supported. See Session management.

15.6.10. Catalog, graph types and diagnostics

GQL feature	Status	Notes
`CREATE GRAPH`, `DROP GRAPH`	Not supported	One ArcadeDB database is one graph. Databases are created through the server API, the console or Studio.
`CREATE GRAPH TYPE`, open and closed graph types, GQL node and edge type definitions	Not supported	ArcadeDB has a native schema, defined with SQL DDL (`CREATE VERTEX TYPE`, `CREATE EDGE TYPE`, `CREATE PROPERTY`). In Cypher, `CREATE CONSTRAINT … IS TYPED / IS NOT NULL / IS UNIQUE / IS KEY` declares property types and constraints.
GQLSTATUS codes	Not supported	Errors carry Neo4j-style classifications; over Bolt they are reported as `Neo.ClientError.*` codes.

Chapter 16. Gremlin

16.1. Gremlin

ArcadeDB supports Gremlin 3.7.x as query engine and in the Gremlin Server. You can execute a gremlin query from pretty much everywhere.

If you're using ArcadeDB as embedded, please add the dependency to the `arcadedb-gremlin` library. If you're using Maven include this dependency in your `pom.xml` file.

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-gremlin</artifactId>
    <version>26.11.1</version>
</dependency>
```

CRITICAL SECURITY NOTICE:

ArcadeDB uses the secure Java Gremlin engine by default.

DO NOT use the Groovy engine (`arcadedb.gremlin.engine=groovy`) in production or with untrusted users. The Groovy engine has a **CRITICAL Remote Code Execution (RCE) vulnerability** that allows authenticated users to execute arbitrary operating system commands.

See Gremlin Security for complete details about this vulnerability.

16.1.1. Gremlin from Java API

In order to execute a Gremlin query, you need to include the relevant jars, i.e. the `apache-tinkerpop-gremlin-server` libraries, plus `gremlin-groovy`, plus `opencypher-util-9.0`, in your class path. To execute a Gremlin query, use "gremlin" as first parameter in the query method. Example:

```
ResultSet result = database.query("gremlin",
"g.V().has('name','Michelle').has('lastName','Besso').out('IsFriendOf')");
```

If you're application is mostly based on Gremlin, the best way is to use the ArcadeGraph class as an entrypoint. Example:

```
try (final ArcadeGraph graph = ArcadeGraph.open("./databases/graph")) {
  Vertex michelleFriend = graph.traversal().V().has("name","Michelle").has("lastName"
,"Besso").out("IsFriendOf").next();
}
```

You can also work with a remote ArcadeDB Server:

```
try( RemoteDatabase database = new RemoteDatabase("127.0.0.1", 2480, "graph", "root",
"playwithdata") ){
  try (final ArcadeGraph graph = ArcadeGraph.open(database)) {
    Vertex michelleFriend = graph.traversal().V().has("name","Michelle").has
("lastName","Besso").out("IsFriendOf").next();
  }
}
```

Since a `RemoteDatabase` instance cannot be shared between threads, if your application is multi-threads, you can use the `ArcadeGraphFactory` to acquire `ArcadeGraph` instances from a pool. Example:

```
try (ArcadeGraphFactory pool = ArcadeGraphFactory.withRemote("127.0.0.1", 2480,
"mydb", "root", "playwithdata")) {
  try( ArcadeGraph graph = pool.get() ){
    // DO SOMETHING WITH ARCADE GRAPH
  }
}
```

Remember to call `ArcadeGraph.close()` to release the `ArcadeGraph` instance back into the pool. `ArcadeGraph` implements `Closable`, so you can just use it in a `try` block like in the example above.

16.1.2. Transactions and closing a graph

Closing an `ArcadeGraph` **rolls back** any transaction still open, which is the TinkerPop default. Changes that never reached `commit()` are not saved just because the graph was closed:

```
try (final ArcadeGraph graph = ArcadeGraph.open("./databases/graph")) {
  graph.tx().begin();
  graph.addVertex("Person");
  graph.tx().commit(); // <-- WITHOUT THIS, THE VERTEX IS NOT SAVED
}
```

If you want the graph to commit when it closes, ask for it explicitly:

```
graph.tx().onClose(Transaction.CLOSE_BEHAVIOR.COMMIT);
```

An `ArcadeGraph` borrowed from an `ArcadeGraphFactory` pool is always returned clean: any transaction left open is rolled back before the instance goes back into the pool, so the next caller never inherits your uncommitted changes. Do not keep using an instance after calling `close()`, and note that `drop()` is not allowed on a pooled instance, because the database is shared with the rest of the pool.

The `ArcadeGraph` API above follows the TinkerPop transaction model. A Gremlin statement sent through `database.command("gremlin", …)` handles its transaction like a Cypher write instead:

- If no transaction is active and the statement writes, it runs in one transaction, committed once the traversal has run to its end, before `command()` returns: the write happens even if the result is closed unread, and the rows it returns are kept in memory until the commit. A statement that fails leaves nothing behind, and one that meets a concurrent conflict is run again, up to `arcadedb.txRetries` times. A Groovy script is treated as a write. *(Since v26.12.1; before, the statement opened a transaction that it left open in the calling thread, so the partial write of a failed traversal was published by the next unrelated `commit()`, issue #10220)*
- If no transaction is active and the statement only reads, its results still stream.
- If a transaction is already active, the statement runs in it and leaves the commit or rollback to its owner.

```
// Automatic transaction - committed when command() returns, rolled back if the
traversal fails
database.command("gremlin", "g.addV('Person').property('name', 'Alice')");
```

16.1.3. Vertex property identifiers

`VertexProperty.id()` returns a `String` in the form `#<bucket>:<offset>-<propertyName>`, consistent with vertex and edge identifiers. It previously returned a `Long` computed from hash codes, which could come out the same for two different properties of the same vertex, making them compare as equal and disappear from results such as `properties().dedup()`.

If your application casts or parses a vertex property id as a number, update it to handle a `String`.

16.1.4. Gremlin through Postgres Driver

You can execute a Gremlin query against ArcadeDB server by using the Postgres driver and prefixing the query with `{gremlin}`. Example:

```
"{gremlin} g.V().has('name','Michelle').has('lastName','Besso').out('IsFriendOf')"
```

ArcadeDB server will execute the query `g.V().has('name','Michelle').has('lastName','Besso').out('IsFriendOf')` using the Gremlin query language.

16.1.5. Gremlin through HTTP/JSON

You can execute a Gremlin query against ArcadeDB server by using HTTP/JSON API. Example of executing an idempotent query with HTTP GET command:

```
curl
"http://localhost:2480/query/graph/gremlin/g.V().has('name','Michelle').has('lastName'
,'Besso').out('IsFriendOf')"
```

Example of executing a non-idempotent query (updates the database):

```
curl -X POST "http://localhost:2480/command/graph" -d "{'language': 'gremlin',
'command': 'g.V().has(\"name\",\"Michelle\").has(\"lastName\",\"Besso\").out(
\"IsFriendOf\")'}"
```

Before running a Gremlin script, ArcadeDB checks whether it writes, without running it. The query endpoint refuses a script that writes, and on a cluster a replica forwards it to the leader, where it runs once. Every statement of a script counts, so `g.addV('Person').iterate(); g.V().count()` is a write. A script that only the Groovy engine can read (`arcadedb.gremlin.engine` set to `auto` or `groovy`) has to be evaluated to be checked: a traversal it runs while being checked runs only if it reads *(since v26.11.1)*.

A traversal that is wrong is an error in the request, answered with HTTP 400 and TinkerPop's own message prefixed by `Invalid gremlin traversal:`, like a Gremlin syntax error. That covers a traversal TinkerPop refuses for its shape (`g.V().emit()`, a `repeat()` with no traversal; `g.V().to('V')`, a step that cannot follow the previous one), a step that meets a value of a type it is not defined for (`g.V().sum()`, `g.V().inV()`), and a step that needs what the path does not hold (`otherV()` with no edge before it). A valid traversal that fails while it runs, such as `.next()` on an empty traversal, is still answered with 500. A step naming only a `null` label (`out(null)`) matches nothing, as in TinkerGraph, while a `null` among other labels (`bothE(null, 'x')`) is refused by TinkerPop. *(Since v26.12.1: these were answered with HTTP 500, and a `null` label failed with a `NullPointerException`.)*

16.1.6. Use the Gremlin Server

Apache TinkerPop Gremlin provides a standalone server to allow remote access with a Gremlin client. In order to use the Gremlin Server with ArcadeDB, you have to enable it from ArcadeDB's server plugin system:

```
~/arcadedb $ bin/server.sh -Darcadedb.server.plugins
="GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugin"
```

If you're using MS Windows OS, replace `bin/server.sh` with `bin\server.bat`.

At startup, the Gremlin Server plugin looks for the file `config/gremlin-server.yaml` under ArcadeDB path. If the file is present, the Gremlin Server will be configured with the settings contained in the YAML file, otherwise the default configuration will be used.

You can also override single configuration settings by using ArcadeDB's settings and prefixing the configuration key with `gremlin.`. All the configuration settings with such a prefix will be passed to the Gremlin Server plugin.

Multi-Database Support

ArcadeDB's Gremlin Server includes `ArcadeGraphManager`, a custom implementation of TinkerPop's `GraphManager` interface that provides **dynamic, on-demand registration** of databases as Gremlin graphs.

Key Features:

- **Automatic Registration**: When a Gremlin client requests a graph/traversal source by database name, `ArcadeGraphManager` automatically registers it if the database exists in ArcadeDB Server
- **Multi-Database Access**: Access any database on the server through Gremlin without static configuration
- **The `g` Alias**: The standard `g` alias automatically maps to the default database ("graph" if it exists, otherwise the first available database)

How It Works:

Instead of static graph configuration in YAML files, databases are registered dynamically when first accessed. Each database name becomes a valid traversal source alias:

```
// Connect to a specific database using its name as the alias
var cluster = Cluster.build()
    .port(8182)
    .addContactPoint("localhost")
    .credentials("root", "password")
    .create();

// Access database "mydb" - registered automatically on first access
var g = traversal().withRemote(DriverRemoteConnection.using(cluster, "mydb"));

// Or use the standard "g" alias for the default database
var gDefault = traversal().withRemote(DriverRemoteConnection.using(cluster, "g"));
```

Script-Based Queries:

For script-based Gremlin queries (e.g., via `client.submit()`), the `g` variable is automatically bound to the default database:

```
Client client = cluster.connect();
// The "g" variable is automatically available
ResultSet results = client.submit("g.V().hasLabel('Person').count()");
```

Benefits:

- No need to manually configure graphs in `gremlin-server.yaml` or `gremlin-server.groovy`
- Supports ArcadeDB's multi-tenant architecture where a single server hosts multiple databases
- Each database is isolated and accessed independently through its name
- Reduces configuration complexity for deployments with many databases

Legacy static configuration via `gremlin-server.yaml` graphs section is no longer required. All databases are available automatically

Migrating from Static Graph Configuration

If you configured graphs statically in `gremlin-server.yaml`, here are the key changes to be aware of:

What Changed:

1. **Graph Registration**: Graphs are now registered dynamically by `ArcadeGraphManager` instead of statically in configuration files
2. **Configuration Files**: The `graphs:` section in `gremlin-server.yaml` is no longer required
3. **The `graph` Binding**: The internal `graph` variable is now managed by `ArcadeGraphManager` to properly support TinkerPop sessions and transactions

Backward Compatibility:

- Existing Gremlin queries continue to work without modification
- The `g` alias still works and maps to the default database
- Script-based queries using `g` are fully supported
- Bytecode-based queries work with any database name as the alias

Configuration Migration:

If you had custom `gremlin-server.yaml` configuration:

```
# OLD configuration (no longer needed)
graphs:
  graph: conf/arcadedb.properties
```

This static configuration is now optional. `ArcadeGraphManager` will automatically register any database when it is first accessed. You can safely remove the `graphs:` section.

Custom gremlin-server.groovy:

If you customized `gremlin-server.groovy`, ensure you are NOT overwriting the `graph` binding:

```
// CORRECT: Only define the traversal source 'g'
globals << [g : traversal().withEmbedded(graph)]

// WRONG: Do NOT overwrite 'graph' - this breaks session support
// globals << [graph : traversal().withEmbedded(graph)]  // DON'T DO THIS
```

The `graph` variable must remain a `Graph` object (not a `GraphTraversalSource`) for TinkerPop's `SessionOpProcessor` to work correctly with transactions.

Accessing Multiple Databases:

To access a specific database, use the database name as the traversal source alias:

```
// Use the database name directly
var g = traversal().withRemote(DriverRemoteConnection.using(cluster, "customers"));
var g2 = traversal().withRemote(DriverRemoteConnection.using(cluster, "products"));
```

Transactions via Remote Driver:

Gremlin transactions now work correctly via the remote driver:

```
GraphTraversalSource g = traversal().withRemote(
    DriverRemoteConnection.using(cluster, "mydb"));

Transaction tx = g.tx();
GraphTraversalSource gtx = tx.begin();
try {
    gtx.addV("Person").property("name", "Alice").iterate();
    tx.commit();
} catch (Exception e) {
    tx.rollback();
}
```

Docker Example:

Start the Gremlin Server with the `OpenBeer` dataset imported as a database named `openbeer`:

```
docker run -d --name arcadeDB -p 2480:2480 -p 8182:8182 \
       --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata \

-Darcadedb.server.defaultDatabases=openbeer[root]{import:https://github.com/ArcadeData
/arcadedb-datasets/raw/main/orientdb/OpenBeer.gz} \

-Darcadedb.server.plugins=GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugi
n " \
          arcadedata/arcadedb:latest
```

When running ArcadeDB with Docker, expose port `8182` for Gremlin Server and use `-e` to pass JVM settings.

Gremlin Server listens on `8182` unless you set `gremlin.port` (for example `-Dgremlin.port=8183`). The remote `ArcadeGraph` client reads the port from the server; if the port is remapped outside ArcadeDB, set `arcadedb.gremlin.client.port` on the client.

Connecting to the Gremlin Server:

```
var cluster = Cluster.build()
    .port(8182)
```

```
    .addContactPoint("localhost")
    .credentials("root", "password")
    .create();

// Connect using the database name as the alias
var g = traversal().withRemote(DriverRemoteConnection.using(cluster, "openbeer"));

// Or use "g" for the default database
var gDefault = traversal().withRemote(DriverRemoteConnection.using(cluster, "g"));
```

16.1.7. Listing and terminating running traversals

Since v26.10.2 a Gremlin traversal can be stopped while it runs, whichever way it was sent. The HTTP `list queries` server command, Cypher `SHOW TRANSACTIONS` and the Studio Running Queries tab list it with language `gremlin`; a request sent to the Gremlin Server is listed with `protocol gremlin`, one sent over HTTP or another protocol with that protocol. `terminate query` or `TERMINATE TRANSACTIONS` stops it: the traversal checks for the request inside its adjacency steps (`out()`, `in()`, `both()`, `outE()`, `bothE()`, ...) and while its results are iterated, so even a traversal that expands a large graph stops promptly, and the client receives an error saying that it was terminated.

16.1.8. Known Limitations with the Gremlin Implementation

- ArcadeDB automatically handles the conversion between compatible types, such as strings and numbers when possible. Gremlin does not. So if you define a schema with the ArcadeDB API and then you use Gremlin for a traversal, ensure you're using the same type you defined in the schema. For example, if you define a property "id" to be a string, and then you're executing traversal by using integers for the ids, the result could be unpredictable.
- ArcadeDB's Gremlin implementation always tries to optimize the Gremlin traversal by using ArcadeDB's internal query. While this is easy with simple traversal using `.has()` and `.hasLabel()`, it is unable to optimize more complex traversal with `select()` and `where()`. Instead of executing an optimized query, it could result in a full scan of the type, leaving to Gremlin the filtering. While the result of the traversal is still correct, the performance would be heavily impacted. Please consider using ArcadeDB's SQL or Native Select for the best performance with complex traversals.
- A `hasLabel('X')` that directly follows `g.V()` or `g.E()` is answered from the type `X` **polymorphically**, like SQL's `SELECT FROM X`: it matches `X` and every type that extends it. This holds whichever way ArcadeDB executes it (a type scan, an index lookup on a property of `X` or of one of its super types, or `count()`), so `g.V().hasLabel('X').count()` always equals the number of elements `g.V().hasLabel('X')` returns. A `hasLabel()` further down a traversal, for example after `out()`, compares the element's label exactly, as TinkerPop defines it.
- *(Since v26.10.2)* A `g.V().hasLabel('X').has('k', v)` answered from an index keeps working while a `CREATE INDEX` or `DROP INDEX` runs on the same type: the traversal checks its index again when it opens the scan and, if that index is gone, uses the index now on the property or scans the type. Before, such a traversal could fail or return a wrong count.
- A `hasLabel('bucket:<name>')` that directly follows `g.V()` or `g.E()` restricts the traversal to that one bucket, and `count()` counts that bucket.

- *(Since v26.11.1)* A `hasLabel()` naming nothing, whether a type or a bucket that does not exist, a bucket no type owns, or a type of the other element kind, matches no element, with or without a following `count()`: `g.V().hasLabel('bucket:Missing').count()` answers `0`, like `g.V().hasLabel('Missing').count()`. Before, the bucket form with `count()` failed with `SchemaException: Bucket with name 'Missing' was not found` (issue #10763). Writing to such a bucket, `addV('bucket:Missing')` or `addE('bucket:Missing')`, is refused with `IllegalArgumentException: Bucket 'Missing' does not exist` (or `does not belong to any type`), since a write cannot create a bucket.
- *(Since v26.11.1)* A query sent as text that ends with an eager terminal step answers the step's value as one result, `null` included: `g.V().coalesce(values('x'), constant(null)).next()` answers one result whose `result` is `null`, as the same traversal without `next()` does. Before, a `null` value failed the query with a `NullPointerException` (issue #10770). `tryNext()` answers the value it holds, or no result when the traversal is empty.
- A query sent as text (`graph.gremlin(…)`, `POST /api/v1/command`) returns a map result, such as `elementMap()` or `groupCount()`, as one result whose properties are the map's keys. The shape is always a flat map, whatever the data. The `id` and `label` tokens of `elementMap()` and `valueMap(true)` are named `@rid` and `@type`, so a property called `id` or `label` never overwrites them. Keys that print the same, for example the keys `1` (Integer) and `1L` of a `groupCount()`, are told apart by their type: `1:Integer` and `1:Long`. A `null` key is named `null`.
- *(Since v26.12.1)* The range predicates `lt`, `lte`, `gt`, `gte` (and `between`, `inside`, `outside`, which are built on them) follow the comparison rules of TinkerPop 3.8.2: a comparison with `NaN` on either side is false, and two lists, sets or maps are compared only when their contents are comparable, otherwise the comparison is false. So `g.V().has('x', gt(1))` keeps working when some vertex stores `x = NaN`, and `g.inject(['a':'x']).is(lt(['b':1]))` matches nothing, as in TinkerGraph. Before, one stored `NaN` made every range `has()` on that property fail with `IllegalStateException` (HTTP 500), and incomparable lists, sets or maps either failed the same way or were ordered by their keys (issue #12018). One difference from TinkerPop is kept on purpose: date and time values of different Java classes (`Date`, `LocalDateTime`, `OffsetDateTime`, …) are compared by the instant they denote.
- *(Since v26.12.1)* `properties()`, `values()` and `valueMap()` return each property of a vertex or an edge at most once, however many times its key is named: `g.V().values('n','n').count()` counts each vertex's `n` once and `valueMap('n','n')` answers `{n=[1]}`, as in TinkerGraph. Before, the property came back once per mention, so `values('n','n').count()` answered twice the number of vertices (issue #12026). The values come in the order their keys are first named, where TinkerGraph uses the element's own property order.
- *(Since v26.12.1)* The Gremlin Server returns the value of an `EMBEDDED` property as a map holding the document's properties and its `@type` and `@cat`, the same content the HTTP endpoint returns, with both the GraphBinary and the GraphSON serializers. This holds for embedded documents nested in a list, a map or another embedded document. Before, any result carrying such a value failed with a serialization error and the whole result was lost (issue #13043). With GraphBinary the document travels as the ArcadeDB custom type `arcadedb.EmbeddedDocument`, like a RID: the `ArcadeGraph` remote client reads it as is, while a stock TinkerPop client has to register `ArcadeIoRegistry` (from `arcadedb-gremlin`) on its serializer.
- The edges of a `LIGHTWEIGHT` edge type have no record, so they have no usable RID. Gremlin still lists, counts and deletes them: `g.E().hasLabel('X')`, `outE()`, `inE()`, `bothE()`, `out('X').count()` and

`drop()` all work. Listing a lightweight type by `g.E()` walks the vertices, so it costs more than the scan of a regular edge type.

- *(Since v26.11.1)* An element dropped earlier in a traversal, for example by `sideEffect(drop())`, behaves as in TinkerGraph: reading it finds no properties, edges or adjacent vertices (`valueMap()` answers `{}`, `inV()` from a dropped edge answers nothing), and writing to it fails with `IllegalStateException: Vertex with id … was removed.`, so `g.V().has('uid', 1).sideEffect(drop()).addE('r').to(…)` creates no edge. Dropping an element that is already gone is still a no-op. Before, the dropped element kept answering its old properties, `property()` on it was silently ignored, and `addE()` from it committed an edge whose source vertex no longer existed (issue #10196). A rolled back transaction makes the element usable again.

For more information about Gremlin:

- Introduction to Gremlin
- Getting Started with Gremlin
- The Gremlin Console
- Gremlin Recipes
- Java Gremlin Documentation
- PRACTICAL GREMLIN: An Apache TinkerPop Tutorial

16.1.9. Recommended Tools with Gremlin

If you're using Gremlin with ArcadeDB, check out gdotv graph tool. It is compatible with ArcadeDB and provides a powerful visual debugger, advanced graph analytics, and much more.

[debugging queries short]

Chapter 17. GraphQL

17.1. GraphQL

ArcadeDB Server supports a subset of the GraphQL specification. Please open an issue or a discussion on GitHub to increase the support for GraphQL.

If you're using ArcadeDB as embedded, please add the dependency to the `arcadedb-graphql` library. If you're using Maven include this dependency in your `pom.xml` file.

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-graphql</artifactId>
    <version>26.11.1</version>
</dependency>
```

GraphQL is supported in ArcadeDB as a query language engine. This means you can execute a GraphQL command from:

- Java API by using the non-idempotent `.command()` and the idempotent `.query()` API by using `"graphql"` as language. Example: `Resultset resultset = db.query("graphql", "{ bookById(id: "book-1"){ id name authors { firstName, lastName } }");`
- HTTP API by using `/command` and `/query` commands and `"graphql"` as language
- Postgres Driver by prefixing with `{graphql}` your query to execute. Example: `{graphql}{ bookById(id: "book-1"){ id name authors { firstName, lastName } }`

17.1.1. Type definition

GraphQL requires to define the types used. If you're using the Document Model and links to connect documents, then you can map 1-1 the GraphQL type to ArcadeDB type. Example:

```
type Book {
  id: ID
  name: String
  pageCount: Int
  authors: [Author]
}
```

If you're using a Graph Model for your domain, then you need to declare with a GraphQL directive how the relationship is translated on the graph model.

In the example below, the `authors` is a collection of Author retrieved by looking at the incoming (`direction: IN`) edges of type "IS_AUTHOR_OF" (`type: "IS_AUTHOR_OF"`):

```
type Book {
```

```
  id: ID
  name: String
  pageCount: Int
  authors: [Author] @relationship(type: "IS_AUTHOR_OF", direction: IN)
}
```

Directives can be defined on both types and queries. Directives defined in queries override any directives defined in types, only for the query execution context.

You can define your model incrementally and apply it to the current database instance by executing a command containing the type definition. Example by using Java API (but the same by using HTTP Command API):

```
String types = "type Query {" +
               "  bookById(id: ID): Book" +
               "}" +
               "type Book {" +
               "  id: ID" +
               "  name: String" +
               "  pageCount: Int" +
               "  authors: [Author] @relationship(type: \"IS_AUTHOR_OF\", direction:
IN)" +
               "}" +
               "type Author {" +
               "  id: ID" +
               "  firstName: String" +
               "  lastName: String" +
               "}";

database.command("graphql", types);
```

With this example the types `Book` and `Author` are defined together with the query `bookById`. You can add new types or replace existing types by just submitting the type(s) again. The GraphQL module will update the current definition of types.

This definition is not saved in the database and must be declared after the database is open, before executing any GraphQL queries.

Submitting a type definition changes the GraphQL schema shared by every user of the database, so it requires the `updateSchema` right. Users without it get a security error on the command endpoint, and the read-only query endpoint refuses type definitions because they are not read-only. Plain GraphQL queries are not affected.

(Since v26.11.1) The type definitions of one command are applied together, before its query runs: a query running at the same time sees all of them or none of them. In 26.10.2 and earlier a query running while types were being defined could fail with `Target type not defined`, or with a `ConcurrentModificationException` on introspection (issue #10764).

17.1.2. Introspection of database types

Introspection (`__type(name:)` and `__schema { types }`) also describes the database types that have no GraphQL type definition, deriving each field from the property type. *(Since v26.10.2)* Every field is described as the GraphQL result serializes it:

Property type	Introspection type
`INTEGER`, `SHORT`, `BYTE`	`SCALAR Int`
`LONG`	`SCALAR Long`
`FLOAT`, `DOUBLE`	`SCALAR Float`
`BOOLEAN`	`SCALAR Boolean`
`MAP`, and `EMBEDDED` with no declared type	`SCALAR JSON`: a JSON object whose keys the schema does not declare
`EMBEDDED OF <T>`	`OBJECT <T>`, with the fields of `<T>`, which a sub-selection can walk
`LINK`	`SCALAR ID`: the value is the record id of the linked record, such as `"#28:0"`
`DECIMAL`	`SCALAR BigDecimal`: a JSON number that keeps every digit
`BINARY`	`LIST` of `Int`, one element per byte
`ARRAY_OF_SHORTS`, `ARRAY_OF_INTEGERS` / `ARRAY_OF_LONGS` / `ARRAY_OF_FLOATS`, `ARRAY_OF_DOUBLES`	`LIST` of `Int` / `Long` / `Float`
`LIST OF <X>`	`LIST` of the type `<X>` maps to (a `LIST` with no declared element type is a `LIST` of `String`)
`DATE`, `DATETIME` and the other temporal types, `STRING`	`SCALAR String`

`JSON` and `BigDecimal` are custom scalars, listed by `__schema { types }` and resolved by `__type(name:)` like `Long`. They describe the value only: no coercion is applied to a result or to an argument.

A sub-selection on a `LINK` property, or on a list of links, resolves the linked records, as it does on an embedded document:

```
{ docs { owner { city } links { city } } }
```

Without a sub-selection the same fields return the record ids (`"#28:0"`, `["#28:0"]`). *(Since v26.10.2)*

Every selected field is in the response, with `null` when the record has no value for it, whether the field is a scalar or an object. A sub-selection on a scalar field, such as `{ books { name { first } } }` with `name: String`, is refused with an error naming the field. *(Since v26.11.1)* In 26.10.2 and earlier an object field with no value (an unset `LINK`, for example) was left out of the response, and a sub-selection on a scalar field dropped the field without an error (issue #10762).

17.1.3. Query arguments

Query fields can declare arguments of the standard GraphQL scalar types: `ID`, `String`, `Int`, `Float` and `Boolean`. Example:

```
type Query {
  books(inStock: Boolean, rating: Float): [Book]
}

type Book {
  id: ID
  name: String
  inStock: Boolean
  rating: Float
}
```

```
ResultSet resultSet = database.query("graphql", "{ books(inStock: true, rating: 4.5) {
name } }");
```

A query field can also be declared without any argument, for example to list every record of a type:

```
type Query {
  allBooks: [Book]
}
```

Argument values are passed to the database as parameters, never written into the generated query text, so a value containing quotes or backslashes is always treated as data. The standard GraphQL escape sequences (`\"`, `\\`, `\/`, `\b`, `\f`, `\n`, `\r`, `\t` and `\uXXXX`) are decoded, so this matches a book whose name really contains a line break:

```
{ bookByName(name: "Line one\nLine two") { id } }
```

An argument value can be any GraphQL literal: a string, a block string, a number, a boolean, `null`, an enum value, a list or an object. A block string is written between triple quotes and taken as written, with no escape sequence but `\"""`; the indentation its lines share and its blank first and last lines are removed, which makes it convenient for the multi-line statement of a `@sql` directive:

```
type Query {
  bookByName(name: String): [Book] @sql(statement: """
    select from Book
    where name = :name
    """)
}
```

An integer literal is checked against the type of the argument it is written for: an `Int` argument accepts only a 32-bit value, while a `Long` or an `ID` argument accepts a wider one, such as a timestamp in milliseconds:

```
{ eventsAt(ts: 1700000000000) { id } }
```

An argument can declare a default value, used when the query leaves the argument out or binds it to a variable that received no value, whether the field has a directive or not. An argument declared non-null (for example `id: ID!`) with no default must be provided and cannot be `null`, otherwise the query is refused. A nullable argument with no default that is left out adds no condition:

```
type Query {
  books(name: String = "Mr. brain"): [Book]
  bookById(id: ID!): Book
}
```

(since v26.12.1) Before, a field without a directive ignored the default of an argument left out and accepted a non-null argument left out, so both returned every record of the type; an argument bound to a variable that received no value was compared with `null`; the `null` literal and block strings were syntax errors; and an integer literal beyond 32 bits failed the whole query, whatever the type of its argument (issues #11447 and #11451).

The `where` argument

An argument named `where` is reserved: its value is used as the SQL condition of the generated query instead of being compared against a property.

```
type Query {
  books(where: WHERE): [Book]
}
```

```
ResultSet resultSet = database.query("graphql", "{ books( where: \"name = 'Mr.
brain'\" ) { name } }");
```

When the field has other arguments, the `where` condition is one parenthesised operand of the `AND` that joins it to them, so `books(name: "Target", where: "pages > 100 or pages < 10")` returns only the books named `Target`, whatever the order of the arguments. The value must therefore be a complete SQL condition on its own: a text that closes and reopens parentheses (`"pages > 100) or (true"`), or that adds clauses such as `ORDER BY` or `LIMIT`, is refused with a parsing error.

(since v26.11.1) Before, the `where` text was appended to the other conditions without parentheses, so an `OR` in it escaped the other arguments (the example above also returned books with other names), and a text balancing its own

parentheses could reshape the generated query.

The `where` value is SQL and is used exactly as written, so it must be authored in the query, never built from end-user input. For this reason it cannot be filled from a variable. If your condition has to change at runtime, use a `@sql` directive with a parameter instead.

17.1.4. Query variables

A query can declare variables and receive their values at execution time, so the same query text can be reused with different values:

```
query($name: String) {
  bookByName(name: $name) { id name }
}
```

```
ResultSet resultSet = database.query("graphql", "query($name: String) {
bookByName(name: $name) { id name } }",
                                     "name", "Harry Potter and the Philosopher's
Stone");
```

From the HTTP API the variables are the request parameters, and from the Postgres Driver they are the query parameters.

A variable can declare a default value, used when no value is passed:

```
query($name: String = "Mr. brain") { bookByName(name: $name) { id } }
```

A variable declared with neither a value nor a default leaves out the argument it is bound to, which then takes its own default. A variable passed as an explicit `null`, or with `null` as its default, is the value `null`.

Variables can also be used in the arguments of a directive written in the query:

```
query($edge: String) { bookById(id: "book-1") { authors @relationship(type: $edge,
direction: IN) { lastName } } }
```

The value passed must match the type the variable is declared with, otherwise the query is rejected:

- `Int` accepts a whole number in the 32-bit range
- `Float` accepts any number, but not `NaN` or infinity
- `Boolean` accepts only a boolean, `String` only a string
- `ID` accepts a string or a whole number of any size

- `Long` accepts a whole number in the 64-bit range, or a string holding one (what a JSON client sends for a value wider than a JavaScript number keeps exact); a decimal number such as `3.14` or `3.0` is refused, as for `Int`, in a variable and in a literal alike *(since v26.12.1: a decimal was accepted and matched nothing)*
- any other type name (a custom scalar, an enum, a list) is passed through unchecked

A variable declared with `!` (for example `$name: String!`) must receive a value, and that value cannot be null. A variable used but never declared is an error, as is a variable used for the `where` argument.

Commas between variable declarations are not supported yet: write `query($a: String $b: Int)`, not `query($a: String, $b: Int)`.

17.1.5. Field aliases

A selection can rename a field with the standard GraphQL alias syntax `alias: field`. The alias becomes the key of the field in the result, while the underlying field is resolved normally.

Example:

```
{ bookById(id: "book-1")
  {
    title: name
    authors {
      first: firstName
      lastName
    }
  }
}
```

This returns `title` (instead of `name`) and `first` (instead of `firstName`) as the result keys.

The top-level query field itself can also be aliased, including when it takes arguments:

```
{ myBooks: books(inStock: true) { name } }
```

17.1.6. Fragments

A selection set can reuse a named fragment with the spread syntax `…Name`, or write an inline fragment with `… on Type { }`. The fields the fragment selects are resolved as if they were written in place of the spread, at any depth, and a fragment can be declared before or after the operation that uses it:

```
fragment AuthorName on Author { firstName lastName }
fragment BookInfo on Book { name authors { ...AuthorName } }

{ bookById(id: "book-1") { id ...BookInfo } }
```

A fragment applies when its type condition names the type the selection is written against, or the database type of the record or one of its super types, so `… on Novel { pageCount }` under a `Book` field returns `pageCount` only for the records that are novels. When the same field is selected more than once, directly or through fragments, the sub-selections are merged into one field. A fragment that is not defined, or fragments that spread each other in a cycle, are rejected before any record is read.

17.1.7. Supported directives

Directives can be defined on both types and queries. Directives defined in queries override any directives defined in types, only for the query execution context.

@relationship

Applies to: `Query Field` and `Field Definition`

Syntax: `@relationship([type: "<type-name>"] [, direction: <OUT|IN|BOTH>])`

Where:

- `type` is the edge type, optional. If not specified, then all the types are considered
- `direction` is the direction of the edge, optional. If not specified, then BOTH is used

Example:

```
friends: [Account] @relationship(type: "FRIEND", direction: BOTH)
```

Relationships can be traversed at any depth, each level being resolved against its own type. The query below walks from a book to its authors, then back to every book each author wrote:

```
{ bookById(id: "book-1")
  {
    name
    authors {
      lastName
      wrote { name }
    }
  }
}
```

@sql

Applies to: `Query Field` and `Field Definition`

Syntax: `@sql( statement: <sql-statement> )`

Executes a SQL query. The query can use parameters passed at invocation time.

Example of definition of a query using SQL in GraphQL:

```
bookByName(bookNameParameter: String): Book @sql(statement: "select from Book where
name = :bookNameParameter")
```

Invoke the query defined above passing the book name as parameter:

```
ResultSet resultSet = database.query("graphql", "{ bookByName(bookNameParameter:
\"Harry Potter and the Philosopher's Stone\")}"));
```

@gremlin

Applies to: `Query Field` and `Field Definition`

Syntax: `@gremlin( statement: <gremlin-statement> )`

Executes a Gremlin query. The query can use parameters passed at invocation time.

Example of definition of a query using Gremlin in GraphQL:

```
bookByName(bookNameParameter: String): Book @gremlin(statement: "g.V().has('name',
bookNameParameter)")
```

Invoke the query defined above passing the book name as parameter:

```
ResultSet resultSet = database.query("graphql", "{ bookByName(bookNameParameter:
\"Harry Potter and the Philosopher's Stone\")}"));
```

@cypher

Applies to: `Query Field` and `Field Definition`

Syntax: `@cypher( statement: <cypher-statement> )`

Executes a Cypher query. The query can use parameters passed at invocation time.

Example of definition of a query using Cypher in GraphQL:

```
bookByName(bookNameParameter: String): Book @cypher(statement: "MATCH (b:Book {name:
$bookNameParameter}) RETURN b")
```

Invoke the query defined above passing the book name as parameter:

```
ResultSet resultSet = database.query("graphql", "{ bookByName(bookNameParameter:
```

```
\"Harry Potter and the Philosopher's Stone\")}"));
```

@rid

Applies to: `Query Field` and `Field Definition`

Syntax: `@rid`

Mark the field as the record identity or concepts/basics.pdf.

Example:

```
{ bookById(id: "book-1")
  {
    rid @rid
    id
    name
    authors {
      firstName
      lastName
    }
  }
}
```

@skip and @include

Applies to: `Query Field`, fragment spread and inline fragment

Syntax: `@skip(if: <Boolean>)`, `@include(if: <Boolean>)`

The two built-in directives of the GraphQL specification. The selection they are written on is left out of the result when the `@skip` condition is `true`, or when the `@include` condition is `false`; written together, the selection is returned only when both let it through. The condition is a literal or a variable, so the same query text can toggle parts of the result at execution time:

```
fragment Details on Book { pageCount authors { lastName } }

query($brief: Boolean!, $withName: Boolean = true) {
  bookById(id: "book-1") {
    id
    name @include(if: $withName)
    ...Details @skip(if: $brief)
  }
}
```

The `if` argument is required and must be a Boolean: a missing, repeated or non-Boolean `if`, an argument other than `if`, the same directive written twice on one selection, or either directive written on a fragment definition rather than on its spread, is rejected before any record is read.

When the only field of the query is left out, the result has no record.

Chapter 18. MongoDB QL

18.1. MongoDB Query Language

ArcadeDB provides support for both MongoDB Query Language and MongoDB protocol.

If you're using ArcadeDB as embedded, please add the dependency to the `arcadedb-mongodbw` library. If you're using Maven include this dependency in your `pom.xml` file.

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-mongodbw</artifactId>
    <version>26.11.1</version>
</dependency>
```

18.1.1. MongoDB Query Language

If you want to use MongoDB Query Language from Java API, you can simply keep the relevant jars in your classpath and execute a query or a command with "mongo" as language.

Example:

```
// CREATE A NEW DATABASE
Database database = new DatabaseFactory("heroes").create();

// CREATE THE DOCUMENT TYPE 'HEROES'
database.getSchema().createDocumentType("Heroes");

// CREATE A NEW DOCUMENT
database.transaction((tx) -> {
  database.newDocument("Heroes").set("name", "Jay").set("lastName", "Miner").set("id",
i).save();
});

// EXECUTE A QUERY USING MONGO AS QUERY LANGUAGE
for (ResultSet resultset = database.query("mongo", // <-- USE 'mongo' INSTEAD OF 'sql'
    "{ collection: 'Heroes', query: { $and: [ { name: { $eq: 'Jay' } }, { lastName: {
$exists: true } }, { lastName: { $eq: 'Miner' } }, { lastName: { $ne: 'Miner22' } } ],
$orderBy: { id: 1 } } }"); resultset.hasNext(); ++i) {
  Result doc = resultset.next();
  ...
}
```

For more information on the MongoDB query language see: MongoDB CRUD Operations

Mongo queries through Postgres Driver

You can execute a Mongo query against ArcadeDB server by using the Postgres driver and prefixing the query with `{mongo}`. Example:

```
"{mongo} { collection: 'Heroes', query: { $and: [ { name: { $eq: 'Jay' } }, {
lastName: { $exists: true } }, { lastName: { $eq: 'Miner' } } ] } }"
```

ArcadeDB server will execute the query `{ collection: 'Heroes', query: { $and: [ { name: { $eq: 'Jay' } }, { lastName: { $exists: true } }, { lastName: { $eq: 'Miner' } } ] } }` using the Mongo query language.

Mongo queries through HTTP/JSON

You can execute a Mongo query against ArcadeDB server by using HTTP/JSON API. Example of executing an idempotent query with HTTP GET command:

```
curl "http://localhost:2480/query/graph/mongo/{ collection: 'Heroes', query: { $and: [
{ name: { $eq: 'Jay' } }, { lastName: { $exists: true } }, { lastName: { $eq: 'Miner'
} } ]} }"
```

You can also execute the same query in HTTP POST, passing the language and query in payload:

```
curl -X POST "http://localhost:2480/query/graph" -d "{'language': 'mongo', 'command':
'{ collection: \"Heroes\", query: { $and: [ { name: { $eq: \"Jay\" } }, { lastName: {
$exists: true } }, { lastName: { $eq: \"Miner\" } } ] } }\"}"
```

18.1.2. MongoDB Protocol Plugin

If your application is written for MongoDB and you'd like to run it with ArcadeDB instead, you can simply replace the MongoDB server with ArcadeDB server with the MongoDB Plugin installed. This plugin supports MongoDB BSON Network protocol. In this way you can use any MongoDB driver for any supported programming language.

ArcadeDB Server supports a subset of the MongoDB protocol, like CRUD operations and queries.

To start the MongoDB plugin, enlist it in the `server.plugins` settings. To specify multiple plugins, use the comma `,` as separator.

Example to start ArcadeDB with the MongoDB Plugin:

```
~/arcadedb $ bin/server.sh -Darcadedb.server.plugins
="MongoDB:com.arcadedb.mongo.MongoDBProtocolPlugin"
```

If you're using MS Windows OS, replace `server.sh` with `server.bat`.

In case you're running ArcadeDB with Docker, use `-e` to pass settings (Port 27017 is the default MongoDB binary port):

```
docker run --rm -p 2480:2480 -p27017:27017 \
       --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata \
          -Darcadedb.server.plugins=MongoDB:com.arcadedb.mongo.MongoDBProtocolPlugin "
\
          arcadedata/arcadedb:latest
```

The Server output will contain this line:

```
2018-10-09 18:47:01:692 INFO  <ArcadeDB_0> - MongoDB Protocol plugin started
[ArcadeDBServer]
```

Supported commands

The plugin implements a subset of the MongoDB protocol, focused on CRUD operations, queries and index creation:

Command	Driver helpers
`find`, `count`, `aggregate`	`find()`, `count()`, `estimatedDocumentCount()`, `countDocuments()`, `aggregate()`
`insert`	`insertOne()`, `insertMany()`
`update`	`updateOne()`, `updateMany()`, `replaceOne()`, `replaceMany()`
`delete`	`deleteOne()`, `deleteMany()`
`create`	`createCollection()`
`createIndexes`	`createIndex()`, `createIndexes()`
`getMore`, `killCursors`	iterating a `find()` cursor past its first batch, `cursor.close()`

Any other command is refused with error `59` (`CommandNotFound`), what MongoDB answers for a command it does not have, so a driver can tell the command is missing *(since v26.12.1)*.

Updating and deleting documents

The `update` and `delete` commands accept the same filter operators as `find` (`$eq`, `$ne`, `$lt`, `$lte`, `$gt`, `$gte`, `$in`, `$nin`, `$and`, `$or`, `$not`, `$exists`, …).

```
// Delete the first matching document
db.doc.deleteOne({ test: { $eq: "abcde" } })

// Delete all matching documents
db.doc.deleteMany({ counter: { $gte: 5 } })

// Replace the whole document (replaceOne / replaceMany)
db.doc.replaceOne({ test: { $eq: "abcde" } }, { test: "vwxyz" })
```

```
// Update with operators ($set, $unset, $inc)
db.doc.updateOne({ test: "v2" }, { $set: { name: "Jay" } })
db.doc.updateMany({ counter: { $gte: 8 } }, { $inc: { counter: 100 } })
db.doc.updateOne({ test: "v1" }, { $unset: { counter: "" } })

// Upsert: insert when nothing matches
db.doc.updateOne({ test: "missing" }, { $set: { counter: 999 } }, { upsert: true })
```

Notes:

- `deleteOne()`, `updateOne()` and `replaceOne()` affect a single matching document; `deleteMany()` and `updateMany()` affect all matches.
- A replacement document (no `$`-prefixed keys) fully replaces the matched document's fields, but keeps its `_id`. The `_id` is immutable: an update or replacement that changes or removes it is refused (error 66).
- The supported update operators are `$set`, `$unset` and `$inc`. They accept dotted paths (`{ $set: { "addr.city": "Rome" } }`): the embedded document is updated and missing levels are created. Writing below a value that is not a document (for example a string) is refused. Another MongoDB update operator (such as `$push`, `$addToSet` or `$setOnInsert`) or a pipeline-style update is refused with error `238` (`NotImplemented`) naming it, and an operator that MongoDB does not have with error `9` (`FailedToParse`), whether or not a document matches *(since v26.12.1)*.
- An update is refused, and changes nothing, in the same cases as MongoDB, with or without a filter *(since v26.12.1)*: `$inc` on a field that is not a number (a string, boolean, array, date, document or `null`) or by an amount that is not a number (TypeMismatch, 14), `$inc` past the range of a 64-bit integer (BadValue, 2), and two operators on the same field or on a field and one below it, such as `{ $set: { s: 2 }, $unset: { s: "" } }` (ConflictingUpdateOperators, 40).
- With `upsert: true`, when no document matches a new one is inserted, seeded from the filter's equality conditions plus the update. If the filter includes an `_id`, that `_id` is used for the new document; otherwise one is generated. Either way, the resulting `_id` is returned.
- The seed follows MongoDB's rules *(since v26.12.1)*: it takes each equality at the top level of the filter, under `$and` and under an `$or` with a single branch, where an equality is a plain value (including an embedded document or an array, but not a regular expression), `$eq`, an `$in` with one value or an `$all` with one value. A dotted path creates the embedded documents on its way: `{ "a.b": 5 }` inserts `{ a: { b: 5 } }`, so running the same upsert again matches it. A replacement upsert (`replaceOne`) takes only the `_id` from the filter. The `_id` is stored as the first field. A filter that names the same path twice (`{ $and: [{ a: 1 }, { a: 2 }] }`), a path and one below it (`{ a: 1, "a.b": 2 }`) or an `$all` with several values cannot seed a document: when nothing matches, that statement is refused with error `54` (`NotSingleValueField`) as a write error, while a matching document is still updated.
- The counts in the reply are what the database actually wrote. A write refused by a `BeforeRecordCreate/Update/DeleteListener` (see Events) is not counted: `delete` and `insert` count only the documents removed or stored, `update` reports it in `n` (matched) but not in `nModified` (changed), and an upsert whose match was refused does not insert a new document *(since v26.10.2)*.
- `nModified` counts only the documents an update changed, as in MongoDB: an update that leaves

a document as it was (`$set` to the current value, `$inc` by 0, `$unset` of a missing field, an identical replacement) counts in `n` but not in `nModified`, and the document is not rewritten. Reordering the fields of an embedded document is a change *(since v26.12.1)*.

- A document refused on its own is reported as a write error of that document, and the other documents of the command are kept, as in MongoDB *(since v26.12.1)*. This covers a duplicate key (`11000`), a value a schema constraint refuses (`121`, `DocumentValidationFailure`, for example a `max` declared on a property), and an update the document cannot take (such as writing below a string). With `ordered: true` (the default) the documents or update statements after the refused one are not applied, and with `ordered: false` they are. The Java driver throws `MongoWriteException` or `MongoBulkWriteException` for these. In production mode the error text does not include the stored values.

Null values, projections, regular expressions and duplicate keys

- `$ne`, `$nin`, `$in` (with `null`) and `$not` also match documents whose field is `null` or missing, like MongoDB. For example `{ age: { $ne: 30 } }` returns documents with no `age`.
- Filters follow MongoDB's rules for types and arrays, in `find`, `count`, `update` and `delete`. A number never equals a string or a boolean (`{ n: 5 }` does not match `"5"`), and `$gt`/`$lt` compare within one type only. A filter on an array field tests its elements: `{ tags: "a" }` matches `["a", "b"]`, `{ tags: { $ne: "a" } }` does not, and `$all`, `$elemMatch`, `$size` and `tags.0` work. `$exists` works on dotted paths (`{ "a.b": { $exists: false } }`), and collection names with dots (`fs.files`) can be read like any other.
- A filter on `_id` (equality or `$in`) uses the unique `_id` index. A filter on any other field is tested on the documents with MongoDB's own rules, and an index narrows which documents are tested only for a field that an index starts with and whose type is declared in the schema as a scalar (`STRING`, `BOOLEAN`, `BYTE`, `SHORT`, `INTEGER`, `LONG` or `DOUBLE`; for example `CREATE PROPERTY products.sku STRING` followed by `CREATE INDEX ON products (sku) UNIQUE`). Equality and `$in` use the index, and so does a range (`$gt`, `$gte`, `$lt`, `$lte`) on an integer field, when the operand is of the kind of the field (a string for a `STRING`, an integer for an `INTEGER`). Lookups, updates, deletes and bulk upserts by such a field do not read the whole collection. A field the schema does not declare is never narrowed by its index, because an array can be stored in it and an index does not see the elements of an array: filters on it, and on dotted paths, negations, regular expressions and temporal values, read every document of the collection and are slower on large ones. The declared type is relied upon: when you declare a property on a collection that already holds data, records stored before the declaration are not converted, so rewrite them if they hold an array or a value of another kind in that field.
- A top-level `$not`, `$expr` and a `$regex` directly inside `$elemMatch` are refused with an error. Regular expressions are time-bounded by `arcadedb.command.regexTimeout`, shared by all the documents one command scans: raise it if a big scan with a regular expression times out. The same bound applies to the `$match` stages of an aggregation, including `countDocuments()` with a filter, and to the `$match` stages nested in `$facet`, `$lookup` and `$unionWith`. A non-regex `$expr` works inside an aggregation `$match`.
- Aggregating or counting a collection that does not exist returns an empty result (`count()` and `estimatedDocumentCount()` return `0` *(since v26.12.1)*), and the first insert creates the collection. A `count` with a `skip` or `limit` that is not a number is refused with error `14` (`TypeMismatch`).
- `find()` honors the projection (`{ name: 1 }`, `{ id: 0 }`, `{ age: 0 }`, dotted paths). Inclusion and

exclusion cannot be mixed, except for _id. A dotted path goes into arrays like in MongoDB: { "items.name": 1 } returns the name of every element of items, and { "items.price": 0 } removes price from every element. An inclusion returns the fields in the order of the stored document. Overlapping paths ({ a: 1, "a.b": 1 }) are refused _(since v26.12.1)_. Projection operators ($slice, $elemMatch, $meta, the positional $) are not supported and return an error. Documents never contain ArcadeDB's @rid, @type or @cat.

- find().sort() follows MongoDB's order *(since v26.12.1)*. Values of different types are ordered by type (null, numbers, strings, documents, arrays, ObjectId, booleans, dates), numbers by value whatever their type, and strings by character code. An array sorts by its smallest element in an ascending sort and by its largest in a descending one, and an empty array sorts before null. skip and limit page through that order. Sorting is done in memory: with a limit only the documents needed are kept, without one every matching document is.
- A MAP property whose keys are not strings (written through SQL or the HTTP API, for example {1: "one"}) is returned with each key as a field named by its text ("1", and "null" for a null key), as the HTTP API shows it. A dotted path such as { $set: { "m.1": "uno" } } updates that same key and keeps its type *(since v26.10.2)*.
- Regular expressions work as literals (/^a/i) and with $regex and $options (i, m, s, x), also under $not. An invalid pattern returns an error.
- _id is unique: inserting an existing _id fails with E11000. In a batch, ordered: true (the default) stops at the first duplicate, ordered: false inserts the others. The first insert creates the collection and a unique index on _id; on a big existing collection that first insert builds the index and other writes to it wait until it is done. A collection written by a version before v26.12.1 is upgraded the same way on its first insert or upsert: a one-time pass over its documents, logged at INFO.
- id values of different types are different ids, as in MongoDB _(since v26.12.1)_: 1 and "1", true and "true", 1.5 and "1.5" can all be stored in one collection. Numbers compare by value whatever their type, so 1, NumberLong(1), 1.0 and NumberDecimal("1.00") are one _id. The index holds the type and the value of each _id as text: an equality or $in on _id uses it, a sort or a range on _id is answered by reading the documents.
- Only the documents written through the MongoDB protocol are covered by its unique indexes. A record written through SQL, the HTTP API, Studio, Gremlin or an importer is not checked for uniqueness, and once the collection has its _id index a filter on _id does not find it (#12592).

An ObjectId and a String with the same 24 hex digits are two different id values, as in MongoDB: both can be stored in one collection, and each one is found only by a filter of its own type _(since v26.12.1)_. A String _id in that shape stored by an older version cannot be told apart from an ObjectId, so it keeps reading back as an ObjectId and is found by an ObjectId filter.

Reading find() results in batches

Since v26.12.1 find() answers from a cursor, like MongoDB, instead of reading and sending the whole result in one reply. The first reply carries 101 documents (or batchSize), and the driver asks for the rest with getMore while the application iterates, each batch holding batchSize documents, or as many as fit in 16 MB when no batchSize is given. A document is read and converted only when its batch is built, so the first document no longer waits for the last one, a client that stops early does

not pay for the whole result, and a result larger than the 48 MB a driver accepts in one message can be read.

- `skip` and `limit` apply across all the batches. `singleBatch: true`, or a negative `limit`, answers one batch and leaves no cursor. A negative `batchSize` is refused with error `2` (`BadValue`), and one that is not a number with error `14` (`TypeMismatch`).
- A cursor is closed when it is read to the end, by `killCursors` (`cursor.close()`), when the collection or the database is dropped, and after 10 minutes without a `getMore`, unless it was opened with `noCursorTimeout`. A database keeps at most 1024 open cursors: past that, the cursor idle for the longest among those of the user holding the most is closed to make room.
- A cursor is not tied to the connection that opened it, so a driver can read the next batch on another connection of its pool. It can only be read by the user that opened it: for another user it answers `CursorNotFound`, and a `getMore` that names another collection is refused with error `13` (`Unauthorized`).
- The `resultSetLimit` of the user's group bounds the documents of all the batches together.
- A sorted `find()` still reads every matching document before the first batch, since it needs them all to order them.

BSON value types

Besides strings, numbers, booleans, dates, embedded documents and arrays, the plugin stores and returns these BSON types with their type kept: binary, regular expression, timestamp, MinKey, MaxKey, JavaScript code, Decimal128 and ObjectId. A value the server cannot store is refused with an error instead of being dropped.

- Decimal128 is stored as an exact decimal. A negative zero (`-0`, `-0.00`), `NaN` and the infinities are kept as they are *(since v26.12.1)*. A decimal with more than 34 digits, written through SQL, is returned as a double.
- An ObjectId outside `id` is stored as the text `$oid:<24 hex digits>` and returned as an ObjectId. An ObjectId filter matches only ObjectIds: plain hex text, including the one stored by versions before v26.10.1, is a String and is matched by a String filter _(since v26.12.1). `id` keeps the plain hex form. Only the _id of the document itself reads back as an ObjectId: an _id field inside an embedded document is returned as it was written _(since v26.12.1).
- Binary, timestamp, regular expression, MinKey, MaxKey, JavaScript and the special Decimal128 values are stored as a small document with the reserved field `$bson`. MongoDB filters and sorts compare them in BSON order; a SQL query sees them as documents.
- Reserved names: a field called `$bson` is refused, so is a field name starting with `$unique:` (error `2`, `BadValue`, also in an index specification) *(since v26.12.1)*, and a text starting with `$oid:` or `$str:` is escaped on write, so it comes back unchanged.
- The Symbol type and binary subtypes other than 0 and 4 are not supported by the bundled MongoDB library.

Authentication

The MongoDB Protocol Plugin supports authentication through the SASL `PLAIN` mechanism (RFC

4616). Credentials are validated against the ArcadeDB server users, exactly like the PostgreSQL protocol.

Because ArcadeDB stores passwords as one-way hashes, the challenge-response mechanisms `SCRAM-SHA-1` and `SCRAM-SHA-256` (the default negotiated by most MongoDB drivers and by `mongosh`) cannot be supported: they would require the clear-text password to be available on the server side. For this reason the client must explicitly request the `PLAIN` mechanism.

Example using a connection string:

```
mongosh "mongodb://root:playwithdata@localhost:27017/mydb?authMechanism=PLAIN"
```

Or, from an already opened shell:

```
db.auth({ user: "root", pwd: "playwithdata", mechanism: "PLAIN" })
```

The `user` and `pwd` are the ArcadeDB server credentials, and the authentication database is checked against the databases the user is authorized to access.

The `PLAIN` mechanism transmits the credentials in clear text. In production always use it over a TLS-encrypted connection, the same recommendation that applies to the PostgreSQL wire protocol.

Index management

The MongoDB Protocol Plugin supports the `createIndexes` command, used by the driver helpers `createIndex()` and `createIndexes()`. Each requested index is mapped to an ArcadeDB LSM-Tree index over the same property list.

```
// Single-field index
db.doc.createIndex({ test: 1 })

// Unique index
db.doc.createIndex({ email: 1 }, { unique: true })

// Compound index
db.doc.createIndex({ lastName: 1, firstName: -1 })
```

Notes:

- If the target collection does not exist yet, it is created automatically (the response reports `createdCollectionAutomatically: true`).
- Because MongoDB collections are schemaless, properties that are not yet declared on the type are kept free-form and a non-unique index serializes their keys as strings, preserving the stored value type.
- A unique index (`unique: true`) follows MongoDB's rules *(since v26.12.1)*: values of different types

are different keys (`1` and `"1"`, `true` and `"true"`), numbers compare by value whatever their type, and a missing field counts as `null`, so a second document without the field, or with `null`, is refused with `E11000`. Creating a unique index on a collection where two documents lack the field therefore fails. The index is built on a hidden property, `$unique:<field>[,<field>…]`, holding the type and the value of the fields; the documents already in the collection get it while the index is built, and the documents do not show it. Unlike MongoDB, an array is one key rather than one key per element: `{ a: [1, 2] }` and `{ a: 2 }` can both be stored, two identical arrays cannot.

- The sort direction (`1` ascending, `-1` descending) is accepted for compatibility but ignored: ArcadeDB LSM-Tree indexes can be scanned in both directions.
- Re-creating an existing index is a no-op, matching MongoDB semantics.
- Specialized index types (`text`, `2dsphere`, `hashed`, …) are created as standard LSM-Tree indexes on the indexed fields.

Listing and terminating running commands

Since v26.10.2 the commands a MongoDB client sends are listed, while they run, by the HTTP `list queries` server command and by Cypher `SHOW TRANSACTIONS`, with `protocol mongodb`, the connection's channel id as `connectionId` and the client's address. `terminate query`, `TERMINATE TRANSACTIONS` or the Studio Running Queries tab stop one: the client receives error `11601` (`Interrupted`), the code MongoDB answers for an operation that `killOp` interrupted. **root** sees and stops every command, any other user only their own.

Writes refused by an HA replica

Since v26.10.2 a write that an HA replica refuses and that names the leader fails with error `10107` (`NotWritablePrimary`), the code MongoDB answers a write sent to a secondary with. The `errmsg` contains `not primary`, which older drivers match on, followed by `The leader is server '<address>'`. A driver answers the code by marking the server unknown and rescanning the topology. During an election, when no leader is known yet, the refusal names nobody and keeps the retryable `112` (`WriteConflict`). See how-to/operations/ha.pdf.

Chapter 19. Redis

19.1. Redis Protocol

ArcadeDB Server supports a subset of the Redis protocol. Please open an issue or a discussion on GitHub to support more commands.

If you're using ArcadeDB as embedded, please add the dependency to the `arcadedb-redisw` library. If you're using Maven include this dependency in your `pom.xml` file.

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-redisw</artifactId>
    <version>26.11.1</version>
</dependency>
```

ArcadeDB Redis plugin works in 2 ways:

- Manage **transient** (non-persistent) entries in the server. This is useful to manage user sessions and other records you don't need to store in the database.
- Manage **persistent** entries in the database. You can save and read any documents, vertices and edges from the underlying database.

19.1.1. Installation

To start the Redis plugin, enlist it in the `server.plugins` settings. To specify multiple plugins, use the comma `,` as separator. Example:

```
~/arcadedb $ bin/server.sh -Darcadedb.server.plugins
="Redis:com.arcadedb.redis.RedisProtocolPlugin"
```

If you're using MS Windows OS, replace `server.sh` with `server.bat`.

In case you're running ArcadeDB with Docker, open the port `6379` and use `-e` to pass settings:

```
docker run --rm -p 2480:2480 -p 6379:6379 \
        --env ARCADEDB_SETTINGS="-Darcadedb.server.rootPassword=playwithdata \
           -Darcadedb.server.plugins=Redis:com.arcadedb.redis.RedisProtocolPlugin " \
           arcadedata/arcadedb:latest
```

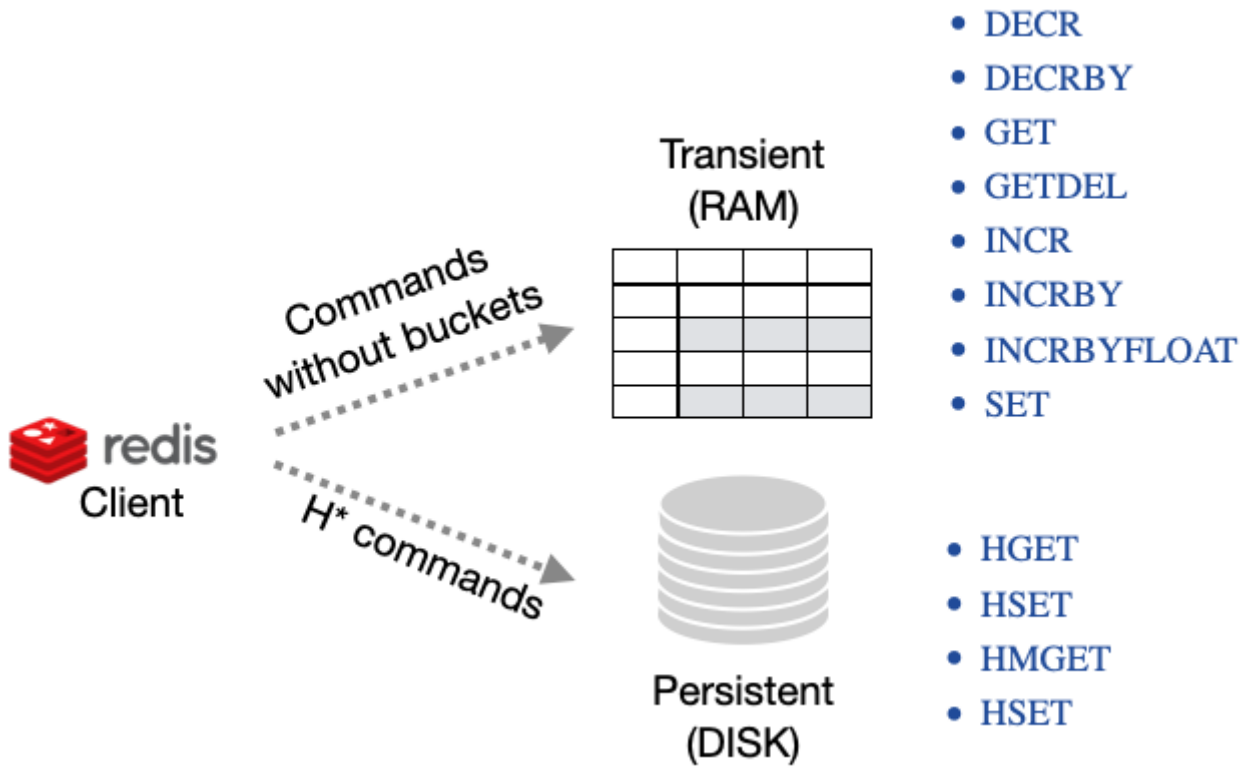
The Server output will contain this line:

```
2018-10-09 18:47:58:395 INFO  <ArcadeDB_0> - Redis Protocol plugin started
[ArcadeDBServer]
```

19.1.2. How it works

ArcadeDB works in 2 ways with the Redis protocol:

- **Transient commands**, key/value pairs saved will be not saved in the database. This is perfect to store transient data, like user sessions.
- **Persistent commands**, key/value pairs allows to store and retrieve ArcadeDB documents, vertices and edges



19.1.3. Transient (RAM Only) Commands

Below you can find the supported commands. The link takes you to the official Redis documentation. Please open an issue or a discussion on GitHub to support more commands.

The following commands do not take the bucket as a parameter because they work only in RAM on a shared (thread-safe) hashmap. This means all the stored values are reset when the server restarts.

`INCR`, `DECR`, `INCRBY` and `INCRBYFLOAT` are atomic, as they are in Redis: concurrent clients incrementing the same key all count, and no update is lost. The guarantee is **per server**, though — these values are not replicated, so in an HA cluster two clients incrementing through different nodes each update their own node's copy. The same applies to `SET` with `NX`/`XX`: it is a genuine check-and-set on one server, not a cluster-wide lock.

The commands check their argument count, over the Redis wire protocol and in the `redis` query language alike, and answer `wrong number of arguments for '<command>' command` when it is wrong, as Redis does: `INCR` and `DECR` take the key only, `INCRBY`, `DECRBY` and `INCRBYFLOAT` take the key and the amount (there is no default amount of 1), and a surplus argument is refused, never read as an amount. Amounts and stored values of `INCR`, `INCRBY`, `DECR` and `DECRBY` must be plain 64-bit integers (

`+5`, `05` and `-0` are refused with `value is not an integer or out of range`). `INCRBYFLOAT` replies and stores the result as decimal text, as Redis does (`0.1` plus `0.2` is `0.3`, an integral result is `3`, not `3.0`), so `INCR` works on an integral result. In the `redis` query language `GET` therefore returns a string for a key written by `INCRBYFLOAT`.

In the `redis` query language the hash commands take the count of the form they are written in: `HGET` and `HEXISTS` take exactly one RID (`HGET #1:0`) or an index and one key (`HGET Account[id] 1`), `HMGET` and `HDEL` one or more RIDs, or an index and one or more keys. `HGET Account[id] 1 2` is refused (use `HMGET`) *(since v26.12.1: a surplus argument was ignored)*.

`HELLO [protover [AUTH username password] [SETNAME clientname]]` reads its words as Redis does: a `protover` that is not an integer answers `ERR Protocol version is not an integer or out of range`, an integer other than 2 or 3 `NOPROTO unsupported protocol version`, and any word after it that is not a complete `AUTH` or `SETNAME` option answers `ERR Syntax error in HELLO option '<word>'`. Nothing is authenticated by a malformed `HELLO`. The client name is accepted and checked as Redis checks it (no spaces or special characters) but not used. A command the server does not know answers `ERR unknown command '<name>'`. *(Since v26.12.1: an unknown option or a word that was no version was ignored, an out-of-range version answered the text of a Java exception, and an unknown command answered* `-Command not found`*.)*

Available transient commands (in alphabetic order):

- DECR, Decrement a value by 1
- DECRBY, Decrement a value by a specific amount (64-bit precision)
- EXISTS, Check if key exists
- GET, Return the value associated with a key
- GETDEL, Remove and return the value associated with a key
- INCR, Increment a value by 1
- INCRBY, Increment a value by a specific amount (64-bit precision)
- INCRBYFLOAT, Increment a value by a specific amount expresses as a float (64-bit precision)
- SET, Set a value associated with a key. The value is binary-safe: any bytes, valid UTF-8 or not, are returned unchanged by `GET`. Keys are binary-safe too (see Keys); the other arguments of the commands (amounts, options, JSON documents, type and index names) are text. `NX` (only set if the key does not exist), `XX` (only set if the key already exists) and `GET` (return the previous value instead of `OK`) are supported. `EX`/`PX`/`EXAT`/`PXAT`/`KEEPTTL` are rejected with an error rather than silently ignored, because transient keys in ArcadeDB have no expiry.

Keys

A key written as `<database>.<key>`, or a plain key after `SELECT <database>` (or with `arcadedb.redis.defaultDatabase` set), lives in that database's global variables, which SQL reads as `$<key>`. A plain key with no database lives in the connection's own map. Since v26.11.1 a key is the client's key in both places, as in Redis:

- every byte of a key counts, valid UTF-8 or not, so two binary keys that differ only in a byte that is not valid UTF-8 are two keys. Before, every such byte was replaced by the same character, so

SET of one key overwrote the other and GETDEL of one deleted both;

- $k and k are two keys, and the names SQL reserves for its context (parent, current, depth, path, stack) are ordinary keys. Before, the leading $ was stripped, so $k was the key k, and the reserved names were refused in a database's key space while the connection's own map took them.

A key SQL can name is still visible to it as $<key>. A reserved name stays SQL's own: $parent or $depth in a SQL statement never reads a Redis key of that name.

19.1.4. Persistent Commands

The following commands act on persistent buckets in the database. Records (documents, vertices and edges) are always in form of JSON embedded in strings. The bucket name is mapped as the database name first, then type, the index or the record's RID based on the use case. An index must exist on the property you used to retrieve the document, otherwise an error is returned.

For the sake of this tutorial, in a database MyDatabase, we're going to create the account document type totally schemaless but for some indexed properties: id as a unique long, email as a unique string and the pair firstName and lastName both strings and indexed with a composite key:

```
CREATE DOCUMENT TYPE Account;

CREATE PROPERTY Account.id LONG;
CREATE INDEX ON Account (id) UNIQUE;

CREATE PROPERTY Account.email STRING;
CREATE INDEX ON Account (email) UNIQUE;

CREATE PROPERTY Account.firstName STRING;
CREATE PROPERTY Account.lastName STRING;
CREATE INDEX ON Account (firstName,lastName) UNIQUE;
```

You can run the following Redis commands, for example, using the redis-cli tool; under Debian / Ubuntu this is part of the redis-tools package.

Now you can create a new document with Redis protocol and the HSET Redis command:

```
HSET MyDatabase Account
"{'id':123,'email':'jay.miner@commodore.com','firstName':'Jay','lastName':'Miner'}"
```

The database and the type are two separate arguments: HSET <database> <type> <json>. Writing them as one argument (HSET MyDatabase.Account <json>) makes the plugin look for a database named MyDatabase.Account and fails with a "database does not exist" error.

To retrieve the document inserted above by id (O(logN) complexity), you can use the HGET Redis

command:

```
HGET MyDatabase.Account[id] 123
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
```

To retrieve the same document by `email` (O(logN) complexity), you can use the HGET Redis command:

```
HGET MyDatabase.Account[email] "jay.miner@commodore.com"
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
```

To retrieve the same document by the pair `firstName` and `lastName` (O(logN) complexity), we are going to use the composite key we created before:

```
HGET MyDatabase.Account[firstName,lastName] "[\"Jay\",\"Miner\"]"
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
```

To retrieve the document inserted above by it `RID` (O(1) complexity), you can use the HGET Redis command:

```
HGET MyDatabase "#1:0"
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
```

You can also get multiple record in one call by using the HMGET Redis command:

```
HMGET MyDatabase "#1:0" "#1:1" "#1:2"
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
"{'@rid':'#1:1','@type':'Account','id':232,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
"{'@rid':'#1:2','@type':'Account','id':12,'email':'jay.miner@commodore.com','firstName
':'Jay','lastName':'Miner'}"
```

Or the same, but by a key:

```
HMGET MyDatabase.Account[id] 123 232 12
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
"{'@rid':'#1:1','@type':'Account','id':232,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
```

```
"{'@rid':'#1:2','@type':'Account','id':12,'email':'jay.miner@commodore.com','firstName
':'Jay','lastName':'Miner'}"
```

Or by a composite key, using the same `[…]` JSON array form `HGET` takes. A key with no record keeps its place in the reply as a null entry, so the Nth reply is always the answer to the Nth key:

```
HMGET MyDatabase.Account[firstName,lastName] "[\"Jay\",\"Miner\"]"
"[\"Nobody\",\"Here\"]"
"{'@rid':'#1:0','@type':'Account','id':123,'email':'jay.miner@commodore.com','firstNam
e':'Jay','lastName':'Miner'}"
(nil)
```

A `RID` that names no record is answered as "not there" by every command that takes one, on the Redis protocol as in the `redis` query language: `HGET` replies `(nil)` (no result in the query language), `HEXISTS` `0`, `HMGET` a null entry and `HDEL` does not count it. *(Since v26.11.1:* `HGET` *and* `HEXISTS` *replied* `ERR Record ⋯ not found`*, and* `HGET` *in the query language failed with* `RecordNotFoundException`*.)*

A key is matched against the index exactly as you wrote it, then converted to the indexed property's declared type. A key is therefore never guessed to be a number just because it looks like one: on a `STRING` property, `007` finds the record whose value is the three characters `007`, not the one whose value is `7`.

While the index a lookup uses is being created, rebuilt or dropped, `HGET`, `HMGET`, `HEXISTS` and `HDEL` never answer from a partial index: after a short retry they reply `-TRYAGAIN`, so a `(nil)` always means the key is not there. Retry the command once the index change is over *(since v26.10.2)*.

To delete the document inserted above by `email`, you can use the HDEL Redis command:

```
HDEL MyDatabase.Account[email] "jay.miner@commodore.com"
:1
```

The number `HDEL` replies is the number of records it removed. A key with no record is not counted, and neither is a delete the database refused because a `BeforeRecordDeleteListener` vetoed it. Likewise the number the persistent `HSET` replies is the number of records it stored: a create a `BeforeRecordCreateListener` vetoed is not counted. A multi-key `HDEL` or `HSET` is one unit of work, on the Redis protocol as in the `redis` query language: if it fails on one key, no earlier key of the same command stays deleted or stored, and a conflict is retried. The transient form of `HSET` checks every JSON object before it stores the first *(since v26.10.2)*.

The returning JSON could have a different ordering of the properties from the one you have inserted. This is because JSON doesn't maintain the order of properties,

> but only of arrays (`[]`).

Available persistent commands (in alphabetic order):

- HDEL, Delete one or more records by a key, a composite key or record's id. `HDEL <database> <rid> [<rid> ⋯]` deletes those records and replies with how many it deleted
- HEXISTS, Check if a key exists
- HGET, Retrieve a record by a key, a composite key or record's id
- HMGET, Retrieve multiple records by a key, a composite key or record's id
- HSET, Create and update one or more records by a key, a composite key or record's id

Available miscellaneous commands:

- PING, Returns its argument (for testing server readiness or latency)

Settings

To change the host where the Redis protocol is listening, set the setting `arcadedb.redis.host`. A name that resolves to several local addresses, such as `localhost` (`127.0.0.1` and `::1`), is bound on every one of them. By default, is `0.0.0.0` which means listen to all the configured network interfaces. To change the default port (6379) set `arcadedb.redis.port`.

Listing and terminating running commands

Since v26.10.2 every command a Redis client sends, except `PING` and `SELECT`, is listed while it runs by the HTTP `list queries` server command and by Cypher `SHOW TRANSACTIONS`, with `protocol redis`, the command line as its text, the connection's name as `connectionId` and the client's address. `terminate query`, `TERMINATE TRANSACTIONS` or the Studio Running Queries tab stop it, and the client receives an `ERR` error saying that it was terminated. **root** sees and stops every command, any other user only their own.

Since v26.11.1 the commands on keys held in memory (`GET`, `SET`, `GETDEL`, `EXISTS`, `INCR`, `INCRBY`, `INCRBYFLOAT`, `DECR`, `DECRBY`) are neither listed nor held by the query admission control: each is a single in-memory map operation, done before a listing could show it. They are no longer refused with `TRYAGAIN` when the server is busy. The commands that read or write records (`HSET`, `HGET`, `HMGET`, `HEXISTS`, `HDEL`) are still listed and gated.

Malformed messages and inline commands

(Since v26.11.1) A malformed message gets one `-ERR Protocol error` reply and one WARNING in the log, and the connection is closed. Before, a client that had not authenticated could make the server log one or two SEVERE records per junk byte.

- A command that does not start with a RESP type marker is read as an inline command, as Redis reads it, so `PING` typed in `nc` or `telnet`, or sent by a health probe, is answered. Quotes and the escapes `\n`, `\r`, `\t`, `\b`, `\a` and `\xHH` are honored, the line is capped at 64 KB, and the argument count at `arcadedb.redis.maxMultiBulkLength`. A blank line gets no reply, and an HTTP request sent to the Redis port is refused at its first line.

- A top-level simple string, integer or bulk string (`+`, `:` or `$`) is refused with `-ERR Protocol error` and the connection is closed. This is stricter than Redis, which reads it as an inline command.
- An array whose first element is not a string (`*1\r\n:1\r\n`) is answered with `-ERR Protocol error: expected '$', got ':'` (`'*'` for a nested array, `the command name is a null bulk string` for `$-1`), and the connection stays open. Redis sends the same error and then closes the connection.

Pipelining

Since v26.11.1 the replies to pipelined commands are sent together, when the client has no more commands waiting, instead of one network write per reply. A client that pipelines (most Redis clients can) gets several times the throughput of one that waits for each reply.

Writes refused by an HA replica

Since v26.10.2 a write that an HA replica refuses and that names the leader is answered with a `READONLY` error, the reply Redis itself gives to a write sent to a replica, and the text ends with `The leader is server '<address>'`. A Sentinel-aware client reconnects to the leader on it. During an election, when no leader is known yet, the refusal names nobody and keeps the retryable `TRYAGAIN`. See how-to/operations/ha.pdf.

A database that is not available

Since v26.11.1 a command that names a database an `overwrite` restore is about to replace, or that a `close database` command closed, is answered with a `LOADING` error, the reply Redis itself gives while it holds a dataset it cannot serve yet, and the client retries later. Before, it was answered with a generic `ERR` as for a server fault. A key written as `<database>.<key>`, and `SELECT <database>`, are refused the same way: the key used to be kept as a plain key of the connection and acknowledged with `OK`, and the value was missing from the database once it was available again. A name that is not a database keeps its meaning, a plain key for `<name>.<key>` and `Database '<name>' not found` for `SELECT`. See reference/http-api/http.pdf.

Since v26.11.1 a database that is on disk and not open yet, which with `arcadedb.server.loadAtStartup=false` is every database until its first request, is a database too: `<database>.<key>` and `SELECT <database>` open it. Before, its key was kept as a plain key of the connection and acknowledged with `OK`, and `SELECT` answered `Database '<name>' not found`. A database an interrupted HA snapshot install left pending is answered with `LOADING` until the install is reconciled. A directory of the database directory that holds no database is not one, whatever its name.

19.1.5. Redis via HTTP API

In addition to the native Redis wire protocol, ArcadeDB also supports Redis commands as a query language via the HTTP API. This allows you to execute Redis commands through the standard ArcadeDB HTTP `/command` endpoint by specifying `language=redis`.

This is useful when:

- You want to use Redis commands without a Redis client library

- You need to integrate Redis operations into existing HTTP-based workflows
- You want to combine Redis commands with other ArcadeDB query languages in the same application

Basic Usage

To execute Redis commands via HTTP, send a POST request to the `/command` endpoint with `language` set to `redis`:

```
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "PING"}'
```

Response:

```
{"result": [{"value": "PONG"}]}
```

Supported Commands

All transient and persistent commands documented above are available via the HTTP API.

Examples of transient commands:

```
# SET a value
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "SET mykey myvalue"}'

# GET a value
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "GET mykey"}'

# INCREMENT a counter
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "INCR counter"}'
```

Examples of persistent commands:

```
# Store a document
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
```

```
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "HSET Account
{\"id\":1,\"name\":\"John\",\"age\":30}"}'

# Retrieve by index
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "HGET Account[id] 1"}'
```

Batch Commands

You can execute multiple commands in a single request by separating them with newlines. Each command is executed sequentially and an array of results is returned.

```
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "SET key1 value1\nSET key2 value2\nGET
key1\nGET key2"}'
```

Response:

```
{"result": [{"value": ["OK", "OK", "value1", "value2"]}]}
```

Comments are supported within batch commands using `#` or `//` prefixes:

```
# This is a comment
SET key1 value1
// This is also a comment
SET key2 value2
```

Transactions with MULTI/EXEC

Redis transactions are supported using the standard `MULTI` and `EXEC` commands. Commands between `MULTI` and `EXEC` are queued and executed atomically.

```
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "MULTI\nSET tx1 value1\nSET tx2 value2\nINCR
counter\nEXEC"}'
```

Response (array of results from each command in the transaction):

```
{"result": [{"value": ["OK", "OK", 1]}]}
```

RAM commands inside a transaction. The RAM commands are not stored in the database, so a transaction treats them in two different ways, depending on whether the caller acts on their reply:

- `SET` follows the transaction. While a transaction is active (a `MULTI`/`EXEC` block, the transaction the HTTP command endpoint runs every request in, or one your application opened), the new value is visible to that transaction only, is published when the transaction commits, and is discarded if it rolls back. A block that fails, for example on a duplicated key, leaves no `SET` behind.
- `INCR`, `INCRBY`, `INCRBYFLOAT`, `DECR`, `DECRBY` and `GETDEL` are applied atomically at the moment they run, as in Redis, so no two clients can ever receive the same counter value or claim the same key. When ArcadeDB retries a block automatically after a conflict, the retry reuses the value the first attempt obtained instead of applying the command a second time. If the block ultimately fails, the increment is not given back, just as a database sequence is not: the next caller gets the next value, leaving a gap. On a key the same transaction already `SET`, these commands work on the transaction's own value and follow the transaction like the `SET`.

The `DISCARD` command can be used to abort a transaction:

```
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "MULTI\nSET discarded value\nDISCARD"}'
```

A batch can mix transaction blocks and standalone commands, and runs every line in order: a command outside a block runs on its own, the commands between `MULTI` and `EXEC` run as one atomic transaction, and the commands between `MULTI` and `DISCARD` are dropped. The reply holds one entry per command that ran, in batch order (the commands of a committed block included), so a batch with no `DISCARD` answers exactly one entry per command. A batch whose only content is a discarded block answers `"OK"`.

```
curl -X POST "http://localhost:2480/api/v1/command/MyDatabase" \
  -H "Content-Type: application/json" \
  -u root:password \
  -d '{"language": "redis", "command": "SET a 1\nMULTI\nINCR a\nSET b x\nEXEC\nGET
b"}'
```

```
{"result": [{"value": ["OK", 2, "OK", "x"]}]}
```

The whole batch is checked before any command runs. `EXEC` or `DISCARD` without `MULTI`, a nested `MULTI` and a `MULTI` never closed by `EXEC` or `DISCARD` are refused with an error, and none of the batch's commands is executed.

Programmatic Access

You can also use Redis commands directly from Java code via the query engine:

```
try (ResultSet rs = database.query("redis", "PING")) {
    Result result = rs.next();
    String value = result.getProperty("value"); // Returns "PONG"
}

// SET and GET
database.command("redis", "SET mykey myvalue");
try (ResultSet rs = database.query("redis", "GET mykey")) {
    String value = rs.next().getProperty("value"); // Returns "myvalue"
}
```

Chapter 20. Graph Algorithms

20.1. Graph Algorithms

ArcadeDB provides a comprehensive library of graph algorithms available as Cypher procedures and SQL functions. Cypher procedures follow the pattern:

```
CALL algo.name([arg1, arg2, ...]) YIELD field1, field2, ...
RETURN field1, field2, ...
```

SQL path functions return a list of vertex RIDs and can be used directly in `SELECT` statements.

Every `relTypes` argument (a name, a comma-separated list or a list of names; `null` or blank for all types) must name existing edge types. A name that is not an edge type of the schema, such as a typo or a weight property passed where the relationship types go, is rejected with an error instead of being answered as a graph without edges. The weighted procedures all take the relationship types before the weight property.

An optional trailing argument can be left out or passed as `null`: both take the default the parameter tables list. A value of the wrong type is rejected with an error naming the parameter, for example `algo.leiden(): maxIterations must be a number, got String`, rather than replaced by the default.

Since v26.10.2. Earlier versions treated an explicit `null` differently from a left-out argument: some procedures failed with a `NullPointerException`, and `algo.dijkstra.singleSource` walked both directions instead of its default `OUT`. Some numeric arguments of the wrong type were silently replaced by the default.

20.1.1. Algorithm Index

Algorithm	Procedure / Function	Category	Complexity
A*	`algo.astar`	Path Finding	CPU
Adamic-Adar Index	`algo.adamicAdar`	Link Prediction	CPU
All-Pairs Shortest Paths (APSP)	`algo.apsp`	Path Finding	CPU+RAM
All Simple Paths	`algo.allsimplepaths`	Path Finding	CPU
ArticleRank	`algo.articlerank`	Centrality	CPU
Articulation Points	`algo.articulationPoints`	Structural Analysis	CPU
Assortativity	`algo.assortativity`	Network Science	CPU
Bellman-Ford	`algo.bellmanford`	Path Finding	CPU
Bellman-Ford (SQL)	`bellmanFord()`	Path Finding	CPU
Betweenness Centrality	`algo.betweenness`	Centrality	CPU

Algorithm	Procedure / Function	Category	Complexity
BFS (Breadth-First Search)	`algo.bfs`	Path Finding	CPU
Biconnected Components	`algo.biconnectedComponents`	Structural Analysis	CPU
Bipartite Check	`algo.bipartite`	Structural Analysis	CPU
Bipartite Matching	`algo.bipartiteMatching`	Structural Analysis	CPU
Bridges	`algo.bridges`	Structural Analysis	CPU
Clique Enumeration	`algo.clique`	Structural Analysis	CPU
Closeness Centrality	`algo.closeness`	Centrality	CPU
Common Neighbors	`algo.commonNeighbors`	Link Prediction	CPU
Conductance	`algo.conductance`	Community Quality	CPU
Cycle Detection	`algo.cycleDetection`	Structural Analysis	CPU
Degree Centrality	`algo.degree`	Centrality	CPU
Densest Subgraph	`algo.densestSubgraph`	Structural Analysis	CPU
DFS (Depth-First Search)	`algo.dfs`	Path Finding	CPU
Dijkstra	`algo.dijkstra`	Path Finding	CPU
Dijkstra Single-Source	`algo.dijkstra.singleSource`	Path Finding	CPU
Dijkstra (SQL)	`dijkstra()`	Path Finding	CPU
Eccentricity	`algo.eccentricity`	Centrality	CPU
Eigenvector Centrality	`algo.eigenvector`	Centrality	CPU
Graph Coloring	`algo.graphColoring`	Structural Analysis	CPU
Graph Summary	`algo.graphSummary`	Graph Statistics	CPU
Harmonic Centrality	`algo.harmonic`	Centrality	CPU
Hierarchical Clustering	`algo.hierarchicalClustering`	Community Detection	CPU
HITS (Hubs and Authorities)	`algo.hits`	Centrality	CPU
Influence Maximization	`algo.influenceMaximization`	Network Science	CPU
FastRP (Fast Random Projection)	`algo.fastrp`	Node Embedding	CPU
GraphSAGE	`algo.graphsage`	Node Embedding	CPU
HashGNN	`algo.hashgnn`	Node Embedding	CPU
Jaccard Similarity	`algo.jaccard`	Similarity	CPU
K-Core Decomposition	`algo.kcore`	Structural Analysis	CPU
K-Nearest Neighbours (KNN)	`algo.knn`	Similarity	CPU
K-Shortest Paths	`algo.kShortestPaths`	Path Finding	CPU+RAM

Algorithm	Procedure / Function	Category	Complexity
K-Truss Decomposition	`algo.kTruss`	Structural Analysis	CPU
Katz Centrality	`algo.katz`	Centrality	CPU
Label Propagation	`algo.labelpropagation`	Community Detection	CPU
Leiden	`algo.leiden`	Community Detection	CPU
Local Clustering Coefficient	`algo.localClusteringCoefficient`	Structural Analysis	CPU
Longest Path (DAG)	`algo.longestPath`	Path Finding	CPU
Louvain	`algo.louvain`	Community Detection	CPU
Max k-Cut (Approximate)	`algo.maxKCut`	Community Detection	CPU
Maximum Flow	`algo.maxFlow`	Network Flow	CPU+RAM
Minimum Spanning Arborescence	`algo.msa`	Path Finding	CPU
Minimum Spanning Tree (MST)	`algo.mst`	Structural Analysis	CPU
Modularity Score	`algo.modularityScore`	Community Quality	CPU
Node2Vec	`algo.node2vec`	Node Embedding	CPU
PageRank	`algo.pagerank`	Centrality	CPU
Personalized PageRank	`algo.personalizedPageRank`	Centrality	CPU
Preferential Attachment	`algo.preferentialAttachment`	Link Prediction	CPU
Random Walk	`algo.randomWalk`	Traversal / Sampling	CPU
Resource Allocation	`algo.resourceAllocation`	Link Prediction	CPU
Rich-Club Coefficient	`algo.richClub`	Network Science	CPU
Same Community	`algo.sameCommunity`	Link Prediction	CPU
Shortest Path (SQL)	`shortestPath()`	Path Finding	CPU
SimRank	`algo.simRank`	Similarity	CPU+RAM
SLPA (Speaker-Listener Label Propagation)	`algo.slpa`	Community Detection	CPU
Strongly Connected Components (SCC)	`algo.scc`	Community Detection	CPU
Steiner Tree	`algo.steinerTree`	Path Finding	CPU+RAM
Topological Sort	`algo.topologicalSort`	Traversal / Ordering	CPU
Total Neighbors	`algo.totalNeighbors`	Link Prediction	CPU
Triangle Count	`algo.triangleCount`	Structural Analysis	CPU
VoteRank	`algo.voteRank`	Centrality	CPU

Algorithm	Procedure / Function	Category	Complexity
Weakly Connected Components (WCC)	`algo.wcc`	Community Detection	CPU

20.1.2. Categories

Detailed reference is split per category. Pick a category from the left navigation, or jump in:

- Shortest Path — which algorithm to use, Dijkstra vs A* vs Contraction Hierarchies, a worked example on the Florida road network, Tallahassee to Coral Gables
- Path Finding — Dijkstra, A*, Bellman-Ford, BFS, DFS, K-Shortest Paths, and more
- Centrality — PageRank, Betweenness, Closeness, Eigenvector, HITS, Katz, and more
- Community Detection — Louvain, Leiden, Label Propagation, SCC, WCC, and more
- Structural Analysis — Articulation Points, Bridges, Cliques, K-Core, K-Truss, and more
- Similarity & Link Prediction — Jaccard, Adamic-Adar, SimRank, Preferential Attachment
- Network Flow — Maximum Flow
- Traversal & Sampling — Random Walk, Topological Sort
- Network Science — Assortativity, Rich-Club, Influence Maximization
- Community Quality — Conductance, Modularity Score
- Graph Statistics — Graph Summary
- Node Embedding — Node2Vec, FastRP, GraphSAGE, HashGNN
- SQL Path Functions — `dijkstra()`, `bellmanFord()`, `shortestPath()` as SQL functions
- Notes — shared parameter conventions and memory considerations

20.2. Shortest Path

ArcadeDB answers "what is the shortest way from A to B?" in several ways, from SQL, Cypher and Java. This page helps you pick one, walks through all of them on the real road network of Florida (1.07 million junctions) from Tallahassee to Coral Gables, and explains when a **Customizable Contraction Hierarchy** (CCH) turns a weighted shortest path from milliseconds into microseconds.

Customizable Contraction Hierarchies (`CCH (…)` on a Graph Analytical View, `cchShortestPath()`, `algo.cch.shortestPath`) are available starting with ArcadeDB 26.10.2.

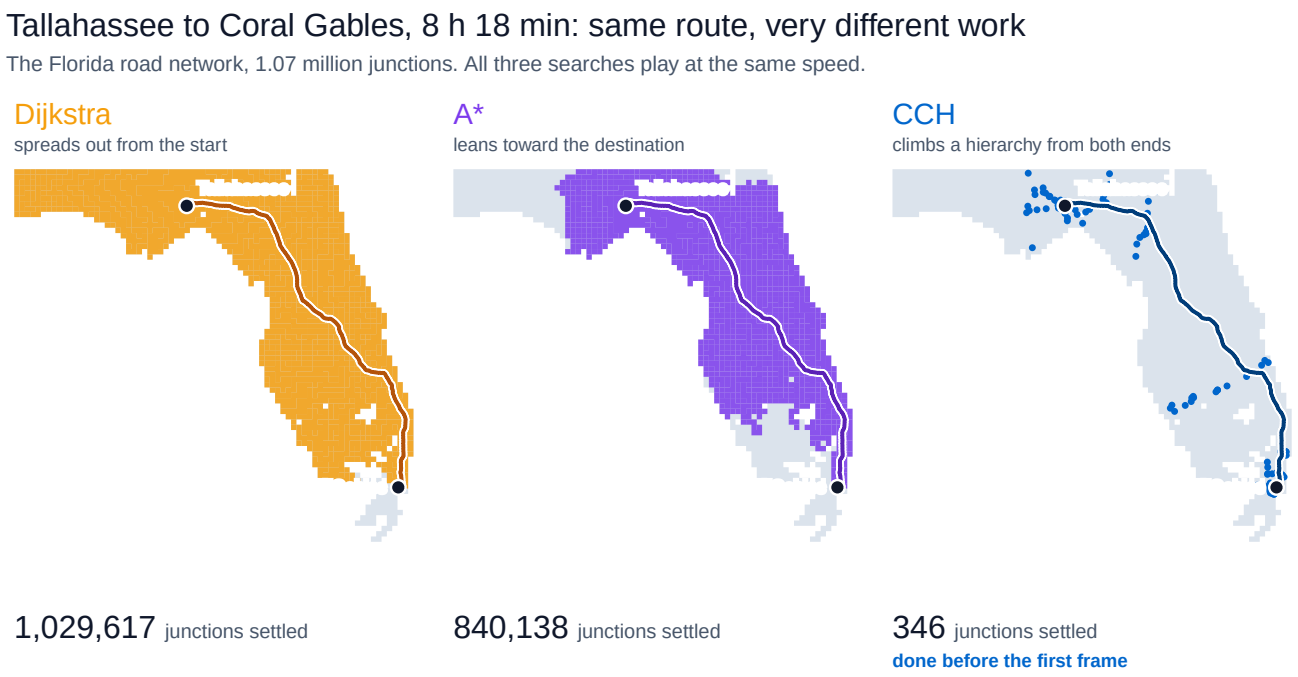
20.2.1. Choosing an Algorithm

You need	SQL	Cypher
The fewest hops, edges unweighted	shortestPath()	the `shortestPath()` pattern, algo.bfs

You need	SQL	Cypher
The lightest path, weights not negative, occasional queries	dijkstra()	algo.dijkstra
The same, guided by coordinates (latitude/longitude, x/y)	astar()	algo.astar
Many point-to-point queries on a road, logistics or utility network	cchShortestPath()	algo.cch.shortestPath
Negative weights	bellmanFord()	algo.bellmanford
Alternatives: the k best routes	-	algo.kShortestPaths
Distances from one source to every vertex	-	algo.dijkstra.singleSource

All weighted path finders read the weight the same way: the numeric value of the weight property, `1` when an edge has none, and an edge whose value is negative, NaN or infinite is not walked (only `bellmanFord()` walks negative weights). Among parallel edges the cheapest counts. Given the same graph they return the same shortest distance.

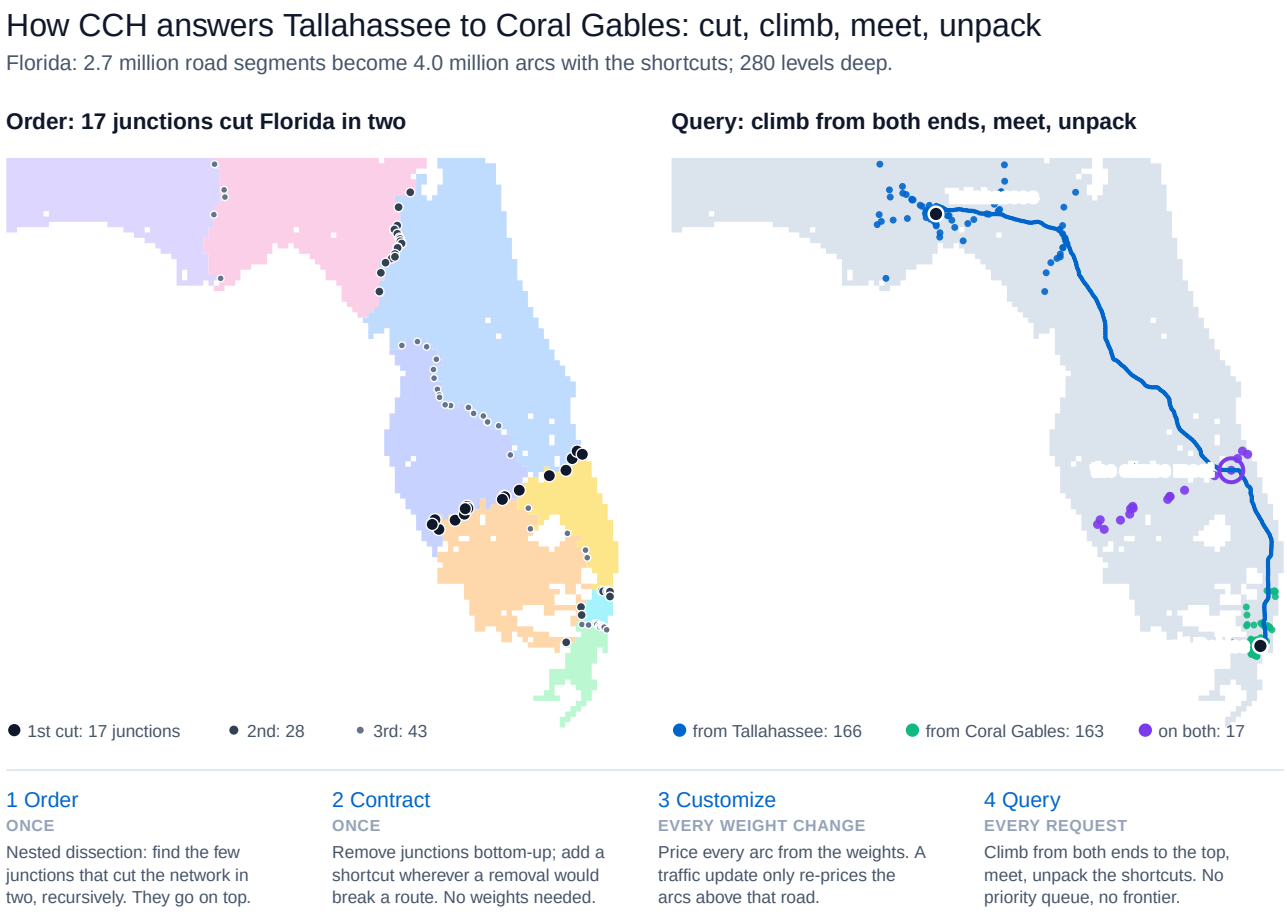
20.2.2. Dijkstra, A* and CCH



Dijkstra settles vertices in order of distance from the source, so its search is a growing circle: the farther the destination, the more of the graph it reads. From Tallahassee to Coral Gables it settles 1,029,617 of Florida's 1,070,376 junctions. **A*** adds an estimate of the remaining distance (for example the straight line between two coordinates) and leans the circle toward the destination: 840,138 junctions. Both need nothing prepared in advance, and both slow down as routes get

longer.

A **Customizable Contraction Hierarchy** moves the work ahead of the queries. Once, it ranks the vertices so that the small sets of vertices that split the network (on a road map: the few junctions where long routes must pass; in Florida, 17 junctions cut the state in two) sit at the top, and adds shortcut edges so that every route can go up the ranking and then down. A query then only climbs from both endpoints to the top and unpacks the shortcuts on the way: 346 junctions for the same route. Its cost depends on the height of the hierarchy, not on the distance between the two vertices: on the Florida road network that is about 0.01 to 0.04 ms for any route, against 8 to 104 ms for Dijkstra and 3 to 132 ms for A* (see Performance).



The **customizable** part matters for real networks: the ranking and the shortcuts depend only on which roads exist, not on what they cost. When a weight changes (traffic, a closure) only the costs of the shortcuts above that road are recomputed, typically in a millisecond. See how-to/data-modeling/graph-olap.pdf for the details of each phase.

20.2.3. Try It: the Florida Road Network

The examples use the Florida road network of the 9th DIMACS Implementation Challenge, derived from the US Census TIGER/Line data: 1,070,376 junctions with their coordinates and 2,712,798 directed road segments, each with a length in kilometers (`km`) and a travel time in minutes (`minutes`). A two-way road is two segments, one per direction.

```
CREATE VERTEX TYPE Junction;
```

```
CREATE PROPERTY Junction.id INTEGER;
CREATE PROPERTY Junction.lat DOUBLE;
CREATE PROPERTY Junction.lon DOUBLE;
CREATE PROPERTY Junction.name STRING;

CREATE EDGE TYPE ROAD;
CREATE PROPERTY ROAD.km DOUBLE;
CREATE PROPERTY ROAD.minutes DOUBLE;
```

The route-planning use case downloads the three DIMACS files (`USA-road-d.FLA.gr.gz` for the lengths, `USA-road-t.FLA.gr.gz` for the travel times, `USA-road-d.FLA.co.gz` for the coordinates), converts them and loads them with GraphBatch. `km` is the DIMACS length, in tenths of a meter, divided by 10,000. `minutes` reads the DIMACS travel time as millionths of an hour: the files do not state the unit, but with that reading the four road classes drive at 40, 60, 80 and 100 km/h. Clone the repository, enter `route-planning`, then pick one:

- **Docker**: `docker compose up -d` and `./setup.sh` start ArcadeDB, create the `RoutePlanning` database, load the network with one request to the batch endpoint (about 11 seconds), name a few junctions after their cities (`sql/02-places.sql`: Tallahassee, Coral Gables, Orlando, Tampa, Jacksonville, Miami, Gainesville, Pensacola) and create the view of Adding a Contraction Hierarchy. `./queries/queries.sh` then runs the queries of this page with curl, and the `java` folder runs them from a Java client.
- **Your own server**: `ARCADEDB_URL=http://localhost:2480 ARCADEDB_PASS=<root password> ./setup.sh`. The load is a 150 MB request: raise `arcadedb.server.httpBodyContentMaxSize` above its 100 MB default, as the use case's `docker-compose.yml` does.
- **Embedded**: see From Java.

The segments are directed, so every query below walks them with `OUT`. On a graph that stores each road once and walks it both ways, use `BOTH`.

Dijkstra, A* and the Fewest Hops in SQL

The path functions return the vertices of the path in order. `expand()` turns them into rows:

```
SELECT lat, lon FROM (
  SELECT expand(dijkstra($from, $to, 'minutes', { direction: 'OUT', edgeTypeNames:
['ROAD'] }))
  LET $from = (SELECT FROM Junction WHERE name = 'Tallahassee'),
      $to   = (SELECT FROM Junction WHERE name = 'Coral Gables')
)
```

1,501 junctions and 498 minutes (8 h 18 min): I-10 east, I-75 south, the Florida Turnpike and I-95 into Miami. On this graph the SQL function takes about 3 seconds: it walks the stored records, and a route across the state reads nearly all of them.

A* needs the vertex properties that hold the coordinates and an estimate that never exceeds the real remaining cost. With `lat`/`lon` in degrees, `dFactor` is the cost per degree of straight line. To keep

A* exact it must not exceed the cheapest cost per degree of any road segment in the data: 57 for `minutes` and 70 for `km` on this network:

```
SELECT astar($from, $to, 'minutes', { direction: 'OUT', edgeTypeNames: ['ROAD'],
                                      vertexAxisNames: ['lat', 'lon'],
heuristicFormula: 'EUCLIDEAN',
                                      dFactor: 57, tieBreaker: false }).size() AS
junctions
LET $from = (SELECT FROM Junction WHERE name = 'Tallahassee'),
    $to   = (SELECT FROM Junction WHERE name = 'Coral Gables')
```

The same 1,501 junctions. An estimate that overshoots makes A* faster but no longer guarantees the shortest path. A* is not faster than Dijkstra here: a few segments of the data are much faster than the straight line between their ends suggests, and the estimate has to stay below those too (see Performance).

`shortestPath()` ignores weights and minimizes the number of road segments: 1,064 here.

```
SELECT shortestPath($from, $to, 'OUT', 'ROAD').size() - 1 AS roads
LET $from = (SELECT FROM Junction WHERE name = 'Tallahassee'),
    $to   = (SELECT FROM Junction WHERE name = 'Coral Gables')
```

In Cypher, the fewest hops is a pattern:

```
MATCH p = shortestPath((a:Junction {name: 'Tallahassee'})-[:ROAD*]->(b:Junction {name:
'Coral Gables'}))
RETURN size(relationships(p)) AS roads
```

Adding a Contraction Hierarchy

A contraction hierarchy lives on a Graph Analytical View, one per weight property. One view can keep several, here one for distance and one for travel time:

```
CREATE GRAPH ANALYTICAL VIEW Florida
  EDGE TYPES (ROAD)
  UPDATE MODE SYNCHRONOUS
  CCH (km, minutes)
```

`./setup.sh` has already created it (`sql/03-view.sql`) and waited for it.

The view and its hierarchies are built in the background: for Florida, the view in about 2 seconds and both hierarchies in about 7. Queries never wait for them: until a hierarchy is ready they are answered by bidirectional Dijkstra, with the same result.

```
SELECT lat, lon FROM (
```

```
  SELECT expand(cchShortestPath($from, $to, 'minutes', { direction: 'OUT',
edgeTypeNames: ['ROAD'] }))
  LET $from = (SELECT FROM Junction WHERE name = 'Tallahassee'),
      $to   = (SELECT FROM Junction WHERE name = 'Coral Gables')
)
```

```
MATCH (a:Junction {name: 'Tallahassee'}), (b:Junction {name: 'Coral Gables'})
CALL algo.cch.shortestPath(a, b, 'ROAD', 'minutes', 'OUT') YIELD path, weight
RETURN weight AS minutes, size(nodes(path)) AS junctions
```

Same route, same 498 minutes, in about 2 ms through the HTTP API instead of 3 seconds. `path` is a Cypher path, read with `nodes(path)`, `relationships(path)` (the cheapest edge between each pair of consecutive nodes) and `length(path)`: see reference/graph-algorithms/path-finding.pdf.

Name the edge types the view covers (`edgeTypeNames` in SQL, the relationship types in Cypher). A query without them walks **every** edge type of the database. The hierarchy still answers it while the edge types the view does not cover have no edges (*since v26.10.2*; earlier versions gave up as soon as the schema had such a type, even an empty one). Once one of them gets an edge, that is a different graph from the one the hierarchy was built for, and the query is answered by Dijkstra. A request a hierarchy turns away because of its edge types is counted in the hierarchy's `edgeTypeMismatches` statistic and logged once.

Two Metrics, Two Routes

The shortest route is not always the fastest one. From Tallahassee to Coral Gables, each route with the other cost summed over its own road segments:

```
MATCH (a:Junction {name: 'Tallahassee'}), (b:Junction {name: 'Coral Gables'})
CALL algo.cch.shortestPath(a, b, 'ROAD', 'km', 'OUT') YIELD path, weight
RETURN weight AS km, reduce(m = 0.0, r IN relationships(path) | m + r.minutes) AS
minutes
```

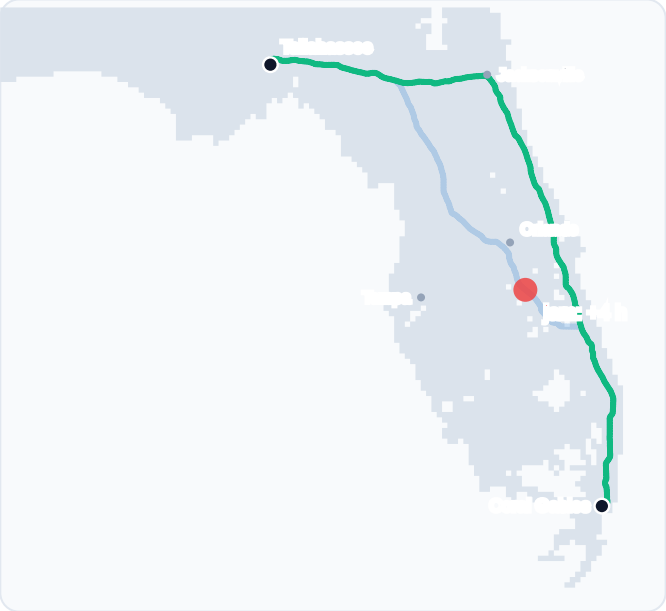
Minimize	km	minutes	route
`km`	742.2	679.1 (11 h 19 min)	highways through the interior, past Lake Okeechobee
`minutes`	788.6	498.0 (8 h 18 min)	I-10, I-75, the Florida Turnpike

Both hierarchies share the same view and are kept up to date together.

Live Traffic

Live traffic: one update, a new route a millisecond later

The C in CCH is for customizable: the hierarchy is kept, only the costs above a changed road are recomputed.



```
// fastest route, by travel time
CALL algo.cch.shortestPath(tallahassee,
     coralGables, 'ROAD', 'minutes', 'OUT')
8 h 18 min  via I-75 and the Turnpike

// 4 hours of queue south of Orlando
MATCH (:Junction {id: 702459})
      -[r:ROAD]->(:Junction {id: 702911})
SET r.minutes = 240
committed
hierarchy caught up in 0.9 ms
CALL algo.cch.shortestPath(...)
8 h 20 min  via I-10 and I-95
```

Only the arcs above the jammed road are re-priced.
Queries never wait: meanwhile, Dijkstra answers.

A weight is an ordinary edge property: change it with a normal `UPDATE` or Cypher `SET` and the hierarchy follows. Four hours of queue on the Turnpike south of Orlando, on the segment between junctions 702459 and 702911 (normally 6.3 minutes):

```
MATCH (:Junction {id: 702459})-[r:ROAD]->(:Junction {id: 702911})
SET r.minutes = 240
```

Right after the commit the hierarchy re-prices the arcs above that road (0.9 ms here), and the fastest route leaves I-75 at Lake City for I-10 east and I-95 down the Atlantic coast, rejoining the old route near Vero Beach: 500.1 minutes (8 h 20 min) and 825.7 km. A query that arrives in between is answered by bidirectional Dijkstra, so no answer is ever older than the data. Setting `minutes` back to 6.3009 restores the original route.

Find the road through the indexed `Junction.id` as above. The SQL form `UPDATE ROAD SET minutes = 240 WHERE @out.id = 702459 AND @in.id = 702911` gives the same result, but scans every `ROAD` edge to evaluate the condition.

A closed road is a road that cannot be walked. Since a negative weight is never walked, closing it is an update too (`SET r.minutes = -1`), and the hierarchy treats it the same way.

From Java

The same steps from an embedded database: no server, no build step. From the `route-planning` directory of the use case, with Java 21 or later and the `lib` directory of an ArcadeDB distribution:

```
java -Xmx4g -cp "$ARCADEDB_HOME/lib/*" embedded/FloridaRoutes.java
```

It downloads the DIMACS files into `data/` the first time and loads them with the GraphBatch Java

API, in about 7 seconds: the junctions first, in DIMACS order so that the DIMACS id `n` is the RID at index `n - 1`, then the road segments with both costs. The core of it:

```
try (Database db = new DatabaseFactory("./target/databases/florida-routes").create())
{
  // ... the schema of sql/01-schema.sql

  RID[] junctions;
  try (GraphBatch batch = db.batch().build()) {
    junctions = batch.createVertices("Junction", properties); // Object[][]: { "id",
1, "lat", 30.43, "lon", -84.28 }, ...
  }
  try (GraphBatch batch = db.batch().withBatchSize(500_000).withExpectedEdgeCount
(2_712_798).build()) {
    // for each DIMACS arc "a from to length" and its travel time:
    batch.newEdge(junctions[from - 1], "ROAD", junctions[to - 1], "km", length /
10_000.0, "minutes", time * 0.00006);
  }

  db.command("sql", "CREATE GRAPH ANALYTICAL VIEW Florida EDGE TYPES (ROAD) UPDATE
MODE SYNCHRONOUS CCH (km, minutes)");

  // optional: wait for the view and its hierarchies, so the first queries already use
them
  GraphAnalyticalView view = GraphAnalyticalViewRegistry.get(db, "Florida");
  view.awaitReady(5, TimeUnit.MINUTES);
  for (ContractionHierarchy cch : view.getContractionHierarchies())
    cch.awaitReady(false, 5, TimeUnit.MINUTES); // directed ('OUT') queries only

  try (ResultSet rs = db.query("opencypher", """
      MATCH (a:Junction {id: $from}), (b:Junction {id: $to})
      CALL algo.cch.shortestPath(a, b, 'ROAD', 'minutes', 'OUT') YIELD path, weight
      RETURN weight, size(nodes(path)) AS junctions""", Map.of("from", 472033, "to",
624638))) {
    Result row = rs.next();
    System.out.println(row.getProperty("junctions") + " junctions, " + row.
getProperty("weight") + " min");
  }
}
```

The view can also be defined with the builder, `GraphAnalyticalView.builder(db).withName("Florida").withEdgeTypes("ROAD").withUpdateMode(UpdateMode.SYNCHRONOUS).withContractionHierarchy("km").withContractionHierarchy("minutes").build()`.

The full example also compares `dijkstra()`, `astar()` and `cchShortestPath()` on the same route, shows the two metrics, and applies the traffic update. Embedded, Tallahassee to Coral Gables takes about 3.5 seconds with `dijkstra()`, 7 seconds with `astar()` and 0.4 ms with `cchShortestPath()`, SQL parsing included. The SQL path functions walk the stored records; the benchmark below compares the algorithms themselves on the same in-memory arrays.

20.2.4. Which Engine Answered?

`algo.cch.shortestPath` and `cchShortestPath()` always return the exact shortest path, and pick the fastest way to compute it: the hierarchy when one is ready for exactly the weight, the edge types and the current state of the data; otherwise bidirectional Dijkstra on the view's columns, or on the records. Two ways to see which one was used:

- Each hierarchy counts the queries it answered and the ones it handed back to Dijkstra:

  ```
  SELECT name, status, contractionHierarchies FROM schema:graphAnalyticalViews
  ```

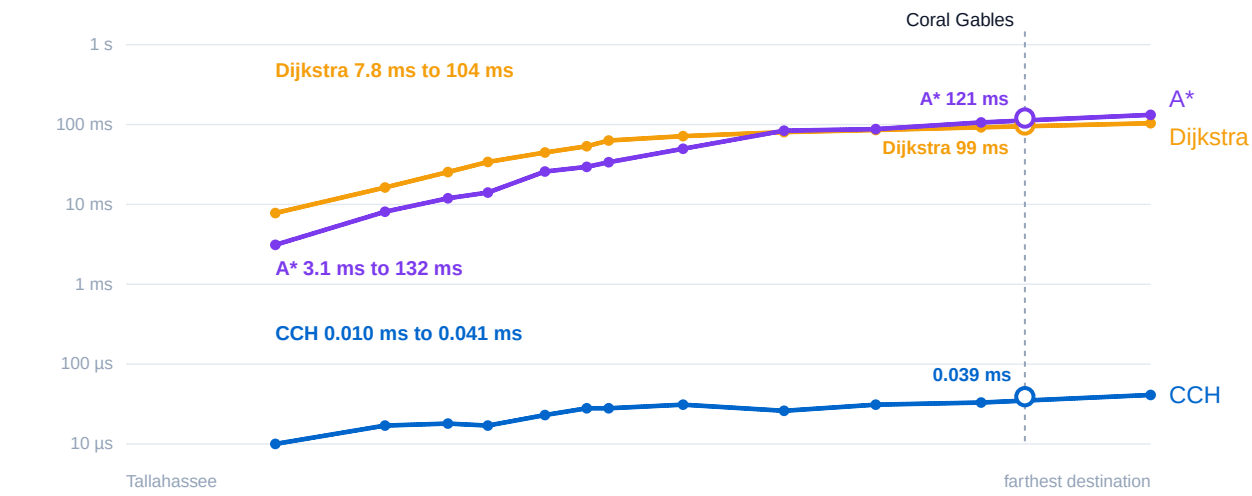
  Each entry reports `weightProperty`, `status` (`PREPARING`, `READY`, `UNSUITABLE`, `UNAVAILABLE`), `arcs`, `memoryUsageBytes`, `queries`, `fallbacks`, the number of full and partial customizations and `lastIncrementalUpdateMicros`, the time the last update took from commit to ready.

- `PROFILE` on a Cypher query prints `CSR-accelerated via Graph Analytical View` when the hierarchy answered.

A query is answered by Dijkstra, exactly and only more slowly, when:

- the hierarchy is still being built or catching up with a commit;
- it is the first undirected (`BOTH`) query on the hierarchy: directed queries are prepared from the start, the undirected metric is prepared on the first request (or by `awaitReady(true, …)` in Java);
- the calling transaction holds uncommitted changes, which no view can see;
- the weight property or the set of edge types differs from the hierarchy's (see the tip above);
- the graph has no small separators, as social graphs or graphs with supernodes: the hierarchy would be many times larger than the graph, so it is not built and reports `UNSUITABLE` (bound by `arcadedb.gavCchMaxArcsPerEdge`, default 16 arcs per edge).

20.2.5. Performance



CCHRoadNetworkBenchmark, Apple M5 Pro, median of 3 runs. From Tallahassee to the quantiles of its shortest-path tree, plus Coral Gables.
A* estimates with the straight line, scaled to 67% so that it never overshoots a road segment: that is what keeps it exact.
Memory per query: Dijkstra 11 to 44 MB, A* 9 to 52 MB, CCH 6 to 43 KB. Through the database, CCH: 0.04 to 0.06 ms.

Measured with `CCHRoadNetworkBenchmark` (engine test sources) on the DIMACS Florida road network (`USA-road-d.FLA`, 1,070,376 junctions, 2,712,798 road segments), Apple M5 Pro, median of three runs. The routes start in Tallahassee and go to 12 destinations at the quantiles of its shortest-path tree, plus Coral Gables:

	CCH	**Dijkstra**	**A***
Query, 12 destinations from near to farthest	0.010 - 0.041 ms	7.8 - 104 ms	3.1 - 132 ms
Tallahassee to Coral Gables	0.039 ms	99 ms	121 ms
Memory per query	6 - 43 KB	11 - 44 MB	9 - 52 MB
Query through the database (record ids in and out)	0.04 - 0.06 ms		
Update 1 / 10 / 100 road weights, commit + catch-up	0.8 / 8.3 / 41 ms		
Close a road	0.95 ms		
Add a road between two junctions the hierarchy does not join	0.46 s		
First build: view, then hierarchy ready	1.7 s, 6.6 s		
View + hierarchy memory	306 MB		

Dijkstra and A* are the benchmark's reference implementations on primitive arrays, the fairest baselines for the algorithms themselves. A* estimates the remaining cost with the straight line to the destination, scaled so that it never exceeds the cost of any road segment, which is what keeps it exact. In this data 112 of the 2.7 million segments are shorter than the straight line between their

ends, so the estimate is capped at 67% of the true distance: A* is 2.5 times faster than Dijkstra for nearby destinations and slower across the state.

To reproduce it, download `USA-road-d.FLA.gr.gz` and `USA-road-d.FLA.co.gz` (the coordinates, for A*) from the 9th DIMACS Implementation Challenge and run, from the ArcadeDB source tree (472033 and 624638 are the DIMACS ids of the junctions the use case names Tallahassee and Coral Gables):

```
mvn -pl engine test -Dtest=CCHRoadNetworkBenchmark -DexcludedGroups=none -
Dsurefire.failIfNoSpecifiedTests=false \
    -Dcch.dimacs=/path/to/USA-road-d.FLA.gr.gz -Dcch.coordinates=/path/to/USA-road-
d.FLA.co.gz \
    -Dcch.source=472033 -Dcch.target=624638
```

Without `-Dcch.dimacs` it runs on a synthetic 300 x 300 grid, the hardest case for this technique. Without `-Dcch.source` the routes start from the first junction of the file, and without `-Dcch.coordinates` A* is skipped.

References

- Dibbelt, Strasser, Wagner: *Customizable Contraction Hierarchies*, ACM Journal of Experimental Algorithmics 21 (2016)
- See how-to/data-modeling/graph-olap.pdf for how the hierarchy is built, kept up to date and persisted, and reference/graph-algorithms/path-finding.pdf / reference/graph-algorithms/sql-path-functions.pdf for the full reference of the procedure and the function.

20.3. Path Finding

The `path` a procedure yields

`algo.dijkstra`, `algo.cch.shortestPath`, `algo.astar`, `algo.bellmanford`, `algo.allsimplepaths`, `algo.kShortestPaths`, `algo.randomWalk` and the `path.expand`, `path.expandConfig` and `path.spanningTree` procedures yield their `path` as a Cypher path, the same value a `MATCH p = ⋯` pattern binds: the nodes from the start to the end node and, between each pair of consecutive nodes, the relationship that joins them. Read it with the path functions, return it as it is, or pass it to the `path.*` functions:

```
MATCH (a:City {name: 'A'}), (b:City {name: 'C'})
CALL algo.dijkstra(a, b, 'ROAD', 'km') YIELD path, weight
RETURN [n IN nodes(path) | n.name] AS route, length(path) AS hops, weight
```

A Bolt driver receives it as a `Path`. Over HTTP, gRPC and the PostgreSQL protocol, which have no path type, it is the object `{"_type": "path", "nodes": [...], "relationships": [...], "length": n}`, with every node and relationship written whole. This shape is not limited to these procedures: since v26.10.2 every Cypher path (a `MATCH p = ⋯` pattern, `shortestPath()`, `allShortestPaths()`, `apoc.path.create`, `apoc.path.slice` and `apoc.path.combine`) is the same path value and reaches the client the same way, and Studio's graph view draws it.

Since v26.10.2. Earlier versions yielded that object itself, as a map, so `nodes(path)`,

`relationships(path)` and `length(path)` answered `[]`, `[]` and `0` on it, and `algo.kShortestPaths` and `algo.randomWalk` left its relationships out. Queries written against the map keep working: `path.nodes`, `path.relationships` and `path.length` still read the path a procedure yields.

The relationship pattern of `algo.dijkstra`, `algo.astar` and `algo.allsimplepaths`

These three procedures read their `relTypes` argument as APOC's relationship pattern, the same syntax as the relationship filter of the `path.*` procedures: `TYPE` follows the type in both directions, `TYPE>` outgoing only, `<TYPE` incoming only, `>` and `<` every type in one direction, and `|` separates the alternatives, each with its own direction (`'ROAD>|<RAIL'`). A comma separates alternatives too, as the list of names did before. A blank pattern (`''`) follows every relationship in both directions, and a type the schema does not have is refused. An edge type whose name contains `|`, `<`, `>` or `,` cannot be named here.

Since v26.12.1. Earlier versions read the argument as a comma-separated list of type names only, so `'ROAD>'` or `'ROAD|RAIL'` was refused as a type that does not exist, and the fifth argument of `algo.dijkstra` was only the direction.

20.3.1. algo.dijkstra

Property	Value
Procedure	`algo.dijkstra`
Category	Path Finding
Complexity	CPU
Min Args	4
Max Args	5

Syntax

```
CALL algo.dijkstra(startNode, endNode, relTypes, weightProperty [, defaultWeight])
YIELD path, weight
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex
`endNode`	Vertex	Yes	—	Destination vertex
`relTypes`	String	Yes	-	Relationship types and directions in APOC's syntax (`'ROAD>'`, `'<RAIL'`, `'ROAD|RAIL'`, `'>'`), see the relationship pattern; pass `''` for all types

Parameter	Type	Required	Default	Description
`weightProperty`	String	Yes	—	Edge property name to use as weight
`defaultWeight`	Number or String	No	-	APOC's `defaultWeight`: the weight of an edge without a numeric `weightProperty` (without it, such an edge weighs `1`). A string is read as the traversal direction (`"IN"`, `"OUT"` or `"BOTH"`, default `"BOTH"`), applied to the types of `relTypes` without a direction marker

Yield Fields

Field	Type	Description
`path`	Path	The path from the start to the end node, read with `nodes(path)`, `relationships(path)` and `length(path)` (see [graph-algo-path-value])
`weight`	Double	Total path weight

Description

Finds the single shortest weighted path between two vertices using Dijkstra's algorithm. Internally delegates to the A* implementation with no heuristic. An edge without `weightProperty` (or whose value is not a number) weighs `defaultWeight`, or `1` when it is not passed, and an edge whose weight is negative, NaN or infinite is not walked: use algo.bellmanford for negative weights.

Use Cases

- Route planning in transportation networks
- Cost-optimal navigation in weighted graphs
- Network latency analysis

Example

```
MATCH (src:City {name:'Rome'}), (dst:City {name:'Berlin'})
CALL algo.dijkstra(src, dst, 'ROAD>', 'distance')
YIELD path, weight
RETURN path, weight
```

References

- Dijkstra's algorithm – Wikipedia

See also: reference/graph-algorithms/sql-path-functions.pdf for the SQL function variant, and astar() / dijkstra() for the SQL function equivalents.

20.3.2. algo.cch.shortestPath

Property	Value
Procedure	`algo.cch.shortestPath`
Category	Path Finding
Complexity	CPU
Min Args	4
Max Args	5

Syntax

```
CALL algo.cch.shortestPath(startNode, endNode, relTypes, weightProperty [, direction])
YIELD path, weight
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	-	Source vertex
`endNode`	Vertex	Yes	-	Destination vertex
`relTypes`	String or List	Yes	-	Relationship type(s): a name, a comma-separated list or a list; `null` for every type
`weightProperty`	String	Yes	-	Edge property name to use as weight
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`

Yield Fields

Field	Type	Description
`path`	Path	The path, with the cheapest usable relationship between each pair of consecutive nodes (see [graph-algo-path-value])
`weight`	Double	Total path weight

Description

Finds the shortest weighted path between two vertices. When a Graph Analytical View keeps a Customizable Contraction Hierarchy for the same weight property and relationship types, the path is answered by the hierarchy in a time that depends on the depth of the hierarchy rather than on how far apart the two vertices are: on road-like graphs it is one to two orders of magnitude faster

than Dijkstra, and the gap grows with the length of the route.

When no hierarchy can answer - none is defined, it is still being prepared after a change, the graph was found unsuitable for it, or the calling transaction holds uncommitted changes the view cannot see - the procedure answers with a bidirectional Dijkstra, on the view's columns when a view materializes the weight and on the records otherwise. The answer is always exact and always reflects the calling transaction's own changes; only the speed varies.

The weight of an edge is the numeric value of `weightProperty`; an edge without one weighs `1`, and an edge whose value is negative, NaN or infinite is not walked. Among parallel edges only the cheapest counts. Every weighted path finder (`algo.dijkstra`, `algo.astar`, `algo.dijkstra.singleSource`, `algo.kShortestPaths`, `algo.steinerTree` and the `dijkstra()`, `astar()`, `duanSSSP()` and `cchShortestPath()` SQL functions) reads the weight by this same rule, so they answer the same path.

Use Cases

- Route planning on road, logistics and utility networks with many point-to-point queries
- Navigation where weights change often (traffic, closures) but the network itself rarely does

Example

```
CREATE GRAPH ANALYTICAL VIEW roads EDGE TYPES (ROAD) UPDATE MODE SYNCHRONOUS CCH
(distance)
```

```
MATCH (src:City {name:'Rome'}), (dst:City {name:'Berlin'})
CALL algo.cch.shortestPath(src, dst, 'ROAD', 'distance', 'OUT')
YIELD path, weight
RETURN path, weight
```

References

- Dibbelt, Strasser, Wagner: *Customizable Contraction Hierarchies*, ACM Journal of Experimental Algorithmics 21 (2016)

See also: cchShortestPath() for the SQL function, and how-to/data-modeling/graph-olap.pdf for defining and monitoring the hierarchy.

20.3.3. algo.astar

Property	Value
Procedure	`algo.astar`
Category	Path Finding
Complexity	CPU
Min Args	4

Property	Value
Max Args	6

Syntax

```
CALL algo.astar(startNode, endNode, relTypes, weightProperty [, latProperty,
lonProperty])
YIELD path, weight
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex
`endNode`	Vertex	Yes	—	Destination vertex
`relTypes`	String	Yes	-	Relationship types and directions in APOC's syntax, see the relationship pattern; pass `''` for all types
`weightProperty`	String	Yes	—	Edge property name to use as weight
`latProperty`	String	No	—	Vertex property name for latitude (enables geographic heuristic)
`lonProperty`	String	No	—	Vertex property name for longitude (enables geographic heuristic)

Yield Fields

Field	Type	Description
`path`	Path	The path from the start to the end node, read with `nodes(path)`, `relationships(path)` and `length(path)` (see [graph-algo-path-value])
`weight`	Double	Total path weight

Description

Finds the shortest weighted path using the A* algorithm. When `latProperty` and `lonProperty` are provided, a geographic (Euclidean) heuristic accelerates the search. Without the geographic properties it behaves identically to Dijkstra. Edges are followed in both directions, like `algo.dijkstra's default. The returned `path` includes the relationships between the vertices. An edge without `weightProperty` (or whose value is not a number) weighs `1`, and an edge whose weight is negative, NaN or infinite is not walked: use algo.bellmanford for negative weights.

Use Cases

- Geospatial route optimization

- Game AI pathfinding with spatial heuristics
- Weighted graph shortest paths with domain-specific heuristics

Example

```
MATCH (src:City {name:'Milan'}), (dst:City {name:'Paris'})
CALL algo.astar(src, dst, 'ROAD', 'km', 'lat', 'lon')
YIELD path, weight
RETURN path, weight
```

References

- A* search algorithm – Wikipedia

20.3.4. algo.bellmanford

Property	Value
Procedure	`algo.bellmanford`
Category	Path Finding
Complexity	CPU
Min Args	4
Max Args	4

Syntax

```
CALL algo.bellmanford(startNode, endNode, relTypes, weightProperty)
YIELD path, weight, negativeCycle
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex
`endNode`	Vertex	Yes	—	Destination vertex
`relTypes`	String	Yes	—	Comma-separated relationship types; pass `''` for all types
`weightProperty`	String	Yes	—	Edge property name to use as weight (may be negative)

Yield Fields

Field	Type	Description
`path`	Path	The path from the start to the end node, read with `nodes(path)`, `relationships(path)` and `length(path)` (see [graph-algo-path-value])
`weight`	Double	Total path weight
`negativeCycle`	Boolean	`true` if a negative-weight cycle is reachable from `startNode`

Description

Finds the shortest weighted path between two vertices using the Bellman-Ford algorithm. Unlike Dijkstra, it correctly handles graphs with negative edge weights. It runs V−1 relaxation iterations followed by one extra iteration to detect negative-weight cycles. If a negative cycle is detected, `negativeCycle` is set to `true` and the path result is unreliable.

Use Cases

- Financial transaction networks with negative-cost edges
- Currency arbitrage detection (negative cycle = profit cycle)
- Graphs where edge weights can be negative

Example

```
MATCH (a:Account {id:1}), (b:Account {id:5})
CALL algo.bellmanford(a, b, 'TRANSFER', 'cost')
YIELD path, weight, negativeCycle
RETURN path, weight, negativeCycle
```

References

- Bellman-Ford algorithm – Wikipedia

See also: reference/graph-algorithms/sql-path-functions.pdf for the SQL function variant.

20.3.5. algo.allsimplepaths

Property	Value
Procedure	`algo.allsimplepaths`
Category	Path Finding
Complexity	CPU
Min Args	4
Max Args	4

Syntax

```
CALL algo.allsimplepaths(startNode, endNode, relTypes, maxDepth)
YIELD path
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex
`endNode`	Vertex	Yes	—	Destination vertex
`relTypes`	String	Yes	-	Relationship types and directions in APOC's syntax, see the relationship pattern; pass `''` for all types
`maxDepth`	Integer	Yes	—	Maximum path length in hops (minimum 1)

Yield Fields

Field	Type	Description
`path`	Path	One simple path from the start to the end node (see [graph-algo-path-value])

Description

Enumerates all simple (loop-free) paths between two vertices up to a maximum depth, using DFS with backtracking. Edges are traversed in both directions. Each result row contains one complete path. Use `maxDepth` to prevent combinatorial explosion in dense graphs.

Use Cases

- Finding all routes between two nodes in a network
- Dependency analysis between components
- Attack path enumeration in security graphs

Example

```
MATCH (s:Service {name:'AuthService'}), (t:Service {name:'Database'})
CALL algo.allsimplepaths(s, t, 'CALLS', 4)
YIELD path
RETURN path
```

References

- Path (graph theory) – Wikipedia

20.3.6. algo.kShortestPaths

Property	Value
Procedure	`algo.kShortestPaths`
Category	Path Finding
Complexity	CPU+RAM
Min Args	3
Max Args	6

Syntax

```
CALL algo.kShortestPaths(startNode, endNode, k [, relTypes, weightProperty,
direction])
YIELD path, weight, rank
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex
`endNode`	Vertex	Yes	—	Destination vertex
`k`	Integer	Yes	—	Number of shortest paths to find
`relTypes`	String	No	all types	Comma-separated relationship types
`weightProperty`	String	No	uniform weight	Edge property name for weights
`direction`	String	No	`BOTH`	`OUT`, `IN` or `BOTH`, as for `algo.dijkstra`

Yield Fields

Field	Type	Description
`path`	Path	The path, with the cheapest relationship between each pair of consecutive nodes (see [graph-algo-path-value])
`weight`	Double	Total path weight
`rank`	Integer	1-based rank (1 = shortest)

Description

Finds the k shortest loopless paths between two vertices using Yen's algorithm: Dijkstra on the adjacency lists, progressively building candidate paths by spur-node deviation from previous

shortest paths. Memory is the graph plus a few values per node. An edge without `weightProperty` weighs `1`, and an edge whose weight is negative, NaN or infinite is not walked. With the same arguments, rank 1 is the path `algo.dijkstra` returns.

Since v26.10.2 the default direction is `BOTH`, as for the other weighted path procedures. Earlier versions followed outgoing edges only, and gave a different (or no) path on graphs whose edges are stored in one direction but walked in both, such as roads. Pass `'OUT'` as the 6th argument to keep the old behaviour. The yielded `path` now also carries its relationships.

Use Cases

- Resilience planning: top-k alternative routes in case of failures
- Network redundancy analysis
- Multi-criteria path exploration

Example

```
MATCH (s:Router {id:'R1'}), (t:Router {id:'R10'})
CALL algo.kShortestPaths(s, t, 3, 'LINK', 'latency')
YIELD path, weight, rank
RETURN rank, path, weight ORDER BY rank ASC
```

References

- Yen's k shortest simple paths – Wikipedia

20.3.7. algo.apsp

Property	Value
Procedure	`algo.apsp`
Category	Path Finding
Complexity	CPU+RAM
Min Args	0
Max Args	2

Syntax

```
CALL algo.apsp([relTypes, weightProperty])
YIELD source, target, distance
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types (pass `null` for all types when only the weight is needed)
`weightProperty`	String	No	uniform weight	Edge property name for weights

Since 26.10.2 the relationship types come first, as in every other weighted `algo.*` procedure. Before that, `algo.apsp` took `(weightProperty, relTypes)`. A call still written in the old order, such as `algo.apsp('weight')`, is now rejected with "relationship type 'weight' does not exist": rewrite it as `algo.apsp(null, 'weight')`.

Yield Fields

Field	Type	Description
`source`	RID	Source vertex identity
`target`	RID	Target vertex identity
`distance`	Double	Shortest distance between source and target

Description

Computes shortest distances between all reachable pairs of distinct vertices using the Floyd-Warshall algorithm. Time complexity is O(V³); memory usage is O(V²). Only reachable pairs (finite distance, source ≠ target) are returned. Suitable for small to medium graphs.

Use Cases

- Graph diameter and radius computation
- Network reachability analysis
- Pre-computing all pairwise distances for downstream analytics

Example

```
CALL algo.apsp('ROAD', 'weight')
YIELD source, target, distance
RETURN source, target, distance ORDER BY distance DESC LIMIT 10
```

References

- Floyd-Warshall algorithm – Wikipedia

20.3.8. algo.bfs

Property	Value
Procedure	`algo.bfs`
Category	Path Finding
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.bfs(startNode [, relTypes, direction, maxDepth])
YIELD node, depth
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex from which the traversal starts
`relTypes`	String	No	all types	Comma-separated relationship types to follow
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`maxDepth`	Integer	No	unbounded	Maximum BFS depth (hop count) from the start node

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity (the start node is not included)
`depth`	Integer	BFS depth from the start node

Description

Performs a Breadth-First Search traversal from a given start node, visiting all reachable vertices in non-decreasing distance order. Each reachable vertex (excluding the start node itself) is yielded together with its BFS depth. An optional `maxDepth` bound limits the traversal radius.

Use Cases

- Shortest hop-count paths from a source
- Neighbourhood exploration at a given radius
- Level-order processing of tree or DAG structures

Example

```
MATCH (start:Person {name:'Alice'})
CALL algo.bfs(start, 'KNOWS', 'BOTH', 3)
YIELD node, depth
RETURN node.name, depth ORDER BY depth
```

References

- Breadth-first search – Wikipedia

20.3.9. algo.dfs

Property	Value
Procedure	`algo.dfs`
Category	Path Finding
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.dfs(startNode [, relTypes, direction, maxDepth])
YIELD node, depth
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex from which the traversal starts
`relTypes`	String	No	all types	Comma-separated relationship types to follow
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`maxDepth`	Integer	No	unbounded	Maximum DFS depth from the start node

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity (the start node is not included)

Field	Type	Description
`depth`	Integer	DFS discovery depth from the start node

Description

Performs an iterative (non-recursive) Depth-First Search from a given start node using an explicit stack, avoiding Java stack-overflow on large graphs. Vertices are yielded in DFS discovery order together with their depth. An optional `maxDepth` bound limits the traversal.

A bounded call costs the nodes it reaches, not the graph (issue #10580, as for `algo.bfs`): over a Graph Analytical View the walk reads the neighbours of the nodes it reaches from the view, and without a view it loads only the vertices it reaches from the start node, charging them to `arcadedb.cypher.algoMaxWorkingMemory`. Before v26.11.1 a call with `maxDepth = 1` loaded or copied the whole graph before it walked a single node. The results and their order are those of the earlier implementation.

Use Cases

- Reachability analysis
- Dependency resolution and cycle detection
- Tree/DAG structure enumeration

Example

```
MATCH (start:Module {name:'core'})
CALL algo.dfs(start, 'DEPENDS_ON', 'OUT')
YIELD node, depth
RETURN node.name, depth
```

References

- Depth-first search – Wikipedia

20.3.10. algo.dijkstra.singleSource

Property	Value
Procedure	`algo.dijkstra.singleSource`
Category	Path Finding
Complexity	CPU
Min Args	3
Max Args	4

Syntax

```
CALL algo.dijkstra.singleSource(startNode, relTypes, weightProperty [, direction])
YIELD node, cost
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Source vertex
`relTypes`	String	Yes	—	Comma-separated relationship types; pass `''` for all types
`weightProperty`	String	Yes	—	Edge property name to use as weight (must be numeric, non-negative)
`direction`	String	No	`"OUT"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`

Yield Fields

Field	Type	Description
`node`	RID	Reachable vertex identity (the start node is not included)
`cost`	Double	Minimum total weight from the start node to this vertex

Description

Computes the Single-Source Shortest Path (SSSP) from a given start node to every reachable vertex using Dijkstra's algorithm with a binary min-heap. This procedure extends `algo.dijkstra` to return results for all reachable targets at once instead of just a single source-target pair. An edge without `weightProperty` (or whose value is not a number) weighs `1`; an edge whose weight is negative, NaN or infinite is not walked.

Use Cases

- Distance maps from a hub in transportation networks
- Risk propagation from an origin node
- Identifying the closest facility to every location in a network

Example

```
MATCH (hub:City {name:'London'})
CALL algo.dijkstra.singleSource(hub, 'ROAD', 'distance', 'OUT')
YIELD node, cost
RETURN node.name, cost ORDER BY cost LIMIT 10
```

References

- Dijkstra's algorithm – Wikipedia

See also: algo.dijkstra for the single source-target variant.

20.3.11. algo.longestPath

Property	Value
Procedure	`algo.longestPath`
Category	Path Finding
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.longestPath([relTypes, weightProperty])
YIELD node, distance, source
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types to follow
`weightProperty`	String	No	`1.0` per hop	Edge property name to use as weight

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`distance`	Double	Maximum distance from the furthest source reaching this vertex
`source`	RID	The source vertex that achieves the maximum distance

Description

Finds the longest path in a Directed Acyclic Graph (DAG) using dynamic programming on topological order (Kahn's algorithm). If the graph contains a cycle the procedure detects it and returns an empty result set. When no `weightProperty` is provided, each hop has weight `1.0` (i.e., the

result is the maximum hop count).

Use Cases

- Critical path analysis in project scheduling (CPM)
- Longest dependency chain in build systems
- Pipeline bottleneck identification

Example

```
CALL algo.longestPath('DEPENDS_ON')
YIELD node, distance, source
RETURN node.name AS task, distance, source.name AS criticalSource
ORDER BY distance DESC LIMIT 5
```

References

- Longest path problem – Wikipedia

See also: reference/graph-algorithms/structural-analysis.pdf for topological ordering.

20.3.12. algo.msa

Property	Value
Procedure	`algo.msa`
Category	Path Finding
Complexity	CPU
Min Args	1
Max Args	3

Syntax

```
CALL algo.msa(rootNode [, relTypes, weightProperty]) YIELD source, target, weight,
totalWeight
```

Parameters

Name	Type	Default	Description
`rootNode`	Vertex	—	Root vertex for the arborescence
`relTypes`	String	all types	Comma-separated edge type names to traverse

Name	Type	Default	Description
`weightProperty`	String	`null`	Edge property for weights; `null` = all weights 1.0

Yield Fields

Field	Type	Description
`source`	Vertex	Source vertex of this MSA edge
`target`	Vertex	Destination vertex of this MSA edge
`weight`	Float	Weight of this edge
`totalWeight`	Float	Total weight of the arborescence (same on every row)

Description

Computes the Minimum Spanning Arborescence (directed MST) rooted at a specified vertex using the Chu-Liu/Edmonds algorithm. Unlike the undirected `algo.mst`, the arborescence is a **directed** spanning tree where every non-root vertex has exactly one incoming edge and all vertices are reachable from the root along directed paths.

Returns n − 1 rows (one per arborescence edge). Returns an empty result if no spanning arborescence exists (the directed graph is not strongly connected from the root).

Use Cases

- Optimising directed distribution networks (water, power, logistics)
- Dependency resolution with minimum total cost
- Directed network backbone extraction

Example

```
MATCH (r:City {name:'Rome'})
CALL algo.msa(r, 'ROAD', 'distance')
YIELD source, target, weight, totalWeight
RETURN source.name, target.name, weight ORDER BY weight
```

References

- Edmonds' algorithm – Wikipedia
- Chu, Y. J. & Liu, T. H. (1965). On the Shortest Arborescence of a Directed Graph. *Science Sinica*.

20.3.13. algo.steinerTree

Property	Value
Procedure	`algo.steinerTree`
Category	Path Finding
Complexity	CPU+RAM
Min Args	1
Max Args	3

Syntax

```
CALL algo.steinerTree(terminalNodes [, relTypes, weightProperty]) YIELD source,
target, weight, totalWeight
```

Parameters

Name	Type	Default	Description
`terminalNodes`	List<Vertex>	—	List of terminal (required) vertices to connect
`relTypes`	String	all types	Comma-separated edge type names to traverse (undirected)
`weightProperty`	String	`null`	Edge property for weights; `null` = all weights 1.0

Yield Fields

Field	Type	Description
`source`	Vertex	Source vertex of this Steiner-tree edge
`target`	Vertex	Destination vertex of this Steiner-tree edge
`weight`	Float	Weight of this edge
`totalWeight`	Float	Total Steiner-tree weight (same on every row)

Description

Finds an approximate minimum-weight connected subgraph that spans all terminal vertices, using the Kou-Markowsky-Berman 2-approximation algorithm (approximation ratio `2(1 − 1/|T|)` where `|T|` is the number of terminals):

1. Run Dijkstra from each terminal to find shortest paths to all other terminals.
2. Build a complete graph on terminals with shortest-path distances.
3. Find the MST of this terminal graph (Kruskal's).
4. Expand each MST edge back to its actual shortest path in the original graph.
5. Prune non-terminal leaves iteratively.

Edges are traversed undirected. Returns an empty result if terminals are unreachable from each other. An edge without `weightProperty` weighs `1`, an edge whose weight is negative, NaN or infinite is not walked, and among parallel edges the cheapest one is the one reported.

Use Cases

- Designing minimum-cost subnetworks connecting mandatory waypoints
- Content distribution network planning
- Pipeline / cable routing connecting required endpoints

Example

```
MATCH (a:City {name:'Rome'}), (b:City {name:'Milan'}), (c:City {name:'Naples'})
CALL algo.steinerTree([a, b, c], 'ROAD', 'distance')
YIELD source, target, weight, totalWeight
RETURN source.name, target.name, weight
```

References

- Steiner tree problem – Wikipedia
- Kou, L., Markowsky, G. & Berman, L. (1981). A fast algorithm for Steiner trees. *Acta Informatica*.

20.4. Centrality

20.4.1. algo.pagerank

Property	Value
Procedure	`algo.pagerank`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.pagerank([{dampingFactor: 0.85, maxIterations: 20, tolerance: 0.0001,
weightProperty: null, direction: 'OUT'}])
YIELD node, score
```

Parameters

Parameter	Type	Required	Default	Description
config map	Map	No	see defaults	Configuration properties (all optional)
`dampingFactor`	Double	No	`0.85`	Probability of following an edge vs. teleporting
`maxIterations`	Integer	No	`20`	Maximum number of power-iteration steps
`tolerance`	Double	No	`0.0001`	Convergence threshold (max score change)
`weightProperty`	String	No	`null`	Edge property for weighted PageRank
`direction`	String	No	`OUT`	Direction rank is pushed along: `OUT`, `IN` or `BOTH`

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`score`	Double	PageRank score

Description

Computes the PageRank centrality score for every vertex using the power-iteration method. Dangling nodes (no edges in the direction rank is pushed along) contribute their rank to all nodes proportionally via the teleportation mechanism. When `weightProperty` is specified, edge weights are used as transition probabilities (normalized per node).

`direction` selects which way rank flows, and unlike most algorithms it defaults to `OUT`:

- `OUT`: rank is pushed along the stored edges, so a vertex is important when important vertices point at it. This is classic PageRank.
- `IN`: rank is pushed along the reverse of every edge, which is PageRank over the reversed graph without building a reversed copy. Use it to rank the sources rather than the sinks, for example the papers that cite the most influential work in a citation graph. Under `IN` a dangling node is one with no incoming edges.
- `BOTH`: rank is pushed both ways, treating the graph as undirected. A dangling node is one with no edges at all.

Unweighted PageRank uses a Graph Analytical View when one covers the graph. Weighted PageRank (`weightProperty` set) always runs on the slower OLTP path, even when such a view exists. `direction` and `tolerance` work the same on both paths, so the scores do not depend on whether a view exists.

Before v26.10.2 a run served by a Graph Analytical View ignored `tolerance` and always ran `maxIterations` iterations, so creating a view could change the scores,

and with a large `tolerance` the ranking, of an unchanged query on unchanged data.

Use Cases

- Web page and document importance ranking
- Identifying authoritative nodes in citation networks
- Social network influence scoring

Example

```
CALL algo.pagerank({dampingFactor: 0.85, maxIterations: 30})
YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 10
```

References

- PageRank – Wikipedia

20.4.2. algo.betweenness

Property	Value
Procedure	`algo.betweenness`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.betweenness([{normalized: true}])
YIELD node, score
```

Parameters

Parameter	Type	Required	Default	Description
config map	Map	No	see defaults	Configuration properties (all optional)
`normalized`	Boolean	No	`true`	Divide scores by `1/n-1)(n-2` to normalize to [0,1]

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`score`	Double	Betweenness centrality score

Description

Measures how often a vertex lies on the shortest path between all other pairs, using the Brandes algorithm with stack-based back-propagation. Traversal follows outgoing edges only (directed). When `normalized` is `true`, scores are divided by `1/n-1)(n-2` (the directed normalization factor, matching NetworkX and Neo4j GDS).

Use Cases

- Identifying bridge nodes and bottlenecks in networks
- Communication hub detection in social graphs
- Critical infrastructure analysis

Example

```
CALL algo.betweenness({normalized: true})
YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 20
```

References

- Betweenness centrality – Wikipedia
- Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2), 163–177.

20.4.3. algo.closeness

Property	Value
Procedure	`algo.closeness`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	3

Syntax

```
CALL algo.closeness([relTypes, direction, normalized])
YIELD node, score
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`normalized`	Boolean	No	`true`	Use Wasserman-Faust formula for disconnected graphs

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`score`	Double	Closeness centrality score

Description

Computes closeness centrality via BFS from each vertex. For disconnected graphs, the Wasserman-Faust formula is applied: `score = (reachable / sumDist) × (reachable / (n-1))`, where `reachable` is the number of vertices reached and `sumDist` is the sum of shortest path distances to those vertices.

Use Cases

- Identifying well-positioned nodes for information spread
- Supply chain hub identification
- Social network influence reach analysis

Example

```
CALL algo.closeness('KNOWS', 'BOTH', true)
YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 10
```

References

- Closeness centrality – Wikipedia
- Wasserman, S. & Faust, K. (1994). *Social Network Analysis*.

20.4.4. algo.degree

Property	Value
Procedure	`algo.degree`

Property	Value
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.degree([relTypes, direction])
YIELD node, inDegree, outDegree, degree, score
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`inDegree`	Long	Number of incoming edges
`outDegree`	Long	Number of outgoing edges
`degree`	Long	Total degree (in + out for BOTH, else directional)
`score`	Double	Normalized degree: `total / (n-1)`

Description

Efficiently counts in-degree, out-degree, and total degree for every vertex using `vertex.countEdges()` to avoid materializing edge objects. The normalized score divides total degree by `(n-1)`, representing the fraction of all possible connections.

A type's degree includes the edges of its subtypes, and each edge is counted once whichever of the listed types it matches: with `E1 EXTENDS E0`, `algo.degree()`, `algo.degree('E0')` and `algo.degree(['E0', 'E1'])` all count an `E1` edge once, with or without a Graph Analytical View. *Since v26.12.1*; earlier versions counted the subtype's edges twice when a Graph Analytical View answered a call over several types (or over all types).

Use Cases

- Identifying high-degree hubs in social networks

- Popularity ranking by connection count
- Network connectivity baseline measurement

Example

```
CALL algo.degree('FOLLOWS', 'IN')
YIELD node, inDegree, score
RETURN node, inDegree, score ORDER BY inDegree DESC LIMIT 10
```

References

- Degree (graph theory) – Wikipedia

20.4.5. algo.harmonic

Property	Value
Procedure	`algo.harmonic`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	3

Syntax

```
CALL algo.harmonic([relTypes, direction, normalized])
YIELD node, score
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`normalized`	Boolean	No	`true`	Divide raw score by `(n-1)`

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`score`	Double	Harmonic centrality score

Description

Computes harmonic centrality as the sum of reciprocal shortest-path distances from a vertex to all reachable vertices: `score = Σ 1/dist(v, u)`. When `normalized` is `true`, divides by `(n-1)`. Harmonic centrality handles disconnected graphs naturally (unreachable vertices contribute 0 to the sum), unlike closeness centrality.

Use Cases

- Centrality measure robust to disconnected components
- Influence reach in sparse networks
- Alternative to closeness in disconnected graphs

Example

```
CALL algo.harmonic('KNOWS', 'BOTH', true)
YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 10
```

References

- Harmonic centrality – Wikipedia

20.4.6. algo.eigenvector

Property	Value
Procedure	`algo.eigenvector`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	4

Syntax

```
CALL algo.eigenvector([relTypes, direction, maxIterations, tolerance])
YIELD node, score
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Parameter	Type	Required	Default	Description
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`maxIterations`	Integer	No	`20`	Maximum power-iteration steps
`tolerance`	Double	No	`1e-6`	Convergence threshold

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`score`	Double	Eigenvector centrality score in [0, 1]

Description

Computes eigenvector centrality using the power iteration method. A node scores highly if it is connected to other high-scoring nodes. L∞ normalization (divide by maximum score) keeps values in [0, 1]. Convergence is checked against the maximum absolute change in scores between iterations.

Use Cases

- Identifying globally influential nodes
- Academic citation influence analysis
- Prestige scoring in directed networks

Example

```
CALL algo.eigenvector('CITES', 'IN', 50, 1e-8)
YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 10
```

References

- Eigenvector centrality – Wikipedia

20.4.7. algo.hits

Property	Value
Procedure	`algo.hits`
Category	Centrality
Complexity	CPU
Min Args	0

Property	Value
Max Args	3

Syntax

```
CALL algo.hits([relTypes, maxIterations, tolerance])
YIELD node, hubScore, authorityScore
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`maxIterations`	Integer	No	`20`	Maximum number of iterations
`tolerance`	Double	No	`1e-6`	Convergence threshold (max score change)

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`hubScore`	Double	Hub score (good hub points to good authorities)
`authorityScore`	Double	Authority score (good authority is pointed to by good hubs)

Description

Computes Hyperlink-Induced Topic Search (HITS) scores using alternating updates: authority scores are the sum of hub scores of incoming neighbors; hub scores are the sum of authority scores of outgoing neighbors. Both score vectors are L2-normalized after each iteration.

Use Cases

- Web link analysis (hubs = index pages, authorities = content pages)
- Identifying experts vs. curators in knowledge networks
- Bipartite influence analysis

Example

```
CALL algo.hits('LINKS_TO', 30, 1e-8)
YIELD node, hubScore, authorityScore
RETURN node, hubScore, authorityScore ORDER BY authorityScore DESC LIMIT 10
```

References

- HITS algorithm – Wikipedia
- Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5), 604–632.

20.4.8. algo.katz

Property	Value
Procedure	`algo.katz`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	4

Syntax

```
CALL algo.katz([relTypes, alpha, maxIterations, tolerance])
YIELD nodeId, score
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`alpha`	Double	No	`0.005`	Attenuation factor (must be less than 1/λ_max)
`maxIterations`	Integer	No	`100`	Maximum number of iterations
`tolerance`	Double	No	`1e-6`	Convergence threshold

Yield Fields

Field	Type	Description
`nodeId`	RID	Vertex identity
`score`	Double	Katz centrality score (normalized to [0, 1])

Description

Computes Katz centrality, which counts all paths between nodes with exponential decay by path length. Formula: `k[i] = alpha x Σ_j(A[j][i] x k[j]) + 1`, normalized by the maximum score. Unlike eigenvector centrality, every node receives a base score (the `+1` term), making it robust for directed and sparse graphs.

Use Cases

- Influence ranking in directed social networks
- Ranking nodes when direct connection is sparse
- Early-stage recommendation systems

Example

```
CALL algo.katz('FOLLOWS', 0.01, 50, 1e-8)
YIELD nodeId, score
RETURN nodeId, score ORDER BY score DESC LIMIT 10
```

References

- Katz centrality – Wikipedia

20.4.9. algo.voteRank

Property	Value
Procedure	`algo.voteRank`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.voteRank([relTypes, topK])
YIELD nodeId, rank
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`topK`	Integer	No	all nodes	Maximum number of top spreaders to return

Yield Fields

Field	Type	Description
`nodeId`	RID	Vertex identity
`rank`	Integer	1-based rank (1 = most influential spreader)

Description

Identifies the most influential spreaders in a network using an iterative election process. In each round, the node with the highest accumulated votes is selected as a spreader. After selection, each of its neighbors has its voting ability reduced by `1/degree`. This suppression mechanism ensures that the selected spreaders are structurally diverse rather than clustered together.

Since 26.12.1 an election recomputes only the scores within two hops of the elected vertex and the next one comes from a priority queue, with the same ranking as before. Before 26.12.1 every election rescanned all the vertices and recomputed every score, so the default call, which elects until no vertex has votes left, took over 90 seconds at 160,000 vertices of average degree 6.

Use Cases

- Viral marketing seed set selection
- Identifying structurally independent influencers
- Epidemic intervention: selecting individuals for immunization

Example

```
CALL algo.voteRank('KNOWS', 5)
YIELD nodeId, rank
RETURN nodeId, rank ORDER BY rank ASC
```

References

- Zhang, J., et al. (2016). Identifying a set of influential spreaders in complex networks. *Scientific Reports*, 6, 27823.

20.4.10. algo.eccentricity

Property	Value
Procedure	`algo.eccentricity`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.eccentricity([relTypes, direction])
YIELD node, eccentricity, isCenter, isPeripheral
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`eccentricity`	Integer	Maximum shortest-path distance to any reachable vertex
`isCenter`	Boolean	`true` if eccentricity equals the graph radius
`isPeripheral`	Boolean	`true` if eccentricity equals the graph diameter

Description

Computes the eccentricity of each vertex (the greatest shortest-path distance to any other reachable vertex) via BFS from each node. The graph radius is the minimum eccentricity and the graph diameter is the maximum. Center nodes have eccentricity equal to the radius; peripheral nodes have eccentricity equal to the diameter.

Use Cases

- Graph center identification for optimal facility placement
- Network diameter and radius computation
- Peripheral node analysis for vulnerability assessment

Example

```
CALL algo.eccentricity('ROAD', 'BOTH')
YIELD node, eccentricity, isCenter, isPeripheral
RETURN node, eccentricity, isCenter, isPeripheral ORDER BY eccentricity ASC
```

References

- Distance (graph theory) – Wikipedia

20.4.11. algo.personalizedPageRank

Property	Value
Procedure	`algo.personalizedPageRank`
Category	Centrality
Complexity	CPU
Min Args	1
Max Args	5

Syntax

```
CALL algo.personalizedPageRank(sourceNodes [, relTypes, dampingFactor, maxIterations,
tolerance])
YIELD nodeId, score

CALL algo.personalizedPageRank(sourceNodes, {relTypes, dampingFactor, maxIterations,
tolerance, direction, weightProperty})
YIELD nodeId, score
```

Parameters

The second argument is either the relationship types, followed by the other positional parameters, or a config map holding any of them plus `direction` and `weightProperty`, which have no position. A config map key not listed here is refused with an error.

Parameter	Type	Required	Default	Description
`sourceNodes`	Vertex or List	Yes	—	The personalization origin: a vertex, a list of vertices (equal weight), or a list of `[vertex, weight]` pairs. Plain vertices and pairs can be mixed (a plain vertex weighs 1)
`relTypes`	String	No	all types	Comma-separated relationship types
`dampingFactor`	Double	No	`0.85`	Probability of following an edge
`maxIterations`	Integer	No	`20`	Maximum power-iteration steps
`tolerance`	Double	No	`1e-6`	Convergence threshold
`direction`	String	No	`"OUT"`	Config map only. The edges the score travels along, with the values of `algo.pagerank`: `"OUT"` follows the stored edges, `"IN"` follows them backwards, `"BOTH"` treats the graph as undirected

Parameter	Type	Required	Default	Description
`weightProperty`	String	No	none	Config map only. Edge property holding the edge's weight: a vertex hands its score to its neighbors in proportion to the weights of the edges it follows (a missing or non-numeric value counts as 1). Weights must be finite and non-negative; a vertex whose edges all weigh 0 hands its score back to the sources, like a vertex with no edge

Yield Fields

Field	Type	Description
`nodeId`	RID	Vertex identity
`score`	Double	Personalized PageRank score relative to the source node

Description

Computes Personalized PageRank (PPR) relative to one or more source vertices. Unlike standard PageRank, the teleportation probability is concentrated at the source vertices (0 everywhere else). With a single vertex, all of it is on that vertex; with a list, it is split equally; with `[vertex, weight]` pairs, it follows the weights, which must be non-negative and are normalized to sum to 1 (only their ratios matter). A vertex listed twice counts twice. Scores represent structural proximity/importance relative to the sources. Dangling nodes contribute their rank back to the sources, in proportion to their weights.

The walk follows `OUT` edges unless `direction` says otherwise. A vertex nothing points at, which is the shape of any record just written that links to existing ones, therefore scores 0 from every source under `OUT`: use `direction: 'BOTH'` to reach it without storing every edge twice.

Before v26.11.1 the walk was always `OUT` and unweighted, with no way to ask for another direction or for edge weights.

Use Cases

- Node-centric similarity and proximity ranking
- Recommendation systems ("nodes similar to X")
- Query-centric graph exploration

Example

```
MATCH (s:Person {name:'Alice'})
CALL algo.personalizedPageRank(s, 'KNOWS', 0.85, 20, 0.000001)
```

```
YIELD nodeId, score
RETURN nodeId, score ORDER BY score DESC LIMIT 10
```

Weighted personalization vector (Alice counts three times as much as Bob):

```
MATCH (a:Person {name:'Alice'}), (b:Person {name:'Bob'})
CALL algo.personalizedPageRank([[a, 3.0], [b, 1.0]], 'KNOWS')
YIELD nodeId, score
RETURN nodeId, score ORDER BY score DESC LIMIT 10
```

Undirected walk over a similarity graph that keeps its score on the edge:

```
MATCH (d:Document {id: 42})
CALL algo.personalizedPageRank(d, {relTypes: 'SIMILAR', direction: 'BOTH',
weightProperty: 'similarity'})
YIELD nodeId, score
RETURN nodeId, score ORDER BY score DESC LIMIT 10
```

References

- Personalized PageRank – Wikipedia

20.4.12. algo.articlerank

Property	Value
Procedure	`algo.articlerank`
Category	Centrality
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.articlerank([config]) YIELD node, score
```

Parameters

Name	Type	Default	Description
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`dampingFactor`	Float	`0.85`	Damping factor (probability of following a link)
`maxIterations`	Integer	`20`	Maximum number of iterations
`tolerance`	Float	`0.0001`	Convergence threshold (sum of absolute score changes)

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`score`	Float	ArticleRank score for the vertex

Description

ArticleRank is a variant of PageRank that dampens the contribution of low-degree nodes. While standard PageRank divides a node's rank equally among its out-degree neighbours, ArticleRank divides by `outDegree(v) + avgOutDegree`. This reduces the influence of nodes with very few connections, producing more balanced centrality scores in sparse or scale-free graphs.

Formula: `AR(u) = (1 □ d) / N + d × Σ AR(v) / (outDeg(v) + avgOutDeg)`

Use Cases

- Web page importance ranking where low-citation pages should have less influence
- Academic citation networks with highly skewed degree distributions
- Any domain where hub dampening is desirable

Example

```
CALL algo.articlerank({dampingFactor: 0.85, maxIterations: 20})
YIELD node, score
RETURN node.name AS name, score
ORDER BY score DESC LIMIT 10
```

References

- ArticleRank: a PageRank-based alternative to numbers of citations – WWW 2004

20.5. Community Detection

20.5.1. algo.wcc

Property	Value
Procedure	`algo.wcc`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.wcc([relTypes])
YIELD node, componentId
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`componentId`	Integer	Sequential component identifier (0-based)

Description

Identifies weakly connected components using a parallel union-find, treating all edges as undirected regardless of their actual direction. Two vertices are in the same component if there exists any undirected path between them. Component IDs are remapped to sequential integers starting at 0.

Use Cases

- Data quality: detecting isolated subgraphs
- Network partition detection
- Graph reachability analysis

Example

```
CALL algo.wcc('CONNECTED_TO')
YIELD node, componentId
RETURN componentId, count(node) AS size ORDER BY size DESC
```

References

- Connected component – Wikipedia

20.5.2. algo.scc

Property	Value
Procedure	`algo.scc`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.scc([relTypes])
YIELD node, componentId
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`componentId`	Integer	Sequential component identifier (0-based)

Description

Identifies strongly connected components using Kosaraju's two-pass algorithm: iterative DFS on the original graph to compute finish-time ordering, then BFS on the reversed graph in reverse finish order. A strongly connected component is a maximal set of vertices where every vertex is reachable from every other.

Use Cases

- Deadlock detection in dependency graphs
- Circular dependency analysis in software systems
- Identifying mutually-reachable clusters in directed networks

Example

```
CALL algo.scc('DEPENDS_ON')
YIELD node, componentId
RETURN componentId, collect(node) AS members ORDER BY size(members) DESC
```

References

- Kosaraju's algorithm – Wikipedia

20.5.3. algo.louvain

Property	Value
Procedure	`algo.louvain`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.louvain([{maxIterations: 10, tolerance: 0.0001, weightProperty: null}])
YIELD node, communityId, modularity
```

Parameters

Parameter	Type	Required	Default	Description
config map	Map	No	see defaults	Configuration properties (all optional)
`maxIterations`	Integer	No	`10`	Maximum number of phase-1 passes
`tolerance`	Double	No	`0.0001`	Stop when modularity gain falls below this value
`weightProperty`	String	No	`null`	Edge property for weighted modularity

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity

Field	Type	Description
`communityId`	Integer	Sequential community identifier (0-based)
`modularity`	Double	Final modularity score of the partition

Description

Detects communities by maximizing modularity using the Louvain method: a greedy phase where each node is moved to the neighboring community that maximally increases modularity, iterated until no improvement is found. Community IDs are remapped to sequential 0-based integers.

Since 26.12.1 the total degree of each community is kept up to date as vertices move, so a pass costs O(E), and the graph is read once, through a Graph Analytical View when one covers it. Before 26.12.1 every candidate community of every vertex summed the degrees of all the vertices, and every pass read the edge records again: the call took over 90 seconds at 160,000 vertices of average degree 6. When several communities offer the same gain, the one with the lowest identifier is chosen. The vertices are visited in the order the graph numbers them, which a Graph Analytical View does its own way, so the partition can differ with and without a view.

Use Cases

- Social community discovery
- Topic clustering in citation networks
- Customer segmentation in purchase graphs

Example

```
CALL algo.louvain({maxIterations: 15, weightProperty: 'weight'})
YIELD node, communityId, modularity
RETURN communityId, collect(node) AS members, modularity ORDER BY size(members) DESC
```

References

- Louvain method – Wikipedia

20.5.4. algo.leiden

Property	Value
Procedure	`algo.leiden`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	3

Syntax

```
CALL algo.leiden([relTypes, maxIterations, resolution])
YIELD nodeId, community
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`maxIterations`	Integer	No	`10`	Maximum number of local-move + refinement iterations
`resolution`	Double	No	`1.0`	Resolution parameter γ (higher = more, smaller communities)

Yield Fields

Field	Type	Description
`nodeId`	RID	Vertex identity
`community`	Integer	Sequential community identifier (0-based)

Description

An improved community detection algorithm over Louvain, featuring a two-phase approach: (1) local greedy moves maximizing modularity, followed by (2) a refinement phase that checks whether nodes should be split out from their community to ensure well-connectedness. The resolution parameter γ controls granularity of the resulting communities.

Use Cases

- High-quality community detection with guaranteed well-connectedness
- Hierarchical community structure analysis
- Fine-grained cluster control via resolution parameter

Example

```
CALL algo.leiden('KNOWS', 10, 1.0)
YIELD nodeId, community
RETURN community, count(nodeId) AS size ORDER BY size DESC
```

References

- Traag, V. A., Waltman, L., & van Eck, N. J. (2019). From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*, 9, 5233.

20.5.5. algo.labelpropagation

Property	Value
Procedure	`algo.labelpropagation`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.labelpropagation([{maxIterations: 10, direction: 'BOTH', seedProperty:
'label', fixedSeeds: true}])
YIELD node, communityId
```

Parameters

Parameter	Type	Required	Default	Description
config map	Map	No	see defaults	Configuration properties (all optional). A key not listed here is refused with an error, so a misspelt one cannot silently fall back to its default
`maxIterations`	Integer	No	`10`	Maximum number of propagation passes
`direction`	String	No	`"BOTH"`	Edge direction to follow: `"IN"`, `"OUT"`, or `"BOTH"`
`tieBreakProperty`	String	No	none	Vertex property used to break ties between equally frequent labels: the smallest value wins (for example the vertex id, as LDBC Graphalytics CDLP requires). Without it, ties follow the internal load order, which a Graph Analytical View does not necessarily share: set it when the communities must not depend on whether a view serves the call. Values must be mutually comparable, and a vertex without the property loses every tie

Parameter	Type	Required	Default	Description
`seedProperty`	String	No	none	Vertex property holding a pre-assigned integer label. Every vertex carrying the same value starts in the same community, a vertex without the property starts with a label of its own. The `communityId` of a community grown from a seed is the seed value itself; any other community is numbered past the largest seed, so the two never collide (the convention of Neo4j GDS `seedProperty`). With `tieBreakProperty` too, a seed label ranks by the smallest tie-break value among the vertices carrying it
`fixedSeeds`	Boolean	No	`false`	Keeps every seeded vertex on its seed label for the whole run, while it still spreads that label to its neighbors. Requires `seedProperty`. Without it the seeds are only starting labels, which their neighbors can overwrite

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`communityId`	Integer	Community label assigned to this vertex: with `seedProperty`, the seed value of a seeded community

Description

Detects communities by iteratively propagating labels. In each pass every vertex adopts, at once, the most frequent label among the neighbors its edges in `direction` reach: `OUT` its successors, `IN` its predecessors, `BOTH` all of them. Ties are broken by choosing the smallest label (see `tieBreakProperty`). The passes are synchronous, so a run is deterministic for a given graph; it ends when no label moves or after `maxIterations` passes, since a graph can oscillate between two labellings forever.

With `seedProperty` and `fixedSeeds: true` the run is semi-supervised classification: the vertices whose class is known keep it, and every other vertex ends in the class of the seeds it is best connected to (or in a community of its own when no seed reaches it).

Before v26.10.2 a run served by a Graph Analytical View ignored `direction` and always propagated over `BOTH`, so `OUT` and `IN` answered different communities once a

view covered the graph.

Before v26.11.1 there was no way to pre-assign a label, and an unknown config key (such as `seedProperty` then, or a misspelt `maxIteration`) was silently ignored. Since v26.11.1 every `algo.*` procedure that takes a config map refuses a key it does not know.

Use Cases

- Fast community detection in very large graphs
- Semi-supervised classification when some labels are pre-assigned (`seedProperty` with `fixedSeeds: true`)
- Near-linear time community discovery

Example

```
CALL algo.labelpropagation({maxIterations: 20, direction: 'OUT'})
YIELD node, communityId
RETURN communityId, count(node) AS size ORDER BY size DESC
```

Semi-supervised classification: a few reviewed records carry their class in `label`, the graph classifies the rest:

```
CALL algo.labelpropagation({seedProperty: 'label', fixedSeeds: true})
YIELD node, communityId
WHERE node.label IS NULL
RETURN node.name AS record, communityId AS predictedLabel
```

References

- Label propagation algorithm – Wikipedia

20.5.6. algo.hierarchicalClustering

Property	Value
Procedure	`algo.hierarchicalClustering`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.hierarchicalClustering([relTypes, numClusters])
YIELD nodeId, cluster
```

Parameters

Parameter	Type	Required	Default	Description
relTypes	String	No	all types	Comma-separated relationship types
numClusters	Integer	No	2	Desired number of clusters

Yield Fields

Field	Type	Description
nodeId	RID	Vertex identity
cluster	Integer	Assigned cluster identifier

Description

Performs agglomerative (bottom-up) hierarchical clustering using single-linkage: each vertex starts in its own cluster, and the two clusters linked by the edge of highest similarity are merged iteratively until the target number of clusters is reached, or until no edge links two clusters (then one cluster per connected component is returned). The similarity of an edge is the number of neighbors its two ends share, each end counted in its own neighborhood. Among edges of equal similarity, the one met first scanning the vertices in order wins.

Since 26.12.1 the similarity of every edge is computed once, triangles are counted with the forward algorithm, and the merges run as Kruskal's algorithm over the edges in decreasing similarity: O(E $^{1.5}$) time and memory linear in the edges, with the same clusters as before. Before 26.12.1 every merge rescanned all the edges against a vertex-count by vertex-count bit matrix, which took 43 seconds at 10,000 vertices of average degree 6.

Use Cases

- Dendrogram-style cluster analysis
- Grouping vertices by structural neighborhood similarity
- Coarse-grained community assignment with controlled count

Example

```
CALL algo.hierarchicalClustering('SIMILAR_TO', 5)
YIELD nodeId, cluster
RETURN cluster, count(nodeId) AS size ORDER BY size DESC
```

References

- Hierarchical clustering – Wikipedia

20.5.7. algo.slpa

Property	Value
Procedure	`algo.slpa`
Category	Community Detection
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.slpa([config]) YIELD node, communities
```

Parameters

Name	Type	Default	Description
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`iterations`	Integer	`20`	Number of propagation iterations (T)
`threshold`	Float	`0.1`	Minimum label frequency in memory to be included in output
`seed`	Long	`-1`	Random seed for reproducibility; `-1` uses a random seed

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`communities`	List<Long>	List of community IDs the vertex belongs to (overlapping)

Description

SLPA (Speaker-Listener Label Propagation Algorithm) detects **overlapping** communities. Each node maintains a memory of labels it has received over T iterations. In each iteration every node acts as a speaker (emitting its most frequent memory label) and a listener (recording the most common label from its neighbours). After T rounds, labels appearing with frequency ≥ threshold are

retained as community assignments.

Unlike non-overlapping algorithms (Louvain, Label Propagation), SLPA allows each vertex to belong to multiple communities simultaneously, making it suitable for social networks where individuals naturally participate in multiple groups.

Use Cases

- Overlapping community detection in social networks
- Finding cross-cutting roles in collaboration networks
- Identifying nodes that bridge multiple communities

Example

```
CALL algo.slpa({iterations: 20, threshold: 0.1, seed: 42})
YIELD node, communities
RETURN node.name AS name, communities
ORDER BY size(communities) DESC
```

References

- Xie, J. et al. (2013). SLPA: Uncovering Overlapping Communities in Social Networks via a Speaker-listener Interaction Dynamic Process. *ICDMW 2011*

20.5.8. algo.maxKCut

Property	Value
Procedure	`algo.maxKCut`
Category	Community Detection
Complexity	CPU
Min Args	1
Max Args	2

Syntax

```
CALL algo.maxKCut(k [, config]) YIELD node, community, cutWeight
```

Parameters

Name	Type	Default	Description
`k`	Integer	—	Number of partitions (≥ 2)
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`maxIterations`	Integer	`100`	Maximum local-search passes per restart
`restarts`	Integer	`3`	Number of random restarts (best result returned)
`weightProperty`	String	`null`	Edge property for weights; `null` = all weights 1.0
`relTypes`	String	all types	Comma-separated edge type names
`direction`	String	`BOTH`	Edge traversal direction: `IN`, `OUT`, or `BOTH`
`seed`	Long	`-1`	Random seed; -1 = random

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`community`	Integer	Partition index in range [0, k − 1]
`cutWeight`	Float	Total weight of inter-partition edges (same on every row)

Description

Approximates the Maximum k-Cut problem: partition all vertices into k disjoint subsets to maximise the total weight of edges that cross between different partitions. Uses a greedy local-search heuristic:

1. Randomly assign each vertex to one of k partitions.
2. For each vertex, compute the cut gain for every possible partition and move it to the best one.
3. Repeat until no vertex can be moved to improve the cut (or `maxIterations` is reached).
4. Repeat steps 1–3 for `restarts` independent runs and return the best result found.

This heuristic achieves a `(k□1)/k` fraction of the optimal cut on unweighted graphs and converges quickly in practice.

Use Cases

- Graph partitioning for parallel processing
- Community detection via balanced cut
- Circuit bipartitioning and VLSI placement

Example

```
CALL algo.maxKCut(3, {restarts: 5, seed: 42})
YIELD node, community, cutWeight
```

```
RETURN node.name AS name, community
ORDER BY community
```

References

- Maximum cut – Wikipedia
- Sahni, S. & Gonzalez, T. (1976). P-Complete Approximation Problems. *Journal of the ACM*.

20.6. Structural Analysis

20.6.1. algo.kcore

Property	Value
Procedure	`algo.kcore`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.kcore([relTypes])
YIELD node, coreNumber
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`coreNumber`	Integer	Maximum k for which this vertex belongs to a k-core

Description

Computes the k-core decomposition of the graph using the Batagelj-Zaversnik O(V+E) algorithm with bucket sort. The k-core of a graph is the maximal subgraph in which every vertex has at least degree k. Each vertex is assigned its coreness: the maximum k for which it belongs to a k-core.

Use Cases

- Identifying dense graph cores and peripheral nodes
- Network robustness analysis
- Cohesive subgroup detection

Example

```
CALL algo.kcore('KNOWS')
YIELD node, coreNumber
RETURN node, coreNumber ORDER BY coreNumber DESC LIMIT 20
```

References

- Batagelj, V. & Zaversnik, M. (2003). An O(m) Algorithm for Cores Decomposition of Networks. *arXiv:cs/0310049*.

20.6.2. algo.triangleCount

Property	Value
Procedure	`algo.triangleCount`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.triangleCount([relTypes])
YIELD node, triangles, clusteringCoefficient
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity

Field	Type	Description
`triangles`	Integer	Number of triangles the vertex participates in
`clusteringCoefficient`	Double	Local clustering coefficient

Description

Counts the number of closed triangles for each vertex, by sorted-merge intersection of neighbor sets, with no memory beyond the adjacency and the result. The local clustering coefficient is computed as `2 x triangles / (deg x (deg-1))`, measuring how close the vertex's neighbors are to forming a complete subgraph.

Both are computed on the simple undirected graph, as NetworkX `triangles` and `clustering` do: three distinct vertices that are pairwise connected are one triangle whatever the direction or the number of edges between them, a self-loop counts for nothing, and `deg` is the number of distinct neighbors. `algo.localClusteringCoefficient` follows the same definition.

Since v26.11.1. Before, a self-loop, a parallel edge and an edge in both directions were counted as extra triangles (a vertex with a self-loop alone had 2).

Use Cases

- Social network cohesiveness analysis
- Spam detection (low clustering = suspicious accounts)
- Community structure pre-screening

Example

```
CALL algo.triangleCount('KNOWS')
YIELD node, triangles, clusteringCoefficient
RETURN node, triangles, clusteringCoefficient ORDER BY triangles DESC LIMIT 20
```

References

- Clustering coefficient – Wikipedia

20.6.3. algo.articulationPoints

Property	Value
Procedure	`algo.articulationPoints`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.articulationPoints([relTypes])
YIELD node
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity of an articulation point

Description

Finds all articulation points (cut vertices) — vertices whose removal would increase the number of connected components — using Tarjan's iterative DFS algorithm with discovery time and low-link arrays. Only vertices identified as articulation points are returned; all other vertices produce no result rows.

Use Cases

- Network resilience analysis: single points of failure
- Critical infrastructure node identification
- Dependency chain vulnerability assessment

Example

```
CALL algo.articulationPoints('CONNECTED_TO')
YIELD node
RETURN node
```

References

- Biconnected component – Wikipedia

20.6.4. algo.bridges

Property	Value
Procedure	`algo.bridges`

Property	Value
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.bridges([relTypes])
YIELD source, target
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`source`	RID	Source vertex of the bridge edge
`target`	RID	Target vertex of the bridge edge

Description

Finds all bridge edges — edges whose removal would disconnect the graph — using Tarjan's iterative DFS. A bridge is detected when the low-link value of the child strictly exceeds the discovery time of the parent: `low[v] > disc[parent]`. Bridges are a property of the undirected graph: the DFS walks every edge from both of its endpoints, so the stored direction of an edge plays no part in whether it is a bridge, and two parallel edges between the same pair of vertices are never bridges. Each bridge is returned oriented as the edge is stored (`source` → `target`). `algo.articulationPoints` and `algo.biconnectedComponents` share the same DFS.

Use Cases

- Network resilience: critical links whose failure causes disconnection
- Infrastructure planning: bottleneck link identification
- Supply chain single-path dependency detection

Example

```
CALL algo.bridges('LINK')
YIELD source, target
```

```
RETURN source, target
```

References

- Bridge (graph theory) – Wikipedia

20.6.5. algo.mst

Property	Value
Procedure	`algo.mst`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.mst([relTypes, weightProperty])
YIELD source, target, weight, totalWeight
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types (pass `null` for all types when only the weight is needed)
`weightProperty`	String	No	uniform weight	Edge property name for weights

Since 26.10.2 the relationship types come first, as in every other weighted `algo.*` procedure. Before that, `algo.mst` took `(weightProperty, relTypes)`. A call still written in the old order, such as `algo.mst('cost')`, is now rejected with "relationship type 'cost' does not exist": rewrite it as `algo.mst(null, 'cost')`.

Yield Fields

Field	Type	Description
`source`	RID	Source vertex of the MST edge
`target`	RID	Target vertex of the MST edge
`weight`	Double	Weight of this MST edge

Field	Type	Description
`totalWeight`	Double	Total weight of the entire MST (same on every row)

Description

Computes the minimum spanning tree of the graph using Kruskal's algorithm with a Union-Find data structure (path halving + union by rank). Edges are sorted by weight; each edge is added if it connects two different components. For disconnected graphs, a minimum spanning forest is returned.

Use Cases

- Network infrastructure with minimal cost
- Clustering by minimal connectivity
- Dendrogram construction for hierarchical analysis

Example

```
CALL algo.mst('CABLE', 'cost')
YIELD source, target, weight, totalWeight
RETURN source, target, weight, totalWeight ORDER BY weight ASC
```

References

- Kruskal's algorithm – Wikipedia

20.6.6. algo.topologicalSort

Property	Value
Procedure	`algo.topologicalSort`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.topologicalSort([relTypes])
YIELD node, order
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`order`	Integer	Topological position (0-based); `-1` if the node is part of a cycle

Description

Computes a topological ordering of the vertices of a directed acyclic graph (DAG) using Kahn's BFS algorithm. Nodes in cycles cannot be topologically sorted and receive `order = -1`. The algorithm processes nodes with zero in-degree first, progressively reducing in-degrees of successors.

Use Cases

- Build system dependency resolution
- Task scheduling with ordering constraints
- Compilation order determination

Example

```
CALL algo.topologicalSort('DEPENDS_ON')
YIELD node, order
RETURN node, order ORDER BY order ASC
```

References

- Topological sorting – Wikipedia

20.6.7. algo.bipartite

Property	Value
Procedure	`algo.bipartite`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.bipartite([relTypes])
YIELD node, partition, isBipartite
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`partition`	Integer	Partition assignment: `0` or `1`
`isBipartite`	Boolean	Global flag: `true` if the graph is bipartite

Description

Tests whether the graph is bipartite by attempting 2-coloring via BFS. A graph is bipartite if and only if it contains no odd-length cycles. The algorithm correctly handles disconnected graphs by running BFS from each unvisited component. All rows share the same `isBipartite` value.

Use Cases

- Verifying bipartite structure before applying bipartite-specific algorithms
- Recommendation system graph validation (users items)
- Conflict-free scheduling verification

Example

```
CALL algo.bipartite('CONNECTS')
YIELD node, partition, isBipartite
RETURN isBipartite, partition, collect(node) AS members
```

References

- Bipartite graph – Wikipedia

20.6.8. algo.graphColoring

Property	Value
Procedure	`algo.graphColoring`

Property	Value
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.graphColoring([relTypes])
YIELD node, color, chromaticNumber
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`color`	Integer	Assigned color (0-based integer)
`chromaticNumber`	Integer	Global chromatic number of the coloring (same on every row)

Description

Assigns colors to vertices such that no two adjacent vertices share the same color, using a greedy algorithm with a `forbidden[]` array to track unavailable colors for each vertex. The result is a valid coloring but not guaranteed to use the minimum number of colors (the chromatic number). The greedy approach works well in practice for most graph types.

Use Cases

- Exam/exam-room scheduling (students sharing courses cannot share a room)
- Frequency assignment in radio networks
- Register allocation in compilers

Example

```
CALL algo.graphColoring('CONFLICTS_WITH')
YIELD node, color, chromaticNumber
RETURN chromaticNumber, color, collect(node) AS group
```

References

- Graph coloring – Wikipedia

20.6.9. algo.cycleDetection

Property	Value
Procedure	`algo.cycleDetection`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.cycleDetection([relTypes])
YIELD node, inCycle, hasCycle
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`inCycle`	Boolean	`true` if the vertex is part of a cycle
`hasCycle`	Boolean	Global flag: `true` if the graph contains any cycle

Description

Detects cycles using Kosaraju's SCC algorithm. A vertex is considered to be in a cycle if it belongs to an SCC of size greater than 1, or if it has a self-loop. The global `hasCycle` flag is `true` if any vertex satisfies this condition.

Use Cases

- Dependency graph cycle detection (circular dependencies)
- Deadlock possibility analysis
- DAG validation before topological sort

Example

```
CALL algo.cycleDetection('DEPENDS_ON')
YIELD node, inCycle, hasCycle
RETURN hasCycle, node, inCycle
```

References

- Cycle detection – Wikipedia

20.6.10. algo.densestSubgraph

Property	Value
Procedure	`algo.densestSubgraph`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.densestSubgraph([relTypes])
YIELD node, inDenseSubgraph, density
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`node`	RID	Vertex identity
`inDenseSubgraph`	Boolean	`true` if the vertex is part of the densest subgraph found
`density`	Double	Edge density of the densest subgraph (same on every row)

Description

Finds the densest subgraph using Charikar's greedy peeling 2-approximation algorithm. The

algorithm iteratively removes the vertex with the lowest current degree, tracking the density (`edges / nodes`) of the remaining subgraph. The subgraph that achieved the maximum density is returned.

Since 26.12.1 the vertices are kept in buckets by current degree, so the call runs in O(V + E). Before 26.12.1 every peeled vertex was chosen with a scan of all of them (60 seconds at 160,000 vertices of average degree 6). Among vertices of equal degree the bucket order now decides instead of the lowest internal index, so on a graph with ties the subgraph reported can differ from the one an earlier version reported; both are valid 2-approximations. A self-loop now leaves the edge count with its vertex: before 26.12.1 it stayed counted, and the density reported for the subgraphs left after peeling such a vertex was too high.

Use Cases

- Identifying tightly-knit communities in social networks
- Quasi-clique detection
- Anomaly detection: unusually dense subgraphs

Example

```
CALL algo.densestSubgraph('KNOWS')
YIELD node, inDenseSubgraph, density
RETURN density, node, inDenseSubgraph ORDER BY inDenseSubgraph DESC
```

References

- Charikar, M. (2000). Greedy approximation algorithms for finding dense components in a graph. *APPROX 2000*.

20.6.11. algo.clique

Property	Value
Procedure	`algo.clique`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.clique([relTypes, minSize])
YIELD clique, size
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`minSize`	Integer	No	`3`	Minimum clique size to report

Yield Fields

Field	Type	Description
`clique`	List<RID>	List of vertex identities forming the clique
`size`	Integer	Number of vertices in the clique

Description

Enumerates all maximal cliques in the graph using the Bron-Kerbosch algorithm with Tomita pivoting, implemented iteratively with an explicit stack to avoid JVM stack overflow on deep recursion. The graph is treated as undirected and simple (both OUT and IN edges merged, self-loops and repeated edges ignored). Only cliques of size ≥ `minSize` are returned, each listing its members once. Since v26.12.1 the search runs once per vertex in degeneracy order (Eppstein, Löffler and Strash) over sorted neighbour sets, so its memory is linear in the graph and a sparse graph of hundreds of thousands of vertices is no longer refused by the working-memory budget; before, it kept a V×V bit matrix (issue #12173).

Use Cases

- Finding tightly connected groups (friend circles, co-author clusters)
- Network motif analysis
- Dense community detection

Example

```
CALL algo.clique('KNOWS', 4)
YIELD clique, size
RETURN size, clique ORDER BY size DESC LIMIT 10
```

References

- Bron-Kerbosch algorithm – Wikipedia

20.6.12. algo.kTruss

Property	Value
Procedure	`algo.kTruss`
Category	Structural Analysis

Property	Value
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.kTruss([relTypes, k])
YIELD nodeId, trussNumber
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`k`	Integer	No	none	Selects the k-truss: only the vertices with a truss number of at least `k` are returned (minimum 2). Omitted or `null`, every vertex is returned

Yield Fields

Field	Type	Description
`nodeId`	RID	Vertex identity
`trussNumber`	Integer	Maximum k for which this vertex belongs to a k-truss

Description

Computes the k-truss decomposition. A k-truss is a maximal subgraph where every edge participates in at least k−2 triangles. The algorithm removes edges with insufficient triangle support, and each vertex is assigned the maximum truss number of any of its incident edges in the full decomposition (0 for a vertex with no edge). The decomposition is defined on the simple undirected graph: direction, repeated edges and self-loops do not count. When `k` is passed, only the vertices of the k-truss are returned: those whose truss number is at least `k`. Without `k` (or with `null`) every vertex is listed with its truss number. A `k` below 2 is refused, since every edge belongs to the 2-truss. Before v26.12.1 `k` was validated but ignored (issue #12303). Since v26.12.1 the edges are peeled in support order over sorted neighbour sets, O(E^1.5) time and O(V + E) memory, instead of two V×V bit matrices, and a self-loop no longer gives the edges of its vertex a phantom triangle (issue #12173).

Use Cases

- Dense subgraph discovery beyond k-cores
- Social cohesion analysis (friend of a friend structures)

- Financial fraud ring detection

Example

```
CALL algo.kTruss('KNOWS', 4)
YIELD nodeId, trussNumber
RETURN nodeId, trussNumber ORDER BY trussNumber DESC LIMIT 20
```

Every vertex with its truss number, including those in no triangle (truss number 2) or in no edge (0):

```
CALL algo.kTruss('KNOWS')
YIELD nodeId, trussNumber
RETURN nodeId, trussNumber ORDER BY trussNumber DESC
```

References

- Cohen, J. (2008). Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*.

20.6.13. algo.biconnectedComponents

Property	Value
Procedure	`algo.biconnectedComponents`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.biconnectedComponents([relTypes]) YIELD node, componentId
```

Parameters

Name	Type	Default	Description
`relTypes`	String	all types	Comma-separated edge type names to traverse

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`componentId`	Integer	Biconnected component identifier

Description

Computes biconnected components using an iterative variant of Tarjan's algorithm with an explicit edge stack. A biconnected component is a maximal subgraph with no articulation point — removing any single vertex leaves the component connected. Articulation points (cut vertices) appear in multiple components because they belong to each component on either side of the cut.

The algorithm uses undirected traversal regardless of the edge direction parameter.

Use Cases

- Network resilience analysis (identifying single points of failure)
- Circuit board layout and VLSI design
- Social network structural analysis

Example

```
CALL algo.biconnectedComponents('ROAD')
YIELD node, componentId
RETURN componentId, collect(node.name) AS members
ORDER BY size(members) DESC
```

References

- Biconnected component – Wikipedia
- Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*.

20.6.14. algo.localClusteringCoefficient

Property	Value
Procedure	`algo.localClusteringCoefficient`
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.localClusteringCoefficient([relTypes]) YIELD node,
```

```
localClusteringCoefficient
```

Parameters

Name	Type	Default	Description
`relTypes`	String	all types	Comma-separated edge type names to traverse (treated as undirected)

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`localClusteringCoefficient`	Float	LCC value in range [0.0, 1.0]

Description

Computes the Local Clustering Coefficient (LCC) for every vertex. The LCC of a node v is the fraction of pairs of neighbours of v that are themselves connected: `LCC(v) = triangles(v) / (k*(k-1)/2)` where k is the degree of v. Nodes with degree < 2 receive a coefficient of 0.0. Triangle counting is performed with a BitSet intersection, making it efficient even for dense graphs.

Use Cases

- Identifying tightly-knit communities and cliques
- Detecting community structure (nodes with high LCC are embedded in local clusters)
- Social network cohesion analysis

Example

```
CALL algo.localClusteringCoefficient('KNOWS')
YIELD node, localClusteringCoefficient
RETURN node.name AS name, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC LIMIT 20
```

References

- Clustering coefficient – Wikipedia
- Watts, D. J. & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*.

20.6.15. algo.bipartiteMatching

Property	Value
Procedure	`algo.bipartiteMatching`

Property	Value
Category	Structural Analysis
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.bipartiteMatching([relTypes]) YIELD node1, node2, matchingSize
```

Parameters

Name	Type	Default	Description
`relTypes`	String	all types	Comma-separated edge type names defining the bipartite graph

Yield Fields

Field	Type	Description
`node1`	Vertex	Left-side matched vertex
`node2`	Vertex	Right-side matched vertex
`matchingSize`	Integer	Total number of matched pairs (same in every row)

Description

Computes a maximum bipartite matching using the Hopcroft-Karp algorithm (O(E√V)). The graph is treated as bipartite: nodes with only outgoing edges are on the left side; nodes with only incoming edges are on the right side. Each output row represents one matched pair. The algorithm finds augmenting paths via BFS (layered graph construction) followed by DFS, maximising the number of matched pairs in O(E√V) time.

Note: The procedure assumes the input graph is already bipartite. Results are undefined for non-bipartite inputs.

Use Cases

- Task-worker assignment optimisation
- Resource allocation (jobs to machines)
- Stable marriage / hospital-resident matching
- Recommendation filtering

Example

```
CALL algo.bipartiteMatching('ASSIGNED_TO')
YIELD node1, node2, matchingSize
RETURN node1.name AS worker, node2.name AS task, matchingSize
```

References

- Hopcroft-Karp algorithm – Wikipedia
- Hopcroft, J. E. & Karp, R. M. (1973). An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*.

20.7. Similarity and Link Prediction

20.7.1. algo.jaccard

Property	Value
Procedure	`algo.jaccard`
Category	Similarity
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.jaccard(node [, relTypes, direction, cutoff])
YIELD node1, node2, similarity
```

Parameters

Parameter	Type	Required	Default	Description
`node`	Vertex	Yes	—	Source vertex to compare against all others
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`cutoff`	Double	No	`0.0`	Minimum similarity threshold; pairs below are excluded

Yield Fields

Field	Type	Description
node1	RID	Source vertex identity
node2	RID	Compared vertex identity
similarity	Double	Jaccard similarity: |N(src) ∩ N(v)| / |N(src) ∪ N(v)|

Description

Computes the Jaccard similarity between the given source vertex and all other vertices, based on neighborhood overlap. Formula: `similarity = |N(src) ∩ N(v)| / |N(src) ∪ N(v)|`. Pairs with similarity below `cutoff` are filtered out. Results are listed in RID order. Since v26.12.1 only the vertices two hops from the source are read, since no other vertex can share a neighbour with it, so the call costs the size of the neighbourhood rather than of the database, with or without a Graph Analytical View (issue #12176).

Use Cases

- Collaborative filtering: "users with similar tastes"
- Friend recommendation based on shared connections
- Document similarity via shared keyword links

Example

```
MATCH (u:User {name:'Alice'})
CALL algo.jaccard(u, 'LIKES', 'OUT', 0.2)
YIELD node1, node2, similarity
RETURN node2, similarity ORDER BY similarity DESC LIMIT 10
```

References

- Jaccard index – Wikipedia

20.7.2. algo.adamicAdar

Property	Value
Procedure	algo.adamicAdar
Category	Link Prediction
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.adamicAdar(node [, relTypes, direction, cutoff])
YIELD node1, node2, score
```

Parameters

Parameter	Type	Required	Default	Description
`node`	Vertex	Yes	—	Source vertex
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`cutoff`	Double	No	`0.0`	Minimum score threshold

Yield Fields

Field	Type	Description
`node1`	RID	Source vertex identity
`node2`	RID	Compared vertex identity
`score`	Double	Adamic-Adar score; higher = more likely to form a link

Description

Computes the Adamic-Adar link prediction score between the source vertex and all non-adjacent vertices. For each pair, the score is the sum over common neighbors w of `1/log(degree(w))`. Highly-connected common neighbors contribute less than rare common neighbors. Results are sorted by score descending, equal scores in RID order. Since v26.12.1 only the vertices two hops from the source are read (issue #12176).

Use Cases

- Social network friend recommendation
- Knowledge graph link prediction
- Co-authorship prediction in citation networks

Example

```
MATCH (p:Person {name:'Bob'})
CALL algo.adamicAdar(p, 'KNOWS', 'BOTH', 0.5)
YIELD node1, node2, score
RETURN node2, score ORDER BY score DESC LIMIT 10
```

References

- Adamic, L. & Adar, E. (2003). Friends and neighbors on the web. *Social Networks*, 25(3), 211–230.

20.7.3. algo.commonNeighbors

Property	Value
Procedure	`algo.commonNeighbors`
Category	Link Prediction
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.commonNeighbors(node [, relTypes, direction, cutoff])
YIELD node1, node2, commonNeighbors
```

Parameters

Parameter	Type	Required	Default	Description
`node`	Vertex	Yes	—	Source vertex
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`cutoff`	Integer	No	`1`	Minimum number of common neighbors

Yield Fields

Field	Type	Description
`node1`	RID	Source vertex identity
`node2`	RID	Compared vertex identity
`commonNeighbors`	Integer	Count of shared neighbors

Description

Counts the number of common neighbors between the source vertex and all other vertices. Pairs with fewer than `cutoff` common neighbors are excluded. Results are sorted by count descending, equal counts in RID order. Since v26.12.1 only the vertices two hops from the source are read (issue #12176).

Use Cases

- Simple friend-of-a-friend recommendation
- Link prediction baseline
- Mutual connection analysis

Example

```
MATCH (u:User {name:'Charlie'})
CALL algo.commonNeighbors(u, 'FOLLOWS', 'BOTH', 3)
YIELD node1, node2, commonNeighbors
RETURN node2, commonNeighbors ORDER BY commonNeighbors DESC LIMIT 10
```

References

- Link prediction – Wikipedia

20.7.4. algo.preferentialAttachment

Property	Value
Procedure	`algo.preferentialAttachment`
Category	Link Prediction
Complexity	CPU
Min Args	1
Max Args	3

Syntax

```
CALL algo.preferentialAttachment(node [, relTypes, direction])
YIELD node1, node2, score
```

Parameters

Parameter	Type	Required	Default	Description
`node`	Vertex	Yes	—	Source vertex
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`

Yield Fields

Field	Type	Description
node1	RID	Source vertex identity
node2	RID	Compared vertex identity
score	Long	Product of the two nodes' degrees

Description

Computes the preferential attachment score for each pair as `degree(node1) × degree(node2)`. Based on the observation that in evolving networks, new links preferentially form between high-degree nodes. A simple but effective baseline for link prediction in scale-free networks.

Use Cases

- Scale-free network link prediction
- Social network "people you may know" baseline
- Network growth modeling

Example

```
MATCH (u:User {id:1})
CALL algo.preferentialAttachment(u, 'KNOWS', 'BOTH')
YIELD node1, node2, score
RETURN node2, score ORDER BY score DESC LIMIT 10
```

References

- Preferential attachment – Wikipedia

20.7.5. algo.resourceAllocation

Property	Value
Procedure	algo.resourceAllocation
Category	Link Prediction
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.resourceAllocation(node [, relTypes, direction, cutoff])
YIELD node1, node2, score
```

Parameters

Parameter	Type	Required	Default	Description
`node`	Vertex	Yes	—	Source vertex
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`cutoff`	Double	No	`0.0`	Minimum score threshold

Yield Fields

Field	Type	Description
`node1`	RID	Source vertex identity
`node2`	RID	Compared vertex identity
`score`	Double	Resource allocation score

Description

Computes the resource allocation link prediction score: `score = Σ 1/degree(w)` over all common neighbors w. Models the fraction of "resources" that each common neighbor can send to both endpoints if it distributes equally to all its neighbors. Conceptually similar to Adamic-Adar but uses degree directly rather than its logarithm. Results are sorted by score descending, equal scores in RID order. Since v26.12.1 only the vertices two hops from the source are read (issue #12176).

Use Cases

- Link prediction with emphasis on rare intermediaries
- Recommendation in sparse networks
- Metabolic network pathway prediction

Example

```
MATCH (p:Protein {id:'P1'})
CALL algo.resourceAllocation(p, 'INTERACTS', 'BOTH', 0.1)
YIELD node1, node2, score
RETURN node2, score ORDER BY score DESC LIMIT 10
```

References

- Zhou, T., Lü, L., & Zhang, Y. C. (2009). Predicting missing links via local information. *European Physical Journal B*, 71(4), 623–630.

20.7.6. algo.simRank

Property	Value
Procedure	`algo.simRank`
Category	Similarity
Complexity	CPU+RAM
Min Args	2
Max Args	5

Syntax

```
CALL algo.simRank(nodeA, nodeB [, relTypes, decayFactor, maxIterations])
YIELD similarity, nodeAId, nodeBId
```

Parameters

Parameter	Type	Required	Default	Description
`nodeA`	Vertex	Yes	—	First vertex
`nodeB`	Vertex	Yes	—	Second vertex
`relTypes`	String	No	all types	Comma-separated relationship types
`decayFactor`	Double	No	`0.8`	Decay constant C □ (0,1)
`maxIterations`	Integer	No	`5`	Number of refinement iterations

Yield Fields

Field	Type	Description
`similarity`	Double	SimRank similarity score between nodeA and nodeB
`nodeAId`	RID	Identity of nodeA
`nodeBId`	RID	Identity of nodeB

Description

Computes SimRank similarity between two vertices using the recursive definition: two nodes are similar if they are pointed to by similar nodes. Maintains a full V×V similarity matrix (O(V²) memory), updated iteratively. Similarity of a node with itself is 1.0. Uses IN adjacency for the standard directed graph formulation.

Use Cases

- Structural equivalence analysis in directed graphs

- Computing vertex pair similarity for matching problems
- Entity resolution in knowledge graphs

Example

```
MATCH (a:Paper {id:'P1'}), (b:Paper {id:'P2'})
CALL algo.simRank(a, b, 'CITES', 0.8, 5)
YIELD similarity, nodeAId, nodeBId
RETURN nodeAId, nodeBId, similarity
```

References

- Jeh, G. & Widom, J. (2002). SimRank: a measure of structural-context similarity. *KDD 2002*.

20.7.7. algo.knn

Property	Value
Procedure	`algo.knn`
Category	Similarity
Complexity	CPU
Min Args	0
Max Args	3

Syntax

```
CALL algo.knn([k, relTypes, direction]) YIELD node1, node2, similarity
```

Parameters

Name	Type	Default	Description
`k`	Integer	`5`	Number of nearest neighbours to return per node
`relTypes`	String	all types	Comma-separated edge type names to consider
`direction`	String	`"BOTH"`	Edge traversal direction: `IN`, `OUT`, or `BOTH`

Yield Fields

Field	Type	Description
`node1`	Vertex	Source vertex
`node2`	Vertex	One of the k nearest neighbours of node1

Field	Type	Description
`similarity`	Float	Jaccard similarity score in [0.0, 1.0]

Description

Constructs a k-Nearest Neighbours (KNN) graph using Jaccard similarity on neighbourhood sets. For each pair of nodes, the similarity is computed as `|N(u) ∩ N(v)| / |N(u) ∪ N(v)|`. Each node retains only its top-k neighbours by similarity score. This produces a new graph representing approximate structural proximity, useful as a pre-processing step for community detection or recommendation. Since v26.12.1 each vertex is compared only with the vertices two hops away, the only ones that can share a neighbour with it, instead of with every other vertex: the call is no longer cubic in the vertex count (21 s at 10,000 vertices before), and `k = 0` returns no rows instead of failing (issue #12172).

Use Cases

- Constructing similarity graphs for downstream community detection
- Collaborative filtering and recommendation
- Detecting structurally similar entities (users, documents, products)

Example

```
CALL algo.knn(10, 'KNOWS', 'BOTH')
YIELD node1, node2, similarity
RETURN node1.name AS source, node2.name AS neighbour, similarity
ORDER BY source, similarity DESC
```

References

- k-nearest neighbors algorithm – Wikipedia

20.7.8. algo.sameCommunity

Property	Value
Procedure	`algo.sameCommunity`
Category	Link Prediction
Complexity	CPU
Min Args	3
Max Args	3

Syntax

```
CALL algo.sameCommunity(node1, node2, communityProperty) YIELD node1, node2,
```

```
coefficient
```

Parameters

Name	Type	Default	Description
`node1`	Vertex	—	First vertex
`node2`	Vertex	—	Second vertex
`communityProperty`	String	—	Property name holding the community identifier on each vertex

Yield Fields

Field	Type	Description
`node1`	Vertex	The first vertex
`node2`	Vertex	The second vertex
`coefficient`	Float	`1.0` if both nodes share the same community value, `0.0` otherwise

Description

A simple link-prediction feature: returns `1.0` when both nodes carry the same value for the given community property and `0.0` otherwise. Useful as a binary feature in machine-learning pipelines or to filter / weight edges based on community membership.

Use Cases

- Feature engineering for link-prediction models
- Filtering candidate edges to intra-community connections
- Evaluating community detection quality (ground-truth comparison)

Example

```
MATCH (a:Person), (b:Person)
WHERE a <> b
CALL algo.sameCommunity(a, b, 'community')
YIELD coefficient
WHERE coefficient = 1.0
RETURN a.name, b.name
```

References

- Link prediction – Wikipedia

20.7.9. algo.totalNeighbors

Property	Value
Procedure	`algo.totalNeighbors`
Category	Link Prediction
Complexity	CPU
Min Args	2
Max Args	4

Syntax

```
CALL algo.totalNeighbors(node1, node2 [, relTypes, direction]) YIELD node1, node2,
coefficient
```

Parameters

Name	Type	Default	Description
`node1`	Vertex	—	First vertex
`node2`	Vertex	—	Second vertex
`relTypes`	String	all types	Comma-separated edge type names to traverse
`direction`	String	`"BOTH"`	Edge traversal direction: `IN`, `OUT`, or `BOTH`

Yield Fields

Field	Type	Description
`node1`	Vertex	The first vertex
`node2`	Vertex	The second vertex
`coefficient`	Long	Size of the union of the two neighbourhood sets

Description

Computes `|N(u) ∪ N(v)|` — the total number of distinct neighbours across both nodes. This is the denominator of Jaccard similarity and is used as a link-prediction feature measuring total neighbourhood reach. Calculated as `|N(u)| + |N(v)| − |N(u) ∩ N(v)|`. Since v26.12.1 only the neighbours of the two vertices are read, not the whole graph (issue #12176).

Use Cases

- Feature engineering for link-prediction models
- Identifying node pairs with large combined reach
- Pre-filtering candidates before more expensive similarity computation

Example

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CALL algo.totalNeighbors(a, b, 'KNOWS', 'BOTH')
YIELD coefficient
RETURN coefficient AS totalNeighbours
```

References

- Jaccard index – Wikipedia

20.8. Network Flow

20.8.1. algo.maxFlow

Property	Value
Procedure	`algo.maxFlow`
Category	Network Flow
Complexity	CPU+RAM
Min Args	2
Max Args	5

Syntax

```
CALL algo.maxFlow(sourceNode, sinkNode [, relTypes, capacityProperty, direction])
YIELD maxFlow, sourceId, sinkId
```

Parameters

Parameter	Type	Required	Default	Description
`sourceNode`	Vertex	Yes	—	Flow source vertex
`sinkNode`	Vertex	Yes	—	Flow sink vertex
`relTypes`	String	No	all types	Comma-separated relationship types
`capacityProperty`	String	No	uniform capacity	Edge property for flow capacity
`direction`	String	No	`OUT`	How a relationship carries flow: `OUT` from its source to its target, `IN` from its target to its source, `BOTH` either way (an undirected pipe)

Yield Fields

Field	Type	Description
`maxFlow`	Double	Maximum flow value from source to sink
`sourceId`	RID	Identity of the source vertex
`sinkId`	RID	Identity of the sink vertex

Description

Computes the maximum flow from a source to a sink using Dinic's algorithm over a sparse residual network: O(V²·E) time and O(V+E) memory. Every relationship is a pipe from its source to its target (with the default `direction OUT`), so flow never runs against a relationship, and relationships between the same two nodes in the same direction add their capacities. A self loop, or a capacity that is not positive, carries nothing. The flow from a node to itself is `0`. Returns a single row with the maximum achievable flow.

Before 26.12.1 every relationship was read as an undirected pipe, the behavior `direction BOTH` now selects (issue #11798).

Use Cases

- Network throughput capacity analysis
- Traffic flow optimization
- Supply chain max throughput calculation

Example

```
MATCH (src:Node {id:'source'}), (sink:Node {id:'sink'})
CALL algo.maxFlow(src, sink, 'PIPE', 'capacity')
YIELD maxFlow, sourceId, sinkId
RETURN maxFlow, sourceId, sinkId
```

References

- Dinic's algorithm – Wikipedia

20.9. Traversal and Sampling

20.9.1. algo.randomWalk

Property	Value
Procedure	`algo.randomWalk`
Category	Traversal / Sampling

Property	Value
Complexity	CPU
Min Args	2
Max Args	5

Syntax

```
CALL algo.randomWalk(startNode, steps [, relTypes, direction, seed])
YIELD path, steps
```

Parameters

Parameter	Type	Required	Default	Description
`startNode`	Vertex	Yes	—	Starting vertex
`steps`	Integer	Yes	—	Maximum number of steps to walk
`relTypes`	String	No	all types	Comma-separated relationship types
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`
`seed`	Long	No	current time	Random seed for reproducibility

Yield Fields

Field	Type	Description
`path`	Path	The walk: the nodes visited and the relationships stepped over (see reference/graph-algorithms/path-finding.pdf)
`steps`	Integer	Actual number of steps taken (may be less than requested if dead-end reached)

Description

Performs a single random walk starting from the given vertex. At each step, one neighbor is chosen uniformly at random from available edges matching the direction and type constraints. The walk stops when the requested number of steps is reached or a dead-end (no valid neighbors) is encountered. Since v26.12.1 the walk reads the neighbours of the vertices it steps on and nothing else, with or without a Graph Analytical View, so its cost follows the number of steps rather than the size of the database (issue #12176).

Use Cases

- Graph embedding feature generation (node2vec-style)

- Sampling large graphs for approximate analysis
- Stochastic exploration of graph neighborhoods

Example

```
MATCH (start:Page {id:'home'})
CALL algo.randomWalk(start, 20, 'LINKS_TO', 'OUT', 42)
YIELD path, steps
RETURN path, steps
```

References

- Random walk – Wikipedia

20.10. Network Science Metrics

20.10.1. algo.assortativity

Property	Value
Procedure	`algo.assortativity`
Category	Network Science
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.assortativity([relTypes])
YIELD assortativity, edgeCount
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`assortativity`	Double	Newman’s degree assortativity coefficient [-1, 1]

Field	Type	Description
edgeCount	Integer	Number of edges used in the computation

Description

Computes Newman's degree assortativity coefficient. A positive value indicates that high-degree nodes tend to connect to other high-degree nodes (assortative); negative indicates disassortative mixing.

Use Cases

- Network topology characterization
- Social network analysis (social networks are typically assortative)
- Comparing mixing patterns across different networks

Example

```
CALL algo.assortativity('KNOWS')
YIELD assortativity, edgeCount
RETURN assortativity, edgeCount
```

References

- Newman, M. E. J. (2002). Assortative mixing in networks. *Physical Review Letters*, 89(20), 208701.

20.10.2. algo.richClub

Property	Value
Procedure	algo.richClub
Category	Network Science
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.richClub([relTypes, minDegree])
YIELD degree, richClubCoefficient, nodeCount, edgeCount
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types
`minDegree`	Integer	No	`2`	Starting degree threshold

Yield Fields

Field	Type	Description
`degree`	Integer	Degree threshold k for this row
`richClubCoefficient`	Double	φ(k): density of edges among nodes with degree > k
`nodeCount`	Integer	Number of nodes with degree > k
`edgeCount`	Integer	Number of edges among those nodes

Description

Computes the rich-club coefficient φ(k) for every degree threshold k from `minDegree` to the maximum degree in the graph. One row is returned per degree threshold. Formula: `φ(k) = 2·E_k / (N_k·(N_k−1))`, where N_k = number of nodes with degree > k and E_k = edges between those nodes.

Use Cases

- Characterizing the "rich get richer" phenomenon
- Network topology analysis: do high-degree nodes interconnect?
- Comparing network structure across domains

Example

```
CALL algo.richClub('KNOWS', 2)
YIELD degree, richClubCoefficient, nodeCount, edgeCount
RETURN degree, richClubCoefficient ORDER BY degree ASC
```

References

- Colizza, V., Flammini, A., Serrano, M. A., & Vespignani, A. (2006). Detecting rich-club ordering in complex networks. *Nature Physics*, 2, 110–115.

20.10.3. algo.influenceMaximization

Property	Value
Procedure	`algo.influenceMaximization`
Category	Network Science

Property	Value
Complexity	CPU
Min Args	1
Max Args	4

Syntax

```
CALL algo.influenceMaximization(k [, relTypes, simulations, propagationProbability])
YIELD nodeId, rank, marginalGain
```

Parameters

Parameter	Type	Required	Default	Description
`k`	Integer	Yes	—	Number of seed nodes to select
`relTypes`	String	No	all types	Comma-separated relationship types
`simulations`	Integer	No	`100`	Monte Carlo simulation count per candidate
`propagationProbability`	Double	No	`0.1`	Independent Cascade activation probability per edge

Yield Fields

Field	Type	Description
`nodeId`	RID	Vertex identity of the selected seed node
`rank`	Integer	Selection rank (1 = best seed)
`marginalGain`	Double	Expected additional nodes activated by adding this seed

Description

Selects k seed nodes that maximize influence spread using a greedy algorithm with Monte Carlo simulation of the Independent Cascade (IC) model. In each round, the candidate that maximizes the expected spread when added to the current seed set is selected.

Use Cases

- Viral marketing campaign planning
- Identifying optimal vaccination targets for epidemic control
- Information cascade maximization in social media

Example

```
CALL algo.influenceMaximization(3, 'FOLLOWS', 200, 0.15)
YIELD nodeId, rank, marginalGain
RETURN nodeId, rank, marginalGain ORDER BY rank ASC
```

References

- Kempe, D., Kleinberg, J., & Tardos, E. (2003). Maximizing the spread of influence through a social network. *KDD 2003*.

20.11. Community Quality Metrics

20.11.1. algo.modularityScore

Property	Value
Procedure	`algo.modularityScore`
Category	Community Quality
Complexity	CPU
Min Args	1
Max Args	2

Syntax

```
CALL algo.modularityScore(communityProperty [, relTypes])
YIELD modularity, communities, edgeCount
```

Parameters

Parameter	Type	Required	Default	Description
`communityProperty`	String	Yes	—	Vertex property name holding the community label
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`modularity`	Double	Newman-Girvan modularity Q □ [-0.5, 1]
`communities`	Integer	Number of distinct communities found
`edgeCount`	Long	Total number of edges in the graph

Description

Evaluates the modularity of an existing community partition stored as a vertex property. Formula: `Q = Σ_c [ L_c/m − (d_c/(2m))² ]`, where L_c is the number of internal edges in community c, d_c is the sum of degrees in c, and m is the total number of edges. Returns a single row.

Use Cases

- Evaluating the output of community detection algorithms
- Comparing alternative community assignments
- Graph partition quality benchmarking

Example

```
CALL algo.modularityScore('community', 'KNOWS')
YIELD modularity, communities, edgeCount
RETURN modularity, communities, edgeCount
```

References

- Newman, M. E. J. & Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E*, 69(2), 026113.

20.11.2. algo.conductance

Property	Value
Procedure	`algo.conductance`
Category	Community Quality
Complexity	CPU
Min Args	1
Max Args	2

Syntax

```
CALL algo.conductance(communityProperty [, relTypes])
YIELD community, conductance, internalEdges, boundaryEdges, nodeCount
```

Parameters

Parameter	Type	Required	Default	Description
`communityProperty`	String	Yes	—	Vertex property name holding the community label

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types

Yield Fields

Field	Type	Description
`community`	Object	Community label value
`conductance`	Double	Conductance score (lower = better community)
`internalEdges`	Integer	Edges with both endpoints inside the community
`boundaryEdges`	Integer	Edges crossing the community boundary
`nodeCount`	Integer	Number of nodes in the community

Description

Computes the conductance of each community in an existing partition stored as a vertex property. Formula: `conductance = cut© / min(vol©, vol(V\C))`, where `cut©` is the number of boundary edges and `vol©` is the sum of degrees within C. Lower conductance indicates a better-separated community. Returns one row per community.

Use Cases

- Evaluating community separation quality
- Identifying poorly-separated or "leaky" communities
- Graph cut analysis for partitioning algorithms

Example

```
CALL algo.conductance('community', 'KNOWS')
YIELD community, conductance, internalEdges, boundaryEdges, nodeCount
RETURN community, conductance ORDER BY conductance ASC
```

References

- Conductance (graph) – Wikipedia

20.12. Graph Statistics

20.12.1. algo.graphSummary

Property	Value
Procedure	`algo.graphSummary`
Category	Graph Statistics
Complexity	CPU
Min Args	0
Max Args	2

Syntax

```
CALL algo.graphSummary([relTypes, nodeLabels])
YIELD nodeCount, edgeCount, avgDegree, maxDegree, minDegree, density, isolatedNodes,
selfLoops
```

Parameters

Parameter	Type	Required	Default	Description
`relTypes`	String	No	all types	Comma-separated relationship types to include
`nodeLabels`	String	No	all labels	Comma-separated node labels to include

Yield Fields

Field	Type	Description
`nodeCount`	Integer	Total number of vertices
`edgeCount`	Long	Total number of edges
`avgDegree`	Double	Average degree across all vertices
`maxDegree`	Integer	Maximum vertex degree
`minDegree`	Integer	Minimum vertex degree
`density`	Double	Graph density: `2E / (V x (V-1))`
`isolatedNodes`	Integer	Number of vertices with degree 0
`selfLoops`	Long	Number of self-loop edges

Description

Computes a comprehensive one-row statistical summary of the graph. Density is computed as `2E / (V x (V-1))` for undirected graphs (0 for graphs with ≤ 1 node). Self-loops are counted separately. Optionally filters by edge type and/or node label.

Use Cases

- Quick graph health check and characterization

- Monitoring graph evolution over time
- Input validation before running expensive algorithms

Example

```
CALL algo.graphSummary('KNOWS', 'Person')
YIELD nodeCount, edgeCount, avgDegree, maxDegree, density, isolatedNodes
RETURN nodeCount, edgeCount, avgDegree, maxDegree, density, isolatedNodes
```

20.13. Node Embedding

Node embedding procedures map each vertex to a dense real-valued vector that encodes its structural position in the graph. The resulting vectors can be used directly as features for machine-learning tasks (classification, clustering, link prediction) or for similarity search using ArcadeDB's vector index.

All embedding procedures yield a `List<Float>` that can be stored as a vertex property or forwarded to a downstream vector index.

20.13.1. algo.fastrp

Property	Value
Procedure	`algo.fastrp`
Category	Node Embedding
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.fastrp([config]) YIELD node, embedding
```

Parameters

Name	Type	Default	Description
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`dimensions`	Integer	`128`	Embedding vector size

Key	Type	Default	Description
`iterations`	Integer	`4`	Number of propagation rounds
`normalization`	Float	`0.0`	Degree-normalisation exponent α: weight of neighbour j contributing to node i is proportional to `deg(i)^{-α} × deg(j)^{-α}`; 0 = no normalisation, 1 = GCN-style
`selfInfluence`	Float	`0.0`	Weight [0,1] given to the node's own previous embedding vs. the aggregated neighbour embedding
`relTypes`	String	all types	Comma-separated edge type names
`direction`	String	`BOTH`	Edge traversal direction: `IN`, `OUT`, or `BOTH`
`seed`	Long	`-1`	Random seed; -1 = random

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`embedding`	List<Float>	L2-normalised embedding vector of length `dimensions`

Description

FastRP (Fast Random Projection, Chen et al. 2019) generates dense node embeddings without any training phase. Each node is initialised with a sparse ternary random vector (values ±√3 with probability 1/6 each, 0 with probability 2/3 — the optimal sparse random projection of Achlioptas 2003). The embedding is then iteratively refined by computing a weighted average of neighbour embeddings, followed by L2 normalisation. The resulting vectors capture multi-hop structural proximity and run orders of magnitude faster than walk-based methods.

Use Cases

- Fast baseline embeddings for downstream ML pipelines
- Similarity search via vector index
- Graph-aware feature generation without labelled data

Example

```
CALL algo.fastrp({dimensions: 64, iterations: 3, seed: 42})
YIELD node, embedding
RETURN node.name AS name, embedding
```

References

- Chen, H. et al. (2019). Fast and Accurate Network Embeddings via Very Sparse Random

Projection. *CIKM 2019*

20.13.2. algo.node2vec

Property	Value
Procedure	`algo.node2vec`
Category	Node Embedding
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.node2vec([config]) YIELD node, embedding
```

Parameters

Name	Type	Default	Description
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`embeddingDimension`	Integer	`128`	Embedding vector size
`walkLength`	Integer	`80`	Number of steps per random walk
`walksPerNode`	Integer	`10`	Random walks generated per node
`iterations`	Integer	`1`	Training epochs over all walks
`windowSize`	Integer	`10`	Skip-gram context window radius
`negSamples`	Integer	`5`	Negative samples per positive (centre, context) pair
`learningRate`	Float	`0.025`	Initial SGD learning rate (linearly decayed per epoch)
`p`	Float	`1.0`	Return parameter: high p → less likely to return to previous node
`q`	Float	`1.0`	In-out parameter: low q → DFS-like exploration; high q → BFS-like
`relTypes`	String	all types	Comma-separated edge type names
`direction`	String	`BOTH`	Edge traversal direction

Key	Type	Default	Description
`seed`	Long	`-1`	Random seed; -1 = random

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`embedding`	List<Float>	L2-normalised embedding vector of length `embeddingDimension`

Description

Node2Vec (Grover & Leskovec, 2016) learns node embeddings using biased second-order random walks combined with a Skip-gram model trained via negative sampling. The walk bias is controlled by two parameters:

- **p** (return parameter): low p makes the walk more likely to revisit the previous node (BFS-like neighbourhood exploration)
- **q** (in-out parameter): low q makes the walk explore outward (DFS-like), revealing structural roles; high q stays close to the source (community-based embeddings)

The Skip-gram model is trained with SGD and negative sampling, using a linearly decaying learning rate per epoch.

Use Cases

- Community-aware embeddings (high q)
- Structural-role embeddings (low q)
- Link prediction feature generation
- Node classification pre-training

Example

```
CALL algo.node2vec({embeddingDimension: 64, p: 1.0, q: 0.5, walkLength: 30, seed: 42})
YIELD node, embedding
RETURN node.name AS name, embedding
```

References

- Grover, A. & Leskovec, J. (2016). node2vec: Scalable Feature Learning for Networks. *KDD 2016*

20.13.3. algo.graphsage

Property	Value
Procedure	`algo.graphsage`
Category	Node Embedding
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.graphsage([config]) YIELD node, embedding
```

Parameters

Name	Type	Default	Description
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`embeddingDimension`	Integer	`64`	Output embedding size per layer
`layers`	Integer	`2`	Number of aggregation layers (receptive field = layers hops)
`relTypes`	String	all types	Comma-separated edge type names
`direction`	String	`BOTH`	Edge traversal direction
`seed`	Long	`-1`	Random seed; -1 = random

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`embedding`	List<Float>	L2-normalised embedding vector of length `embeddingDimension`

Description

GraphSAGE (Hamilton et al., 2017) — inductive neighbourhood aggregation. This implementation is **unsupervised**: no training labels are needed. Node features are initialised from structural properties (log-normalised degree + Gaussian noise) and then propagated through `layers` rounds of mean aggregation, each followed by a random linear projection (Xavier initialisation) and ReLU activation. The resulting embeddings capture multi-hop structural similarity.

Because the projection matrices are randomly initialised rather than trained, the embeddings are structurally consistent (nodes with similar neighbourhoods receive similar embeddings) and can be

used directly as ML features or refined with downstream fine-tuning.

Use Cases

- Inductive embeddings for graphs that grow over time
- Feature generation for node classification / link prediction
- Similarity search and clustering on graph data

Example

```
CALL algo.graphsage({embeddingDimension: 32, layers: 2, seed: 7})
YIELD node, embedding
RETURN node.name AS name, embedding
```

References

- Hamilton, W. et al. (2017). Inductive Representation Learning on Large Graphs. *NeurIPS 2017*

20.13.4. algo.hashgnn

Property	Value
Procedure	`algo.hashgnn`
Category	Node Embedding
Complexity	CPU
Min Args	0
Max Args	1

Syntax

```
CALL algo.hashgnn([config]) YIELD node, embedding
```

Parameters

Name	Type	Default	Description
`config`	Map	`{}`	Configuration map (see below)

Config Parameters

Key	Type	Default	Description
`embeddingDimension`	Integer	`128`	Output embedding size (max 4096)

Key	Type	Default	Description
`embeddingDensity`	Integer	`embeddingDimension / 8` (at least 1)	Samples drawn per node, so the most non-zero entries an embedding can have. At most `embeddingDimension`
`iterations`	Integer	`4`	Message-passing rounds: the farthest hop a sample can come from
`neighborInfluence`	Double	`1.0`	Weight of a node's neighbours, all together, against the node itself, applied again at every hop. Lower keeps the embedding local, higher reaches farther; `0` leaves every node with only itself
`relTypes`	String	all types	Comma-separated edge type names
`direction`	String	`BOTH`	Edge traversal direction
`seed`	Long	`-1`	Random seed; -1 = random

Yield Fields

Field	Type	Description
`node`	Vertex	The vertex
`embedding`	List<Double>	L2-normalised binary vector of length `embeddingDimension`: every set entry is `1 / sqrt(setCount)`, the others are 0

Description

HashGNN is a **training-free** graph neural network that replaces learned neighbourhood aggregation with MinHash sampling. Each node draws `embeddingDensity` samples from the nodes around it, and nodes whose neighbourhoods overlap draw the same samples: a node and its neighbours share many, two nodes far apart share none.

Each sample is a consistent weighted sample (an exponential race). Every node draws one random race time per sample slot from a hash shared by the whole graph, and each node keeps, per slot, the node with the smallest race time scaled by its distance: in every round a node may take its neighbours' best times multiplied by `degree / neighborInfluence`. So the node itself weighs 1, its neighbours together weigh `neighborInfluence` whatever the degree, and every further hop weighs `neighborInfluence` again of the one before. Because the hash is shared, two nodes keep the same sample with a probability that grows with how much of their weighted neighbourhoods they share. Each sample is then hashed to one of `embeddingDimension` positions, and the embedding is the normalised binary vector of the positions hit, so the cosine similarity of two embeddings measures how much their neighbourhoods overlap.

More `iterations` widen the neighbourhood without ever making every node identical: the node itself always keeps its weight. The samples are node identities rather than random features, so two unrelated nodes share a position only when two of their samples hash to it, with a probability of about `embeddingDensity / embeddingDimension` per sample.

HashGNN is extremely fast (no matrix multiplication, no gradient computation): a round costs `embeddingDensity` comparisons per edge, and it handles graphs without node features.

Before 26.11.1 the propagation rounds only OR-combined the neighbours' feature sets and reduced them once, after the last round, so on connected graphs of a few dozen nodes every node got the same vector, all zeros with the default settings (issue #11144). Embeddings computed by earlier versions are not comparable with the current ones.

Use Cases

- Ultra-fast structural embeddings as features for downstream ML
- Graph-level similarity via pooled embeddings
- Anomaly detection (unusual neighbourhood structure)

Example

```
CALL algo.hashgnn({embeddingDimension: 64, iterations: 3, seed: 42})
YIELD node, embedding
RETURN node.name AS name, embedding
```

References

- Wu, W., Li, B., Luo, C., Nejdl, W. (2021). Hashing-Accelerated Graph Neural Networks for Link Prediction. *The Web Conference 2021*
- Ioffe, S. (2010). Improved Consistent Sampling, Weighted Minhash and L1 Sketching. *ICDM 2010*
- MinHash – Wikipedia

20.14. SQL Path Functions

The following algorithms are also available as SQL functions that return a list of vertex RIDs (identities). They can be used in any SQL `SELECT` statement.

20.14.1. dijkstra() SQL Function

Property	Value
Function	`dijkstra()`
Category	Path Finding
Complexity	CPU

Syntax

```
dijkstra(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName> [, <direction>])
```

```
dijkstra(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName>,
         { direction: <string>, edgeTypeNames: [...], maxDepth: <long>,
emptyIfMaxDepth: <bool> })
```

Parameters

Parameter	Type	Required	Default	Description
`sourceVertex`	Vertex / RID	Yes	—	Source vertex
`destinationVertex`	Vertex / RID	Yes	—	Destination vertex
`weightEdgeFieldName`	String	Yes	—	Edge property name for weights
`direction`	String	No	`"OUT"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`. Can also be passed as `direction` key in the trailing options map.

The 4th argument can also be an options map with keys `direction`, `edgeTypeNames`, `maxDepth`, `emptyIfMaxDepth`. Unknown keys are rejected with a descriptive error.

`maxDepth` is the number of edges the path may have, as for `shortestPath()` and the Cypher `*..n`: the result is the cheapest path of at most `maxDepth` edges, which can be a dearer path than the cheapest one overall when that one is longer than the bound. `emptyIfMaxDepth` is accepted and has no effect: when no path fits the bound the result is an empty list, and a path that does not end at the destination is never returned.

`maxDepth` must be a non-negative whole number (a long), and `emptyIfMaxDepth` a boolean or the string `'true'` or `'false'` (in any case): any other value (`2.5`, `'abc'`, `-1`, `'yes'`, `1`) is rejected with an error that names the option. *(Since v26.11.1. Before, a boolean option read any other value as `false`.)*

Return Value

`List<RID>` — Ordered list of vertex identities on the shortest path (including source and destination), empty when there is none within `maxDepth` edges.

Description

SQL function equivalent of `algo.dijkstra`. Internally delegates to the A* implementation with no heuristic. Returns a `LinkedList` of vertex RIDs.

When several edges connect the same pair of vertices, the cheapest one is used. An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule.

Examples

```
-- Positional form (backward compatible)
SELECT dijkstra(#12:0, #12:5, 'weight', 'OUT') AS path

-- Options map form
SELECT dijkstra(#12:0, #12:5, 'weight', { direction: 'OUT', maxDepth: 20 }) AS path
```

See also: dijkstra() in SQL Functions reference and reference/graph-algorithms/path-finding.pdf for the Cypher procedure variant.

20.14.2. cchShortestPath() SQL Function

Property	Value
Function	`cchShortestPath()`
Category	Path Finding
Complexity	CPU

Syntax

```
cchShortestPath(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName> [,
<direction>])
cchShortestPath(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName>,
                { direction: <string>, edgeTypeNames: [...] })
```

Parameters

Parameter	Type	Required	Default	Description
`sourceVertex`	Vertex / RID	Yes	-	Source vertex
`destinationVertex`	Vertex / RID	Yes	-	Destination vertex
`weightEdgeFieldName`	String	Yes	-	Edge property name for weights
`direction`	String	No	`"OUT"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`. Can also be passed as `direction` key in the trailing options map.

The 4th argument can also be an options map with keys `direction` and `edgeTypeNames` (every edge type when absent). Unknown keys are rejected with a descriptive error.

Return Value

`List<RID>` - Ordered list of vertex identities on the shortest path (including source and destination),

empty when there is none.

Description

SQL function equivalent of algo.cch.shortestPath: answered by a Customizable Contraction Hierarchy when a Graph Analytical View keeps one for the weight and the exact edge types asked for, and by bidirectional Dijkstra otherwise. Exact either way, and inside a transaction it sees that transaction's own changes.

An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule.

Examples

```
SELECT cchShortestPath(#12:0, #12:5, 'distance', 'OUT') AS path
SELECT cchShortestPath(#12:0, #12:5, 'distance', { direction: 'OUT', edgeTypeNames:
['ROAD'] }) AS path
```

20.14.3. duanSSSP() SQL Function

Property	Value
Function	`duanSSSP()`
Category	Path Finding
Complexity	CPU

Syntax

```
duanSSSP(<sourceVertex>, <destinationVertex> [, <weightEdgeFieldName>] [, <direction
>])
duanSSSP(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName>,
         { direction: <string>, edgeTypeNames: [...] })
```

Parameters

Parameter	Type	Required	Default	Description
`sourceVertex`	Vertex / RID	Yes	-	Source vertex
`destinationVertex`	Vertex / RID	Yes	-	Destination vertex
`weightEdgeFieldName`	String	No	`"weight"`	Edge property name for weights

Parameter	Type	Required	Default	Description
`direction`	String	No	`"OUT"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`. Can also be passed as `direction` key in the trailing options map.

The 4th argument can also be an options map with keys `direction` and `edgeTypeNames` (every edge type when absent). Unknown keys are rejected with a descriptive error.

Return Value

`List<RID>` - Ordered list of vertex identities on the shortest path (including source and destination), empty when there is none.

Description

Named after Duan et al., Breaking the Sorting Barrier for Directed Single-Source Shortest Paths (2025). The paper's algorithm has constant factors too large to win at practical graph sizes, so the function answers through the same engine as cchShortestPath(): a Customizable Contraction Hierarchy when a Graph Analytical View keeps one, bidirectional Dijkstra otherwise. The only difference is that the weight property is optional here.

An edge without the weight property (or whose value is not a number) weighs 1, and an edge whose weight is negative, NaN or infinite is not walked: use `bellmanFord()` for negative weights. Every weighted path function reads the weight by this same rule.

Examples

```
SELECT duanSSSP(#12:0, #12:5) AS path
SELECT duanSSSP(#12:0, #12:5, 'distance', { direction: 'OUT', edgeTypeNames: ['ROAD']
}) AS path
```

20.14.4. bellmanFord() SQL Function

Property	Value
Function	`bellmanFord()`
Category	Path Finding
Complexity	CPU

Syntax

```
bellmanFord(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName> [, <direction
>])
bellmanFord(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName>, { direction:
```

```
<string> })
```

Parameters

Parameter	Type	Required	Default	Description
`sourceVertex`	Vertex / RID	Yes	—	Source vertex
`destinationVertex`	Vertex / RID	Yes	—	Destination vertex
`weightEdgeFieldName`	String	Yes	—	Edge property name for weights (may be negative)
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`. Can also be passed as `direction` key in the trailing options map.

Return Value

`List<RID>` — Ordered list of vertex identities on the shortest path.

Description

SQL function equivalent of `algo.bellmanford`. Supports negative edge weights and detects negative-weight cycles (returns an empty path if a negative cycle is reachable).

Example

```
SELECT bellmanFord(#12:0, #12:5, 'cost') AS path
SELECT bellmanFord(#12:0, #12:5, 'cost', { direction: 'OUT' }) AS path
```

See also: reference/graph-algorithms/path-finding.pdf for the Cypher procedure variant.

20.14.5. shortestPath() SQL Function

Property	Value
Function	`shortestPath()`
Category	Path Finding
Complexity	CPU

Syntax

```
shortestPath(<sourceVertex>, <destinationVertex>)
shortestPath(<sourceVertex>, <destinationVertex>, <direction>, <edgeType>, { maxDepth,
```

```
edge })
shortestPath(<sourceVertex>, <destinationVertex>,
             { direction: <string>, edgeType | edgeTypeNames: <string|list>, maxDepth:
<int>, edge: <bool> })
```

Parameters

Parameter	Type	Required	Default	Description
`sourceVertex`	Vertex / RID	Yes	—	Source vertex
`destinationVertex`	Vertex / RID	Yes	—	Destination vertex
`direction`	String	No	`"BOTH"`	Traversal direction: `"IN"`, `"OUT"`, or `"BOTH"`. Positional or map key.
`edgeType`	String / List	No	all types	Edge type(s) to restrict traversal. Positional or map key (`edgeType` or `edgeTypeNames`).
`maxDepth`	Int	No	unlimited	Stop after this many hops. Options-map only (or via the trailing map in the positional form).
`edge`	Bool	No	`false`	Include edge RIDs in the result. Options-map only.

The 3rd argument can be a direction string (backward compatible) OR an options map consolidating all knobs. Unknown keys are rejected with a descriptive error. The trailing map of the positional form (5th argument) accepts only `maxDepth` and `edge`: `direction` and `edgeType` / `edgeTypeNames` written there are rejected instead of being ignored.

`maxDepth` must be a non-negative whole number, and `maxDepth: null` means no bound. `edge` takes a boolean or the string `'true'` or `'false'` (in any case). Any other value (`2.5`, `'abc'`, `-1`, `'yes'`, `1`) is rejected with an error that names the option. *(Since v26.11.1. Before,* `maxDepth: null` *was read as 0 and* `edge` *read any other value as* `false`*.)*

Return Value

`List<RID>` — Ordered list of vertex identities on the shortest path (unweighted; minimizes hop count).

Description

Finds the unweighted shortest path (minimum hop count) between two vertices using bidirectional BFS (meets in the middle), which can be significantly faster than single-direction BFS for large graphs.

Examples

```
-- Positional form (backward compatible)
SELECT shortestPath(#12:0, #12:9, 'BOTH', 'KNOWS') AS path

-- Options map form
SELECT shortestPath(#12:0, #12:9, { direction: 'BOTH', edgeTypeNames: ['KNOWS'],
maxDepth: 6 }) AS path
```

References

- Bidirectional search – Wikipedia

See also: shortestPath() in SQL Functions reference.

20.15. Notes

20.15.1. relTypes Parameter

All procedures that accept a `relTypes` parameter use comma-separated relationship type names. To traverse **all** relationship types, pass an empty string `''` or omit the parameter where it is optional:

```
CALL algo.pagerank()              -- all types (parameter omitted)
CALL algo.wcc('')                 -- all types (empty string)
CALL algo.wcc('KNOWS,FOLLOWS')    -- only KNOWS and FOLLOWS edges
```

20.15.2. direction Parameter

Where supported, the `direction` parameter controls which edges are considered:

- `"OUT"` — only outgoing edges from each vertex
- `"IN"` — only incoming edges to each vertex
- `"BOTH"` — edges in either direction (default for most algorithms; `algo.pagerank` defaults to `"OUT"`)

The value is case-insensitive. Any other value is rejected with an error:

```
CALL algo.bfs(start, 'ROAD', 'INCOMING', 3)
-- unknown direction 'INCOMING', expected one of OUT, IN or BOTH
```

Earlier versions silently treated an unrecognised value as `"BOTH"`, so a typo returned a plausible but wrong result instead of an error. If a query of yours passes a malformed direction, it will now fail rather than quietly traverse both directions. Omitting the parameter still selects the algorithm's own default.

The same applies to the `bellmanFord()` SQL function and to `node.degree()`, `node.relationship.exists()` and `node.relationship.types()`.

20.15.3. Unidirectional Edge Types

Since 26.10.2 the `algo. and path.` procedures, the path-finding SQL functions and the `node.degree*` / `node.relationship.*` functions follow the edges of a `UNIDIRECTIONAL` edge type from their target too, so `"IN"` and `"BOTH"` answer the same as with a bidirectional type or a Graph Analytical View. See `CREATE TYPE` for the heap a vertex-by-vertex walk takes.

20.15.4. Dangling Edge Entries

An `algo.*` procedure reads the adjacency of the vertices without loading the edge records, with or without a Graph Analytical View. When the record of an edge is gone but the adjacency lists still name it (a dangling entry, left behind by a data-integrity anomaly), the procedure follows the entry. Before 26.12.1 a procedure running without a view loaded every edge and skipped such an entry, so its answer could differ from the one with a view. `CHECK DATABASE FIX` removes the dangling entries.

20.15.5. Config Map Parameters

Several algorithms (PageRank, Louvain, Betweenness, LabelPropagation) accept an optional configuration map as their first argument:

```
CALL algo.pagerank({dampingFactor: 0.9, maxIterations: 50})
YIELD node, score
```

Since v26.11.1 a key the procedure does not know is refused with an error that lists the accepted ones, rather than ignored: every key is optional, so a misspelt one (`maxIteration`, `weightProprety`) used to run with the default in its place and answer a plausible-looking result.

20.15.6. Memory Considerations

Algorithms marked **CPU+RAM** build in-memory data structures that scale with V^2 or E^2:

Algorithm	Memory Usage
`algo.apsp`	O(V^2) distance matrix
`algo.maxFlow`	O(V^2) capacity matrix
`algo.kShortestPaths`	O(V + E) adjacency, k × path length searches
`algo.simRank`	O(V^2) similarity matrix
`algo.triangleCount`	O(V^2) neighbor bit-matrix
`algo.knn`	O(V + E) neighbour sets, plus two node-sized counters (a neighbour bit-matrix before v26.12.1)
`algo.kTruss`	O(V + E) sorted neighbour sets and edge arrays (two neighbour bit-matrices before v26.12.1)

Algorithm	Memory Usage
`algo.clique`	O(V + E) sorted neighbour sets, plus a search stack bounded by the neighbourhood each search starts from (a neighbour bit-matrix before v26.12.1)

Every allocation above — including the graph itself (the RIDs of the vertices it loads when no Graph Analytical View covers it, and the adjacency list every procedure builds from it, not only the table in this list) — is checked against the `arcadedb.cypher.algoMaxWorkingMemory` setting before it is allocated, and the call is refused with a client error naming the offending component rather than risking an `OutOfMemoryError`. Because the setting now also covers the graph load itself, a call that used to succeed under a given limit may now be refused if the graph alone consumes a large share of that limit — raise the setting if a call that previously worked starts being refused after upgrading. Since v26.11.1 the default (0) sizes the budget at each call from the heap the call can actually get: half of the heap free after the last collection plus the page cache, which the call takes back from the cache before it allocates, and never less than one eighth of the maximum heap (64 MB at least). Before v26.11.1 the default was that fixed eighth, which refused calls the heap had room for, such as `algo.wcc()` on a 2 GB heap over a graph whose adjacency needed 280 MB (issue #11011). A positive value is still a fixed budget.

Since v26.12.1 a call that no Graph Analytical View serves holds only the RID of every vertex (16 bytes each) and reads a vertex record when it needs one. Before, it held every vertex record and an index over them, which took several times what the call reserved, so two calls at once could each pass the check and then run out of heap instead of the second being refused (issue #12554).

For graphs with more than a few thousand vertices, these algorithms may still require significant heap space even within the budget.

Chapter 21. Extended Functions

21.1. Extended Functions Reference

ArcadeDB provides a comprehensive set of extended functions organized by namespace. These functions are available in both SQL and Cypher queries, providing powerful data manipulation capabilities.

21.1.1. Overview

These extended functions provide:

- **Zero configuration** - Functions are available immediately without any installation
- **Native performance** - Direct integration with the query engine
- **Type safety** - Proper type handling and error messages
- **Cross-language support** - Available in both SQL and Cypher queries
- **APOC compatibility** - Full support for the `apoc.` prefix for Neo4j migration

For a complete reference of all graph algorithms including complexity analysis, use cases, and examples, see the Graph Algorithms Appendix.

21.1.2. APOC Prefix Compatibility

ArcadeDB automatically supports the `apoc.` prefix for all extended functions and procedures. This means existing Neo4j/APOC queries work without modification.

When ArcadeDB encounters a function or procedure call with the `apoc.` prefix, it automatically strips the prefix and resolves to the corresponding ArcadeDB function:

APOC Call	Resolves To	Result
`apoc.text.indexOf("hello", "l")`	`text.indexOf("hello", "l")`	Same function
`apoc.map.merge({a:1}, {b:2})`	`map.merge({a:1}, {b:2})`	Same function
`CALL apoc.merge.relationship(…)`	`CALL merge.relationship(…)`	Same procedure

This compatibility layer is:

- **Automatic** - No configuration required
- **Case-insensitive** - `APOC.TEXT.INDEXOF` works the same as `apoc.text.indexOf`
- **Zero overhead** - Simple string prefix check, no performance impact

Example - These queries are equivalent:

```
-- Neo4j/APOC style (works in ArcadeDB)
RETURN apoc.text.join(["a", "b", "c"], ",") AS result
```

```
-- ArcadeDB native style
RETURN text.join(["a", "b", "c"], ",") AS result
```

21.1.3. Function Namespaces

Functions are organized into the following namespaces:

Namespace	Description
`agg.*`	Aggregation and collection functions
`algo.*`	Graph algorithms (Dijkstra, A*, shortest paths, and 45+ more — see Graph Algorithms Appendix)
`coll.*`	Collection/list operations (flatten, sort, distinct, min/max)
`convert.*`	Type conversion functions
`create.*`	Creation functions (UUIDs, virtual nodes/relationships)
`date.*`	Date/time operations
`do.*`	Conditional execution procedures
`map.*`	Map/object operations
`math.*`	Mathematical functions
`merge.*`	Merge procedures for nodes and relationships
`meta.*`	Schema and database introspection procedures
`node.*`	Node/vertex operations (degree, labels, relationships)
`number.*`	Number formatting
`path.*`	Path operations (create, combine, slice, expansion)
`refactor.*`	Graph refactoring procedures (merge nodes, clone subgraphs)
`rel.*`	Relationship/edge operations (type, endpoints)
`text.*`	String manipulation and text processing
`util.*`	Utility functions (hashing, compression, validation)
`vector.*`	Vector/embedding operations (similarity, distance, quantization)

21.1.4. Function Categories

Detailed reference is split per category. Pick a category from the left navigation, or jump straight in:

- Aggregation — `agg.first`, `agg.last`, `agg.median`, percentiles
- Collection & Map — `coll.*` and `map.*` operations
- Convert & Create — type conversion, UUIDs, virtual nodes
- Date & Math — date/time operations, extended math, number formatting

- Graph Elements — `node.*` and `rel.*` operations
- Path & Algorithms — `algo.*` procedures and path expansion
- Refactor & Control Flow — `refactor.*` and `do.*` procedures
- Schema & Meta — `meta.*` and `merge.*` procedures
- Text — string manipulation, similarity, distance
- Utility — hashing, compression, validation
- Vector — similarity, distance, quantization, sparse vectors

21.1.5. Migration Guide

Converting APOC Queries to ArcadeDB

Good news: In most cases, no changes are required!

ArcadeDB automatically supports the `apoc.` prefix, so your existing Neo4j/APOC queries will work without modification:

```
-- This Neo4j/APOC query works directly in ArcadeDB
RETURN apoc.text.join(["a", "b"], ",")

-- This procedure call also works directly
CALL apoc.merge.relationship(a, "KNOWS", {}, {}, b) YIELD rel
```

Optional: Removing the APOC Prefix

If you prefer to use the cleaner ArcadeDB-native syntax, you can optionally remove the `apoc.` prefix:

1. **Functions** - Remove the `apoc.` prefix

   ```
   -- Neo4j/APOC style (works in ArcadeDB)
   RETURN apoc.text.join(["a", "b"], ",")

   -- ArcadeDB native style (also works)
   RETURN text.join(["a", "b"], ",")
   ```

2. **Procedures** - Remove the `apoc.` prefix

   ```
   -- Neo4j/APOC style (works in ArcadeDB)
   CALL apoc.merge.relationship(a, "KNOWS", {}, {}, b) YIELD rel

   -- ArcadeDB native style (also works)
   CALL merge.relationship(a, "KNOWS", {}, {}, b) YIELD rel
   ```

3. **Function signatures are 100% compatible** - Parameters are in the same order with the same semantics

Unsupported APOC Functions

Some APOC procedures are not yet implemented in ArcadeDB:

- Periodic/batch operations (`apoc.periodic.*`) - Use ArcadeDB's transaction API
- Schema modification operations (`apoc.schema.*`) - Use ArcadeDB's Schema API
- Export/import procedures (`apoc.export.*`, `apoc.import.*`) - Use ArcadeDB's native import/export

Of the `apoc.refactor.*` procedures, `apoc.refactor.mergeNodes` and `apoc.refactor.cloneNodesWithRelationships` are supported — see Refactor & Control Flow. The rest of the namespace is not implemented; use ArcadeDB's native capabilities instead.

21.2. Aggregation Functions

The `agg.*` functions below take one list argument and return one value per row: `RETURN agg.median([1, 2, 3, 4])` returns `2.5`.

`apoc.agg.` in Cypher aggregates over rows.* The same functions called with the `apoc.agg.` prefix behave as in APOC: like `avg()` or `collect()`, they fold the rows of each group into one value, and they skip null values.

```
UNWIND [1, 2, 3, 4] AS x RETURN apoc.agg.median(x) AS result
// Returns one row: 2.5

UNWIND [{n: 'a', v: 1}, {n: 'b', v: 5}, {n: 'c', v: 5}] AS m RETURN apoc.agg.maxItems
(m.n, m.v) AS result
// Returns one row: {items: ["b", "c"], value: 5}
```

Function	Returns, for the rows of a group
`apoc.agg.first(value)`, `apoc.agg.last(value)`	The first or the last non-null value
`apoc.agg.nth(value, offset)`	The non-null value at the 0-based `offset`, the last one for `-1`, `null` past the end
`apoc.agg.slice(value, from = 0, to = -1)`	A list of the non-null values from the `from`-th on, at most `to` of them: as in APOC, `to` is a length, and `-1` means all
`apoc.agg.median(value)`	The median of the numbers as a FLOAT, `null` for no numbers
`apoc.agg.product(value)`	The product, an INTEGER while every value is one (overflow is an error), a FLOAT otherwise, `0.0` for no rows
`apoc.agg.percentiles(value, percentiles = [0.5, 0.75, 0.9, 0.95, 0.99])`	A list with the value at each percentile

Function	Returns, for the rows of a group
`apoc.agg.statistics(value, percentiles = [0.5, 0.75, 0.9, 0.95, 0.99])`	A map with `total`, `min`, `minNonZero`, `max`, `mean`, `stdev` (population) and one key per percentile (`"0.5"`, …)
`apoc.agg.maxItems(item, value, groupLimit = -1)`, `apoc.agg.minItems(…)`	`{items, value}`: the highest (lowest) value and the items that carry it, at most `groupLimit` of them (all for a negative one)

Unlike APOC, percentile and statistics values are exact rather than read from histogram buckets, negative numbers are accepted, and `maxItems`/`minItems` can compare INTEGER and FLOAT values with each other.

21.2.1. agg.first()

Get the first non-null element from a list.

Syntax: `agg.first(list)`

Returns: Any - First non-null value

APOC Compatible: `apoc.agg.first`

```
RETURN agg.first([null, "a", "b"]) AS result
// Returns: "a"
```

21.2.2. agg.last()

Get the last non-null element from a list.

Syntax: `agg.last(list)`

Returns: Any - Last non-null value

APOC Compatible: `apoc.agg.last`

21.2.3. agg.maxItems()

Get items with maximum value.

Syntax: `agg.maxItems(list, valueFunction)`

Returns: List - Items with maximum value

APOC Compatible: `apoc.agg.maxItems`

21.2.4. agg.median()

Calculate the median of a list of numbers.

Syntax: `agg.median(list)`

Returns: Double - Median value

APOC Compatible: `apoc.agg.median`

```
RETURN agg.median([1, 2, 3, 4, 5]) AS result
// Returns: 3.0

RETURN agg.median([1, 2, 3, 4]) AS result
// Returns: 2.5
```

21.2.5. agg.minItems()

Get items with minimum value.

Syntax: `agg.minItems(list, valueFunction)`

Returns: List - Items with minimum value

APOC Compatible: `apoc.agg.minItems`

21.2.6. agg.nth()

Get the nth element from a list.

Syntax: `agg.nth(list, n)`

Returns: Any - Element at index n

APOC Compatible: `apoc.agg.nth`

21.2.7. agg.percentiles()

Calculate percentiles of a list of numbers.

Syntax: `agg.percentiles(list, percentiles)`

Returns: List - Percentile values

APOC Compatible: `apoc.agg.percentiles`

21.2.8. agg.product()

Calculate the product of a list of numbers.

Syntax: `agg.product(list)`

Returns: Double - Product of all values

APOC Compatible: `apoc.agg.product`

21.2.9. agg.slice()

Get a slice of a list.

Syntax: `agg.slice(list, start, [end])`

Returns: List - Sliced list

APOC Compatible: `apoc.agg.slice`

21.2.10. agg.statistics()

Get full statistics for a list of numbers.

Syntax: `agg.statistics(list)`

Returns: Map - Statistics map with count, min, max, sum, mean, stdev, median

APOC Compatible: `apoc.agg.statistics`

```
RETURN agg.statistics([1, 2, 3, 4, 5]) AS stats
// Returns: {count: 5, min: 1.0, max: 5.0, sum: 15.0, mean: 3.0, stdev: 1.414, median:
3.0}
```

Collection & Map Functions

21.2.11. Collection Functions

The `list` argument accepted by every `coll.*` function below follows the same rules as the built-in Cypher list functions (`head()`, `tail()`, `reverse()`, `size()`): a Cypher `LIST`, a numeric array parameter, or any `Iterable`/`Iterator` are all accepted. An argument that is none of these — a string, a number, a map — raises a client-facing `Type mismatch` error rather than succeeding silently or failing with a generic server error.

`coll.toSet()` and `coll.distinct()` are the same function under two names: both preserve the order of first occurrence and both recognize duplicates by object equality, so `coll.toSet([1, 1.0])` and `coll.distinct([1, 1.0])` both keep both elements.

21.2.12. coll.avg()

Returns the arithmetic mean of a list of numbers.

Syntax: `coll.avg(list)`

Parameters:

Parameter	Type	Description
list	List	A list of numbers

Returns: Double - The average, `null` if the list is null, empty, or contains only nulls

Null elements are skipped rather than counted, so `coll.avg([1, 2, null])` averages over the two non-null elements. A non-numeric element raises an error.

APOC Compatible: `apoc.coll.avg`

```
RETURN coll.avg([1, 2, 3, 4]) AS result
// Returns: 2.5

RETURN coll.avg([]) AS result
// Returns: null

RETURN coll.avg([1, 2, null]) AS result
// Returns: 1.5
```

21.2.13. coll.distinct()

Returns a new list with duplicate values removed, preserving the order of first occurrence.

Syntax: `coll.distinct(list)`

Parameters:

Parameter	Type	Description
list	List	The input list

Returns: List - Deduplicated list, or null if input is null

APOC Compatible: `apoc.coll.distinct`

```
RETURN coll.distinct([1, 2, 2, 3, 1]) AS result
// Returns: [1, 2, 3]
```

21.2.14. coll.flatten()

Flattens nested lists into a single list, with configurable depth control.

Syntax: `coll.flatten(list [, depth])`

Parameters:

Parameter	Type	Description
list	List	The nested list to flatten
depth	Integer	(Optional) Levels of nesting to flatten. Default: 1. Use -1 for unlimited depth.

Returns: List - Flattened list, or null if input is null

APOC Compatible: `apoc.coll.flatten`

```
RETURN coll.flatten([[1, 2], [3, 4]]) AS result
// Returns: [1, 2, 3, 4]

RETURN coll.flatten([[1, [2, 3]], [4]], -1) AS result
// Returns: [1, 2, 3, 4]

RETURN coll.flatten([[1, [2, 3]], [4]], 1) AS result
// Returns: [1, [2, 3], 4]
```

21.2.15. coll.indexOf()

Returns the index of the first occurrence of a value in a list, or -1 if not found.

Syntax: `coll.indexOf(list, value)`

Parameters:

Parameter	Type	Description
list	List	The list to search
value	Any	The value to find

Returns: Integer - Zero-based index, -1 if not found, null if list or value is null

APOC Compatible: `apoc.coll.indexOf`

```
RETURN coll.indexOf(["a", "b", "c"], "b") AS result
// Returns: 1

RETURN coll.indexOf([1, 2, 3], 5) AS result
// Returns: -1
```

21.2.16. coll.insert()

Returns a new list with a value inserted at the given index.

Syntax: `coll.insert(list, index, value)`

Parameters:

Parameter	Type	Description
list	List	The original list
index	Integer	Position to insert at (0-based)
value	Any	Value to insert

Returns: List - New list with value inserted, or null if input is null

APOC Compatible: `apoc.coll.insert`

```
RETURN coll.insert([1, 2, 3], 1, 99) AS result
// Returns: [1, 99, 2, 3]
```

21.2.17. coll.max()

Returns the maximum value in a list using Cypher comparison semantics.

Syntax: `coll.max(list)`

Parameters:

Parameter	Type	Description
list	List	The input list

Returns: Any - Maximum value, or null if list is null or empty

APOC Compatible: `apoc.coll.max`

```
RETURN coll.max([3, 1, 4, 1, 5]) AS result
// Returns: 5
```

21.2.18. coll.min()

Returns the minimum value in a list using Cypher comparison semantics.

Syntax: `coll.min(list)`

Parameters:

Parameter	Type	Description
list	List	The input list

Returns: Any - Minimum value, or null if list is null or empty

APOC Compatible: `apoc.coll.min`

```
RETURN coll.min([3, 1, 4, 1, 5]) AS result
// Returns: 1
```

21.2.19. coll.pairsMin()

Returns the consecutive-element pairs of a list, dropping the trailing incomplete pair.

Syntax: `coll.pairsMin(list)`

Parameters:

Parameter	Type	Description
list	List	The input list

Returns: List - A list of two-element lists, empty for a list of fewer than two elements, or null if input is null

Unlike `apoc.coll.pairs`, which pads the last pair with `null`, this function drops it. A list of `n` elements therefore yields `n - 1` pairs.

APOC Compatible: `apoc.coll.pairsMin`

```
RETURN coll.pairsMin([1, 2, 3]) AS result
// Returns: [[1, 2], [2, 3]]
```

```
RETURN coll.pairsMin([1]) AS result
// Returns: []
```

Elements may be of any type, including lists, which are paired as values rather than flattened:

```
RETURN coll.pairsMin([[1, 2], [3, 4]]) AS result
// Returns: [[1, 2], [3, 4]]
```

21.2.20. coll.remove()

Returns a new list with element(s) removed starting at the given index.

Syntax: `coll.remove(list, index [, count])`

Parameters:

Parameter	Type	Description
list	List	The original list
index	Integer	Starting position (0-based)
count	Integer	(Optional) Number of elements to remove. Default: 1

Returns: List - New list with elements removed, or null if input is null

APOC Compatible: `apoc.coll.remove`

```
RETURN coll.remove([1, 2, 3, 4, 5], 1) AS result
// Returns: [1, 3, 4, 5]

RETURN coll.remove([1, 2, 3, 4, 5], 1, 2) AS result
// Returns: [1, 4, 5]
```

21.2.21. coll.sort()

Returns a sorted copy of the list using Cypher comparison semantics.

Syntax: `coll.sort(list)`

Parameters:

Parameter	Type	Description
list	List	The input list

Returns: List - Sorted list, or null if input is null

APOC Compatible: `apoc.coll.sort`

```
RETURN coll.sort([3, 1, 4, 1, 5]) AS result
// Returns: [1, 1, 3, 4, 5]
```

21.2.22. coll.sum()

Returns the sum of a list of numbers.

Syntax: `coll.sum(list)`

Parameters:

Parameter	Type	Description
list	List	A list of numbers

Returns: Double - The sum, `0.0` for an empty list, or null if input is null

Null elements are skipped. A non-numeric element raises an error.

APOC Compatible: `apoc.coll.sum`

```
RETURN coll.sum([1, 2, 3, 4]) AS result
// Returns: 10.0

RETURN coll.sum([]) AS result
// Returns: 0.0

RETURN coll.sum([1, 2, 'x']) AS result
// Error: Type mismatch: coll.sum() expects an INTEGER or a FLOAT argument but got
STRING
```

21.2.23. coll.toSet()

Returns a copy of the list with duplicates removed, preserving the order of first occurrence.

Syntax: `coll.toSet(list)`

Parameters:

Parameter	Type	Description
list	List	The input list

Returns: List - Unique list, or null if input is null

Deduplication uses object equality, so values of different numeric types are kept separately: `coll.toSet([1, 1.0])` returns both elements. This matches `coll.distinct()`, which applies the same rule.

APOC Compatible: `apoc.coll.toSet`

```
RETURN coll.toSet([1, 2, 2, 3]) AS result
// Returns: [1, 2, 3]

RETURN coll.toSet([1, null, 1, null]) AS result
// Returns: [1, null]
```

21.2.24. coll.union()

Returns the distinct union of two lists, preserving the order of first occurrence.

Syntax: `coll.union(list1, list2)`

Parameters:

Parameter	Type	Description
list1	List	The first list
list2	List	The second list

Returns: List - Union without duplicates. A null argument contributes no elements, so `coll.union(null, [1])` returns `[1]`.

As with `coll.toSet()`, deduplication is by object equality: `coll.union([1], [1.0])` keeps both values.

APOC Compatible: `apoc.coll.union`

```
RETURN coll.union([1, 2, 3], [2, 3, 4]) AS result
// Returns: [1, 2, 3, 4]
```

21.2.25. coll.unionAll()

Returns the concatenation of two lists, preserving duplicates.

Syntax: `coll.unionAll(list1, list2)`

Parameters:

Parameter	Type	Description
list1	List	The first list
list2	List	The second list

Returns: List - All elements of both lists in order. A null argument contributes no elements.

APOC Compatible: `apoc.coll.unionAll`

```
RETURN coll.unionAll([1, 2, 3], [2, 3, 4]) AS result
// Returns: [1, 2, 3, 2, 3, 4]
```

21.2.26. Map Functions

21.2.27. map.clean()

Remove null values and empty strings from a map.

Syntax: `map.clean(map, [keys], [values])`

Returns: Map - Cleaned map

APOC Compatible: `apoc.map.clean`

21.2.28. map.flatten()

Flatten a nested map using a delimiter.

Syntax: `map.flatten(map, delimiter)`

Returns: Map - Flattened map

APOC Compatible: `apoc.map.flatten`

```
RETURN map.flatten({a: {b: 1, c: 2}}, ".") AS result
// Returns: {"a.b": 1, "a.c": 2}
```

21.2.29. map.fromLists()

Create a map from a list of keys and a list of values.

Syntax: `map.fromLists(keys, values)`

Returns: Map - New map

APOC Compatible: `apoc.map.fromLists`

```
RETURN map.fromLists(["a", "b", "c"], [1, 2, 3]) AS result
// Returns: {a: 1, b: 2, c: 3}
```

21.2.30. map.fromPairs()

Create a map from a list of key-value pairs.

Syntax: `map.fromPairs(pairs)`

Returns: Map - New map

APOC Compatible: `apoc.map.fromPairs`

21.2.31. map.groupBy()

Group a list of maps by a key.

Syntax: `map.groupBy(list, key)`

Returns: Map - Map of groups

APOC Compatible: `apoc.map.groupBy`

```
RETURN map.groupBy([{type: "A", val: 1}, {type: "B", val: 2}, {type: "A", val: 3}],
"type") AS result
// Returns: {A: [{type: "A", val: 1}, {type: "A", val: 3}], B: [{type: "B", val: 2}]}
```

21.2.32. map.merge()

Merge two maps, with the second map's values overriding the first.

Syntax: `map.merge(map1, map2)`

Returns: Map - Merged map

APOC Compatible: `apoc.map.merge`

```
RETURN map.merge({a: 1, b: 2}, {b: 3, c: 4}) AS result
// Returns: {a: 1, b: 3, c: 4}
```

21.2.33. map.mergeList()

Merge a list of maps.

Syntax: `map.mergeList(listOfMaps)`

Returns: Map - Merged map

APOC Compatible: `apoc.map.mergeList`

21.2.34. map.removeKey()

Remove a key from a map.

Syntax: `map.removeKey(map, key)`

Returns: Map - Map with key removed

APOC Compatible: `apoc.map.removeKey`

21.2.35. map.removeKeys()

Remove multiple keys from a map.

Syntax: `map.removeKeys(map, keys)`

Returns: Map - Map with keys removed

APOC Compatible: `apoc.map.removeKeys`

21.2.36. map.setKey()

Add or update a key in a map.

Syntax: `map.setKey(map, key, value)`

Returns: Map - Map with key added/updated

APOC Compatible: `apoc.map.setKey`

```
RETURN map.setKey({a: 1}, "b", 2) AS result
// Returns: {a: 1, b: 2}
```

21.2.37. map.sortedProperties()

Get map entries sorted by key.

Syntax: `map.sortedProperties(map)`

Returns: List - Sorted list of [key, value] pairs

APOC Compatible: `apoc.map.sortedProperties`

21.2.38. map.submap()

Extract a subset of keys from a map.

Syntax: `map.submap(map, keys, [defaults])`

Returns: Map - Submap with specified keys

APOC Compatible: `apoc.map.submap`

21.2.39. map.unflatten()

Unflatten a map with delimited keys into a nested map.

Syntax: `map.unflatten(map, delimiter)`

Returns: Map - Nested map

APOC Compatible: `apoc.map.unflatten`

21.2.40. map.values()

Get all values from a map.

Syntax: `map.values(map, [keys])`

Returns: List - List of values

APOC Compatible: `apoc.map.values`

Convert & Create Functions

21.2.41. Convert Functions

21.2.42. convert.fromJsonList()

Parse a JSON string to a list.

Syntax: `convert.fromJsonList(json)`

Returns: List - Parsed list

APOC Compatible: `apoc.convert.fromJsonList`

21.2.43. convert.fromJsonMap()

Parse a JSON string to a map.

Syntax: `convert.fromJsonMap(json)`

Returns: Map - Parsed map

APOC Compatible: `apoc.convert.fromJsonMap`

```
RETURN convert.fromJsonMap('{"name":"John","age":30}') AS result
// Returns: {name: "John", age: 30}
```

21.2.44. convert.toBoolean()

Convert a value to boolean.

Syntax: `convert.toBoolean(value)`

Returns: Boolean - Converted value

APOC Compatible: `apoc.convert.toBoolean`

```
RETURN convert.toBoolean("true") AS result
// Returns: true

RETURN convert.toBoolean(1) AS result
// Returns: true

RETURN convert.toBoolean(0) AS result
// Returns: false
```

21.2.45. convert.toFloat()

Convert a value to float.

Syntax: `convert.toFloat(value)`

Returns: Float - Converted value

APOC Compatible: `apoc.convert.toFloat`

21.2.46. convert.toInteger()

Convert a value to integer.

Syntax: `convert.toInteger(value)`

Returns: Integer - Converted value

APOC Compatible: `apoc.convert.toInteger`

21.2.47. convert.toJson()

Convert a value to JSON string.

Syntax: `convert.toJson(value)`

Returns: String - JSON representation

APOC Compatible: `apoc.convert.toJson`

```
RETURN convert.toJson({name: "John", age: 30}) AS result
// Returns: '{"name":"John","age":30}'

RETURN convert.toJson([1, 2, 3]) AS result
// Returns: '[1,2,3]'
```

21.2.48. convert.toList()

Convert a value to a list.

Syntax: `convert.toList(value)`

Returns: List - Converted list

APOC Compatible: `apoc.convert.toList`

21.2.49. convert.toMap()

Convert a value to a map.

Syntax: `convert.toMap(value)`

Returns: Map - Converted map

APOC Compatible: `apoc.convert.toMap`

21.2.50. convert.toSet()

Convert a value to a set (unique values).

Syntax: `convert.toSet(value)`

Returns: List - Set of unique values

APOC Compatible: `apoc.convert.toSet`

21.2.51. convert.toString()

Convert a value to a string. This is the namespaced entry point for the built-in `toString()` function and has identical semantics.

Syntax: `convert.toString(value)`

Returns: String - String representation, or null if the value is null

Strings, numbers, booleans and temporal values convert directly. *(Since v26.12.1.)* A time or a datetime is written as `toString()` writes it, as Neo4j 5 does: `convert.toString(localtime('10:00'))` is `10:00:00`, and the fraction of a second has no trailing zeros. Lists, maps, nodes, relationships and paths are rejected with a `TypeError: InvalidArgumentValue` error rather than being stringified.

APOC Compatible: `apoc.convert.toString`

```
RETURN convert.toString(42) AS result
// Returns: "42"

RETURN convert.toString([1, 2]) AS result
// Error: TypeError: InvalidArgumentValue - toString() cannot convert ArrayList
```

21.2.52. Create Functions

21.2.53. create.uuid()

Generate a UUID.

Syntax: `create.uuid()`

Returns: String - UUID string

APOC Compatible: `apoc.create.uuid`

21.2.54. create.uuidBase64()

Generate a UUID in Base64 format.

Syntax: `create.uuidBase64()`

Returns: String - Base64-encoded UUID

APOC Compatible: `apoc.create.uuidBase64`

21.2.55. create.vNode()

Create a virtual node (not persisted).

Syntax: `create.vNode(labels, [properties])` or `CALL create.vNode(labels, [properties]) YIELD node`

Returns: Node - Virtual node

The virtual node is a node like any other for the query: `labels()`, `properties()`, `keys()`, `n.prop` and label predicates work on it. It is never stored, so it cannot be changed, deleted or connected with `CREATE`. Its `id()` is negative and its `elementId()` is that number as text, as in APOC, and two virtual nodes are never equal.

APOC Compatible: `apoc.create.vNode`

```
RETURN labels(create.vNode(['Person'], {name: 'Alice'})) AS labels
```

21.2.56. create.vRelationship()

Create a virtual relationship (not persisted).

Syntax: `create.vRelationship(startNode, type, properties, endNode)` or `CALL create.vRelationship(startNode, type, properties, endNode) YIELD rel`

Returns: Relationship - Virtual relationship

The start and end nodes can be stored or virtual nodes. `type()`, `startNode()`, `endNode()`, `properties()` and `r.prop` work on the result. A null start node, end node or type is an error.

APOC Compatible: `apoc.create.vRelationship`

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
RETURN create.vRelationship(a, 'SIMILAR', {score: 0.9}, b) AS rel
```

21.3. Date & Math Functions

21.3.1. Date Functions (Extended)

21.3.2. date.add()

Add time to a timestamp.

Syntax: `date.add(timestamp, value, unit)`

Parameters:

Parameter	Type	Description
timestamp	Long	The timestamp (in milliseconds)
value	Long	Amount to add (can be negative)
unit	String	Unit of the value (ms, s, m, h, d)

Returns: Long - New timestamp

APOC Compatible: `apoc.date.add`

```
RETURN date.add(date.currentTimestamp(), 1, "d") AS tomorrow
// Returns: timestamp for tomorrow

RETURN date.add(date.currentTimestamp(), -1, "h") AS oneHourAgo
// Returns: timestamp for 1 hour ago
```

21.3.3. date.convert()

Convert a timestamp between units.

Syntax: `date.convert(timestamp, fromUnit, toUnit)`

Returns: Long - Converted timestamp

APOC Compatible: `apoc.date.convert`

21.3.4. date.currentTimestamp()

Get the current timestamp in milliseconds.

Syntax: `date.currentTimestamp()`

Returns: Long - Current timestamp in milliseconds since epoch

APOC Compatible: `apoc.date.currentTimestamp`

```
RETURN date.currentTimestamp() AS now
// Returns: 1705314600000 (example)
```

21.3.5. date.field()

Extract a specific field from a timestamp.

Syntax: `date.field(timestamp, field)`

Returns: Long - Field value

APOC Compatible: `apoc.date.field`

21.3.6. date.fields()

Extract all fields from a timestamp as a map.

Syntax: `date.fields(timestamp)`

Returns: Map - Map of date fields

APOC Compatible: `apoc.date.fields`

21.3.7. date.format()

Format a timestamp to a date string.

Syntax: `date.format(timestamp, unit, format)`

Parameters:

Parameter	Type	Description
timestamp	Long	The timestamp value
unit	String	Unit of the timestamp (ms, s, m, h, d)
format	String	Date format pattern (Java SimpleDateFormat)

Returns: String - Formatted date string

APOC Compatible: `apoc.date.format`

```
RETURN date.format(1705314600000, "ms", "yyyy-MM-dd HH:mm:ss") AS result
// Returns: "2024-01-15 10:30:00"
```

21.3.8. date.fromISO8601()

Parse an ISO 8601 string to timestamp.

Syntax: `date.fromISO8601(dateString)`

Returns: Long - Timestamp in milliseconds

APOC Compatible: `apoc.date.fromISO8601`

```
RETURN date.fromISO8601("2024-01-15T10:30:00Z") AS result
// Returns: 1705314600000
```

21.3.9. date.parse()

Parse a date string to timestamp.

Syntax: `date.parse(dateString, format)`

Returns: Long - Timestamp in milliseconds

APOC Compatible: `apoc.date.parse`

21.3.10. date.systemTimezone()

Get the system's default timezone.

Syntax: `date.systemTimezone()`

Returns: String - Timezone ID (e.g., "America/New_York")

APOC Compatible: `apoc.date.systemTimezone`

21.3.11. date.toISO8601()

Convert a timestamp to ISO 8601 format.

Syntax: `date.toISO8601(timestamp)`

Returns: String - ISO 8601 formatted string

APOC Compatible: `apoc.date.toISO8601`

```
RETURN date.toISO8601(1705314600000) AS result
// Returns: "2024-01-15T10:30:00Z"
```

21.3.12. Math Functions (Extended)

21.3.13. math.cosh()

Calculate the hyperbolic cosine.

Syntax: `math.cosh(x)`

Returns: Double - Hyperbolic cosine value

APOC Compatible: `apoc.math.cosh`

21.3.14. math.maxDouble()

Get the maximum double value.

Syntax: `math.maxDouble()`

Returns: Double - Maximum double value

APOC Compatible: `apoc.math.maxDouble`

21.3.15. math.maxLong()

Get the maximum long value.

Syntax: `math.maxLong()`

Returns: Long - Maximum long value (9223372036854775807)

APOC Compatible: `apoc.math.maxLong`

21.3.16. math.minLong()

Get the minimum long value.

Syntax: `math.minLong()`

Returns: Long - Minimum long value

APOC Compatible: `apoc.math.minLong`

21.3.17. math.round()

Round a number to the nearest integer, or to a given number of decimal places using a chosen rounding mode. This is the namespaced entry point for the built-in `round()` function and has identical semantics.

Syntax: `math.round(value [, precision [, mode]])`

Parameters:

Parameter	Type	Description
value	Number	The number to round
precision	Integer	(Optional) Number of decimal places to keep
mode	String	(Optional) Rounding mode: `UP`, `DOWN`, `CEILING`, `FLOOR`, `HALF_UP`, `HALF_DOWN` or `HALF_EVEN`. Default: `HALF_UP`

Returns: Double - The rounded value, or null if any argument is null

An unknown rounding mode raises an error rather than falling back to the default. `NaN` and infinity are returned unchanged.

APOC Compatible: `apoc.math.round`

```
RETURN math.round(2.5) AS result
// Returns: 3.0

RETURN math.round(3.456, 1) AS result
// Returns: 3.5

RETURN math.round(3.456, 2, 'FLOOR') AS result
// Returns: 3.45
```

21.3.18. math.sigmoid()

Calculate the sigmoid function.

Syntax: `math.sigmoid(x)`

Returns: Double - Sigmoid value (0-1)

APOC Compatible: `apoc.math.sigmoid`

```
RETURN math.sigmoid(0) AS result
// Returns: 0.5

RETURN math.sigmoid(10) AS result
// Returns: ~0.9999
```

21.3.19. math.sigmoidPrime()

Calculate the derivative of the sigmoid function.

Syntax: `math.sigmoidPrime(x)`

Returns: Double - Sigmoid derivative

APOC Compatible: `apoc.math.sigmoidPrime`

21.3.20. math.sinh()

Calculate the hyperbolic sine.

Syntax: `math.sinh(x)`

Returns: Double - Hyperbolic sine value

APOC Compatible: `apoc.math.sinh`

21.3.21. math.tanh()

Calculate the hyperbolic tangent.

Syntax: `math.tanh(x)`

Returns: Double - Hyperbolic tangent value

APOC Compatible: `apoc.math.tanh`

21.3.22. Number Functions

21.3.23. number.format()

Format a number as a string using a `DecimalFormat` pattern.

Syntax: `number.format(number [, pattern])`

Parameters:

Parameter	Type	Description
number	Number	The number to format
pattern	String	(Optional) A `DecimalFormat` pattern. Default: `, 0.`

Returns: String - The formatted number, or null if the number or an explicitly passed pattern is null

Formatting always uses the root locale, so the grouping and decimal separators do not depend on the server's locale.

APOC Compatible: `apoc.number.format`

```
RETURN number.format(1234567.891) AS result
// Returns: "1,234,567.891"

RETURN number.format(1000000, '#,###') AS result
// Returns: "1,000,000"

RETURN number.format(0.4567, '#.##%') AS result
// Returns: "45.67%"
```

Graph Element Functions

21.3.24. Node Functions

The `relTypes` argument of `node.degree()`, `node.degree.in()`, `node.degree.out()`, `node.relationship.exists()` and `node.relationship.types()` is APOC's relationship pattern, the same syntax as the relationship filter of the `path.*` procedures: `TYPE` matches the type in both directions, `TYPE>` outgoing only, `<TYPE` incoming only, `>` and `<` every type in one direction, and `|` separates the alternatives (`'KNOWS>|<LIKES'`). A null or blank pattern matches every relationship. A count sums per alternative, as APOC does. The optional `direction` argument (`IN`, `OUT` or `BOTH`) narrows the pattern: an alternative without a marker is followed in that direction, and one marked the other way matches nothing. `node.degree.in()` and `node.degree.out()` read the pattern with their direction fixed, so `node.degree.in(n, 'KNOWS>')` is `0`. *Since v26.12.1*; earlier versions read the whole string as

one type name, so `node.degree(n, 'KNOWS>')` answered `0`.

21.3.25. node.degree()

Get the total degree (in + out) of a node.

Syntax: `node.degree(node, [relTypes], [direction])`

Returns: Long - Total degree

APOC Compatible: `apoc.node.degree`

21.3.26. node.degree.in()

Get the incoming degree of a node.

Syntax: `node.degree.in(node, [relTypes])`

Returns: Long - Incoming degree

APOC Compatible: `apoc.node.degree.in`

21.3.27. node.degree.out()

Get the outgoing degree of a node.

Syntax: `node.degree.out(node, [relTypes])`

Returns: Long - Outgoing degree

APOC Compatible: `apoc.node.degree.out`

21.3.28. node.id()

Get the internal ID of a node.

Syntax: `node.id(node)`

Returns: String - Node ID (RID)

APOC Compatible: `apoc.node.id`

21.3.29. node.labels()

Get all labels of a node.

Syntax: `node.labels(node)`

Returns: List - List of labels

APOC Compatible: `apoc.node.labels`

21.3.30. node.relationship.exists()

Check if a relationship exists from a node.

Syntax: `node.relationship.exists(node, [relTypes], [direction])`

Returns: Boolean - true if relationship exists

APOC Compatible: `apoc.node.relationship.exists`

21.3.31. node.relationship.types()

Get the relationship types connected to a node. With a pattern, the answer is the types the pattern names that the node has a relationship of, in the pattern's order.

Syntax: `node.relationship.types(node, [relTypes], [direction])`

Returns: List - List of relationship types

Since v26.12.1 the second argument is the relationship pattern, as in APOC, and the direction is the third one: `node.relationship.types(n, null, 'IN')` (or `(n, '<')`) lists the incoming types. A call written for the old signature keeps its answer: a second argument that is a direction word (`IN`, `INCOMING`, `OUT`, `OUTGOING`, `BOTH`, in any case) with no third argument is still read as the direction, unless the schema has an edge type with exactly that name.

APOC Compatible: `apoc.node.relationship.types`

21.3.32. Relationship Functions

21.3.33. rel.endNode()

Get the end node of a relationship.

Syntax: `rel.endNode(relationship)`

Returns: Node - End node

APOC Compatible: `apoc.rel.endNode`

21.3.34. rel.id()

Get the internal ID of a relationship.

Syntax: `rel.id(relationship)`

Returns: String - Relationship ID (RID)

APOC Compatible: `apoc.rel.id`

21.3.35. rel.startNode()

Get the start node of a relationship.

Syntax: `rel.startNode(relationship)`

Returns: Node - Start node

APOC Compatible: `apoc.rel.startNode`

21.3.36. rel.type()

Get the type of a relationship.

Syntax: `rel.type(relationship)`

Returns: String - Relationship type

APOC Compatible: `apoc.rel.type`

Path & Algorithm Procedures

21.3.37. Algorithm Procedures

21.3.38. algo.allsimplepaths()

Find all simple paths (without repeated nodes) between two nodes.

Syntax: `CALL algo.allsimplepaths(startNode, endNode, relTypes, maxDepth) YIELD path`

Parameters:

Parameter	Type	Description
startNode	Node	Starting node

Parameter	Type	Description
endNode	Node	Target node
relTypes	String/List	Relationship types and directions in APOC's syntax (e.g., "KNOWS>|<WORKS_WITH"), see the relationship pattern
maxDepth	Integer	Maximum path length

Returns: `path` - Each simple path found

APOC Compatible: `apoc.algo.allSimplePaths`

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CALL algo.allsimplepaths(a, b, 'KNOWS', 5) YIELD path
RETURN path
```

See reference/graph-algorithms/path-finding.pdf in the Graph Algorithms appendix for full parameter reference and examples.

21.3.39. algo.astar()

Find the shortest path using A* algorithm with optional geographic heuristics.

Syntax: `CALL algo.astar(startNode, endNode, relType, weightProperty, [latProperty], [lonProperty]) YIELD path, weight`

Parameters:

Parameter	Type	Description
startNode	Node	Starting node
endNode	Node	Target node
relType	String	Relationship types and directions in APOC's syntax, see the relationship pattern
weightProperty	String	Edge property to use as weight
latProperty	String	Optional - Node property for latitude (for geographic heuristic)
lonProperty	String	Optional - Node property for longitude (for geographic heuristic)

Returns:

- `path` - The shortest path
- `weight` - Total path weight

APOC Compatible: `apoc.algo.aStar`

```
MATCH (a:City {name: 'Seattle'}), (b:City {name: 'Miami'})
CALL algo.astar(a, b, 'FLIGHT', 'distance', 'lat', 'lon') YIELD path, weight
RETURN path, weight
```

See reference/graph-algorithms/path-finding.pdf in the Graph Algorithms appendix for full parameter reference and examples.

21.3.40. algo.dijkstra()

Find the shortest weighted path between two nodes using Dijkstra's algorithm.

Syntax: `CALL algo.dijkstra(startNode, endNode, relType, weightProperty, [defaultWeight]) YIELD path, weight`

Parameters:

Parameter	Type	Description
startNode	Node	Starting node
endNode	Node	Target node
relType	String	Relationship types and directions in APOC's syntax (e.g., "ROAD>"), see the relationship pattern
weightProperty	String	Edge property to use as weight
defaultWeight	Number/String	Optional - Weight of an edge without a numeric weight (default: such an edge weighs 1). A string is read as the traversal direction ("OUT", "IN", "BOTH", default: "BOTH") of the types without a direction marker

Returns:

- `path` - The shortest path
- `weight` - Total path weight

APOC Compatible: `apoc.algo.dijkstra`

```
MATCH (a:City {name: 'New York'}), (b:City {name: 'Los Angeles'})
CALL algo.dijkstra(a, b, 'ROAD', 'distance') YIELD path, weight
RETURN path, weight
```

See reference/graph-algorithms/path-finding.pdf in the Graph Algorithms appendix for full parameter reference and examples.

21.3.41. Path Functions (Extended)

The `path.` functions take any Cypher path: one bound by a MATCH pattern or `shortestPath()`, one a procedure yields (see `reference/graph-algorithms/path-finding.pdf`), or one another `path.` function returned. A value that is not a path is an error.

21.3.42. path.combine()

Combine two paths into one: the second path continues from where the first ends. The end node of `path1` must be the start node of `path2`, otherwise it is an error; a `null` path returns the other one. `path.combine([p1, p2, …])` combines a list of paths in order.

Syntax: `path.combine(path1, path2)`

Returns: Path - Combined path

```
MATCH p1 = (a:N {id: 1})-[:R]->(b:N {id: 2}), p2 = (b)-[:R]->(c:N {id: 3})
RETURN path.combine(p1, p2) AS combined
```

Since v26.10.2. Earlier versions took a single argument (a list of paths) and refused the documented two-argument form.

APOC Compatible: `apoc.path.combine`

21.3.43. path.create()

Create a path from a start node and the relationships that continue it. Each relationship must touch the node the path has reached so far, in either direction, and the path continues to its other node; a `null` relationship is skipped.

Syntax: `path.create(startNode, relationships)`

Returns: Path - Created path

Since v26.10.2 the nodes of the path are the ones the relationships reach, and a relationship that does not continue the path is an error. Earlier versions returned a path holding the start node alone beside all the relationships.

APOC Compatible: `apoc.path.create`

21.3.44. path.elements()

Get all elements (nodes and relationships) of a path, in path order: `[node, relationship, node, …]`.

Syntax: `path.elements(path)`

Returns: List - List of nodes and relationships

APOC Compatible: `apoc.path.elements`

21.3.45. path.slice()

Get a slice of a path: `length` relationships starting at relationship `offset` (0-based), and the nodes they join. `offset` defaults to 0 and `length` to -1, which reads to the end of the path. Any other negative length is an empty slice, and an offset past the end is an empty path at the last node.

Syntax: `path.slice(path, [offset], [length])`

Returns: Path - Sliced path

APOC Compatible: `apoc.path.slice`

21.3.46. Path Expansion Procedures

21.3.47. Relationship and label filters

All the path expansion procedures read `relationshipFilter` and `labelFilter` in APOC's syntax.

Relationship filter: `[<]TYPE1[>]|[<]TYPE2[>]|⋯`

Entry	Follows
`TYPE`	`TYPE` in both directions
`TYPE>`	`TYPE` outgoing only
`<TYPE`	`TYPE` incoming only
`>`	every relationship type, outgoing only
`<`	every relationship type, incoming only

`|` separates the alternatives of one hop, each with its own direction (`'ROAD>|<RAIL'`). A comma separates the hops of a repeating sequence: `'NEXT>,<FOLLOWS'` follows `NEXT` outgoing on the first hop, `FOLLOWS` incoming on the second, `NEXT` outgoing on the third, and so on. A null or empty filter follows every relationship in both directions.

Label filter: `[-/>]LABEL1|[-/>]LABEL2|⋯`

Entry	Meaning
`-Label`	Denylist: no node of the path can carry the label
`/Label`	Termination: only paths ending at a node with the label are returned, and the expansion stops there

Entry	Meaning
`>Label`	End node: only paths ending at a node with the label are returned, and the expansion continues past it if the allowlist lets it
`+Label` or `Label`	Allowlist: every node of the path must carry one of the labels
`*`	Any label
`A:B`	A node carrying both labels

Each entry carries its own operator, and the operators apply in the order denylist, termination, end node, allowlist, whatever order they are written in. A comma separates the filters of a repeating sequence, one per depth. A node carries a label when its type is, or extends, that vertex type, so a node with several labels matches each of them.

The filters do not apply to the start node unless `filterStartNode: true` is set. When the label filter names termination or end nodes, the zero-length path at the start node is not returned unless it qualifies.

Node lists: `endNodes`, `terminatorNodes`, `allowlistNodes` and `denylistNodes` are the operators `>`, `/`, `+` and `-` of the label filter, naming nodes instead of labels. Each takes a list of nodes (a single node is a list of one; `whitelistNodes` and `blacklistNodes` are the older names of the last two):

Option	Meaning
`denylistNodes`	No node of the path can be one of them
`terminatorNodes`	Only paths ending at one of them are returned, and the expansion stops there
`endNodes`	Only paths ending at one of them are returned, and the expansion continues past it
`allowlistNodes`	Every node of the path must be one of them; the end and terminator nodes are allowed too

The lists combine with the label filter and with each other: a node is an answer, and the expansion goes on past it, only if every filter says so. A terminator node closer to the start than `minLevel` does not stop the expansion. The start node is left out of the denylist and the allowlist, and cannot be an end or terminator node, unless `filterStartNode: true` is set.

```
MATCH (a:City {name: 'A'}), (c:City {name: 'C'}), (p:City {name: 'P'})
CALL path.expandconfig(a, {relationshipFilter: 'ROAD>', endNodes: [c, p]}) YIELD path
RETURN path
```

Additional configuration options (`path.expandconfig`, `path.spanningtree`, `path.subgraphall`, `path.subgraphnodes`):

Option	Default	Description
sequence		Label and relationship filters alternating, one pair per step of a repeating sequence, for example `'Post, NEXT>, Reply, <FROM'`. It takes precedence over `relationshipFilter` and `labelFilter`
filterStartNode	`false`	Whether the label filter and the node lists apply to the start node too
beginSequenceAtStart	`true`	Whether the first label filter of a sequence applies to the start node. When `false`, the sequence starts one hop away, and a `sequence` starts with a relationship filter
endNodes, terminatorNodes, allowlistNodes, denylistNodes		Node lists, see above
uniqueness	`RELATIONSHIP_PATH`	What the walk may visit more than once, see uniqueness. Only `path.expandconfig` honours it
optional	`false`	When the walk finds nothing, answer one row with a null `path` (or `node`) instead of none. Not read by `path.subgraphall`, whose answer is two lists
limit	no limit	The most paths (nodes, for `path.subgraphnodes` and `path.subgraphall`) to return. A negative value is no limit
maxLevel	no limit	The longest path. A negative value is no limit

Uniqueness. The `uniqueness` option of `path.expandconfig` takes APOC's nine values, case-insensitively:

Value	A walk may visit
`RELATIONSHIP_PATH` (default)	a node many times, but a relationship once per path: every trail is a path
`NODE_PATH`	a node once per path: every simple path is a path
`NODE_GLOBAL`	a node once in the whole walk: each node is reached once, along the first route to it (the shortest one when `bfs` is `true`)
`NODE_LEVEL`	a node once at each distance from the start
`NODE_RECENT`	a node again only after 10000 other nodes were visited
`RELATIONSHIP_GLOBAL`	a relationship once in the whole walk
`RELATIONSHIP_LEVEL`	a relationship once at each distance from the start
`RELATIONSHIP_RECENT`	a relationship again only after 10000 others were walked
`NONE`	anything

Whatever the value, the expansion never leaves a node by the relationship it arrived by. A value that is none of these is refused (`uniqueness 'X' is not one of NODE_GLOBAL, ⋯`); APOC quietly uses

its default instead. The values scoped to a level or to the walk keep the first route that reaches a node or a relationship, so which of several equal routes they report depends on the order the expansion meets them in. `path.spanningtree`, `path.subgraphnodes` and `path.subgraphall` are always `NODE_GLOBAL`, as in APOC: the option is read and checked there, and does not change the answer.

The trails of a graph with cycles or with several routes to a node are exponential in the depth, and the default `uniqueness` returns all of them. Bound the walk with `maxLevel` and `limit`, or choose `NODE_GLOBAL` or `NODE_PATH`. The walk produces its paths as the query reads them, so a `limit` (or a `LIMIT` after the call) ends it early; the command timeout ends it too, and the branches it holds are charged to the query's heap (`arcadedb.queryMaxHeapElementsAllowedPerOp`, `arcadedb.queryMaxHeapRAM`), so a walk too wide for them fails the query instead of the server. That includes a breadth-first walk of a very wide component, `path.spanningtree` included: a level with more branches than `arcadedb.queryMaxHeapElementsAllowedPerOp` (auto-scaled with the heap, never below 500000) fails with a heap-limit error that names the setting, which can be raised.

Every relationship type and label named in a filter must exist in the schema, and a name of the wrong kind is refused: `relationshipFilter: 'ROADS'` raises `relationship type 'ROADS' does not exist` rather than returning nothing. Neo4j answers such a filter with an empty result instead.

Before 26.10.2 the direction markers and the label operators were read as part of the name, so `'ROAD>'` or `'+Person'` matched nothing and the procedures answered an empty result, and a comma in a filter separated alternatives rather than the hops of a sequence. Since 26.10.2 `path.spanningtree`, `path.subgraphall` and `path.subgraphnodes` also honour `minLevel` (0 includes the start node, 1 leaves it out, any other value is refused, as in APOC), and a `config` argument that is not a map is refused instead of being ignored.

Before 26.10.2 the node lists, `uniqueness`, `optional` and the negative `maxLevel` and `limit` were ignored. In `path.expandconfig` the breadth-first walk always reached each node once (what is now `NODE_GLOBAL`) and the depth-first walk, and `path.expand`, always walked each node once per path (`NODE_PATH`). The default is now APOC's, so a call on a graph with a cycle or with several routes to a node returns more paths than before. To keep the old answer, add `uniqueness: 'NODE_GLOBAL'` to a call, and `uniqueness: 'NODE_PATH'` to a call with `bfs: false`. A `path.expand(n, rels, labels, min, max)` is `path.expandconfig(n, {relationshipFilter: rels, labelFilter: labels, minLevel: min, maxLevel: max, bfs: false, uniqueness: 'NODE_PATH'})` as it used to answer.

21.3.48. path.expand()

Expand paths from a starting node following relationship types and node labels.

Syntax: `CALL path.expand(startNode, relationshipFilter, labelFilter, minDepth, maxDepth) YIELD path`

Parameters:

Parameter	Type	Description
startNode	Node	Starting node
relationshipFilter	String	Relationship types and directions, see filters (e.g., "KNOWS>|<WORKS_WITH")
labelFilter	String	Node label filter, see filters (e.g., "+Person")
minDepth	Integer	Minimum path length (a negative value is 0)
maxDepth	Integer	Maximum path length (a negative value is no limit)

Returns: `path` - Each expanded path

The walk is APOC's for this form: breadth-first, over the trails of the graph (`RELATIONSHIP_PATH` uniqueness, see uniqueness), so a path may pass through a node more than once. Use `path.expandconfig` to choose another.

APOC Compatible: `apoc.path.expand`

```
MATCH (a:Person {name: 'Alice'})
CALL path.expand(a, 'KNOWS|WORKS_WITH', 'Person', 1, 3) YIELD path
RETURN path
```

21.3.49. path.expandconfig()

Expand paths using a configuration map for more control.

Syntax: `CALL path.expandconfig(startNode, config) YIELD path`

Configuration options:

Option	Type	Description
relationshipFilter	String/List	Relationship types and directions, see filters
labelFilter	String/List	Node label filter, see filters
sequence, filterStartNode, beginSequenceAtStart		See filters
endNodes, terminatorNodes, allowlistNodes, denylistNodes	List	Node lists, see filters

Option	Type	Description
uniqueness	String	What the walk may visit more than once, see uniqueness (default: `RELATIONSHIP_PATH`)
minLevel	Integer	Minimum depth (default: 0)
maxLevel	Integer	Maximum depth (default: unlimited; a negative value is unlimited)
bfs	Boolean	Use BFS (true) or DFS (false) (default: true)
limit	Integer	Maximum number of paths to return (default: unlimited; a negative value is unlimited)
optional	Boolean	When no path is found, return one row with a null `path` (default: false)

Returns: `path` - Each expanded path

APOC Compatible: `apoc.path.expandConfig`

```
MATCH (a:Person {name: 'Alice'})
CALL path.expandconfig(a, {
  relationshipFilter: 'KNOWS|WORKS_WITH',
  labelFilter: 'Person',
  minLevel: 1,
  maxLevel: 3,
  bfs: true,
  limit: 100
}) YIELD path
RETURN path
```

21.3.50. path.spanningtree()

Get a spanning tree from the start node to all reachable nodes.

Syntax: `CALL path.spanningtree(startNode [, config]) YIELD path`

Returns: `path` - Each path in the spanning tree

APOC Compatible: `apoc.path.spanningTree`

```
MATCH (root:Category {name: 'Root'})
CALL path.spanningtree(root, {relationshipFilter: 'HAS_CHILD', maxLevel: 5}) YIELD
path
RETURN path
```

21.3.51. path.subgraphall()

Get all nodes and relationships reachable from a starting node.

Syntax: `CALL path.subgraphall(startNode [, config]) YIELD nodes, relationships`

Returns:

- `nodes` - List of all reachable nodes
- `relationships` - List of every relationship between those nodes

APOC Compatible: `apoc.path.subgraphAll`

```
MATCH (a:Person {name: 'Alice'})
CALL path.subgraphall(a, {relationshipFilter: 'KNOWS', maxLevel: 2}) YIELD nodes,
relationships
RETURN size(nodes) AS nodeCount, size(relationships) AS relCount
```

Yield only what you need. `YIELD nodes` on its own does not read a single edge record, while `YIELD nodes, relationships` has to load one per relationship in the component. On a large component that is the difference between a walk whose cost follows the number of nodes and one that follows the number of relationships.

`nodes` and `relationships` are returned in breadth-first order from the starting node.

`relationships` is every relationship, of any type, whose two endpoints are both in `nodes`, whatever the `relationshipFilter` that found the nodes, matching `apoc.path.subgraphAll`: with `relationshipFilter: 'R>'` a relationship of type `S` between two of the nodes is in the answer, though the walk did not follow it. A relationship that leads out of the subgraph is not, and neither is one that touches the start node when `minLevel: 1` leaves it out of `nodes`. Before 26.10.2 the answer was the relationships the walk followed. Finding them reads the outgoing adjacency of each returned node (an entry per relationship, by the identity of its far end: the record of a relationship that leads out of the subgraph is never read), so a node with a very large number of relationships of filtered-out types pays for scanning them.

When the record of a relationship is gone but the adjacency lists of its two endpoints still name it (a dangling entry, left behind by a data-integrity anomaly such as an incomplete replication or a partially rolled-back transaction), `path.subgraphnodes` and `path.subgraphall` follow the entry by default, as a fixed-length hop does, while the procedures that read the relationships skip it. `CHECK DATABASE FIX` prunes the dangling entries, after which every procedure agrees. With `arcadedb.graph.ghostEdgeProbe` set to `true`, the entry is skipped and reported in the log by every path procedure alike: `path.subgraphnodes`, `path.subgraphall`, `path.spanningtree` and `path.expandconfig` return the same nodes, and each node of `path.subgraphall` is the start node or an endpoint of one of the `relationships` returned with it. Another, live, relationship to the same node still reaches it. The check asks the bucket whether the record exists, once per node reached, and reads no record. The setting is `false` by default since v26.12.1; in 26.11.1 it was `true`.

21.3.52. path.subgraphnodes()

Get all nodes reachable from a starting node within configured constraints.

Syntax: `CALL path.subgraphnodes(startNode [, config]) YIELD node`

Returns: `node` - Each reachable node

APOC Compatible: `apoc.path.subgraphNodes`

```
MATCH (a:Person {name: 'Alice'})
CALL path.subgraphnodes(a, {relationshipFilter: 'KNOWS', maxLevel: 3}) YIELD node
RETURN node.name
```

Nodes are returned in breadth-first order from the starting node.

Refactor & Control Flow Procedures

These are procedures, not functions: they are invoked with `CALL … YIELD …` and they write to the database. All three are available under their bare name and under the `apoc.` prefix.

21.3.53. Control Flow Procedures

21.3.54. do.when()

Run one of two Cypher sub-queries depending on a condition, and yield each row the sub-query produces.

Syntax: `CALL do.when(condition, ifQuery, elseQuery, params) YIELD value`

Parameters:

Parameter	Type	Description
condition	Boolean	Selects the branch to run. Must be a boolean; a string, number or list is rejected rather than coerced.
ifQuery	String	Cypher run when the condition is true
elseQuery	String	Cypher run when the condition is false. Pass `null` or an empty string for "no else branch", in which case the call yields no rows.
params	Map	Named parameters bound into the sub-query. Pass `{}` or `null` for none.

All four arguments are required.

Returns: `value` - One row per row the sub-query produces, each holding that row as a map of its columns: `RETURN p` yields `{p: node}`, so the node is read as `value.p` (before v26.10.2 a row returning a single node or relationship yielded that entity's properties instead)

The sub-query is routed to a read or a write execution automatically, so both `MATCH … RETURN` and `CREATE`/`SET`/`DELETE` branches work without the caller having to say which. `ifQuery` is type-checked on every call, including calls where the condition is false and the branch never runs.

APOC Compatible: `apoc.do.when`

```
CALL do.when(
  size($names) > 0,
  'RETURN $names[0] AS first',
  'RETURN null AS first',
  {names: $names}
) YIELD value
RETURN value.first AS first
```

A write branch:

```
MATCH (p:Person {name: 'Alice'})
CALL do.when(
  p.score > 100,
  'MATCH (n:Person) WHERE elementId(n) = $id SET n.tier = "gold" RETURN n.tier AS
tier',
  null,
  {id: elementId(p)}
) YIELD value
RETURN value.tier AS tier
```

`ifQuery` and `elseQuery` run with the caller's privileges, exactly as any other command string the engine executes. Never build either argument by concatenating untrusted input — pass values through `params` instead.

21.3.55. Refactor Procedures

21.3.56. refactor.cloneNodesWithRelationships()

Clone each given node together with every relationship touching it, leaving the originals untouched.

Syntax: `CALL refactor.cloneNodesWithRelationships(nodes, config = {}) YIELD input, output, error`

Parameters:

Parameter	Type	Description
nodes	List	The nodes to clone. A node listed twice is cloned once.
config	Map	Configuration. Optional; omitting it, or passing `{}` or `null`, all fall back to the defaults.

Configuration keys:

Key	Type	Description
skipProperties	List	Property names to leave off the clones

Returns: One row per input node:

- `input` - The original node
- `output` - The clone, or null if cloning it failed
- `error` - The failure message, or null on success

Each clone is a new vertex of the same type carrying the same properties. A relationship whose other endpoint is also being cloned reconnects the two clones; a relationship to a node outside the list reconnects the clone to that same original node. This includes the incoming relationships of a `UNIDIRECTIONAL` edge type, which the node itself does not store *(since v26.10.2)*.

Cloning is best-effort per item: a node that fails to clone gets its own row with `output` null and a message in `error`, and an edge that fails to clone is skipped. This covers failures raised at `save()` time: a `UNIQUE` index violation is checked at commit, so it still fails the whole transaction.

APOC Compatible: `apoc.refactor.cloneNodesWithRelationships`

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CALL refactor.cloneNodesWithRelationships([a, b], {}) YIELD input, output, error
RETURN input, output, error

// Excluding properties from the clones:
MATCH (p:Person)
WITH collect(p) AS people
CALL refactor.cloneNodesWithRelationships(people, {skipProperties: ['ssn', 'email']})
YIELD output
RETURN count(output) AS cloned
```

21.3.57. refactor.mergeNodes()

Merge a list of nodes into the first one, rewiring their relationships and deleting the absorbed nodes.

Syntax: `CALL refactor.mergeNodes(nodes, config = {}) YIELD node`

Parameters:

Parameter	Type	Description
nodes	List	The nodes to merge. The first is the survivor; the rest are absorbed. A node listed twice counts once, and at least two distinct nodes are required.
config	Map	Configuration. Optional; omitting it, or passing `{}` or `null`, all fall back to the defaults (the `overwrite` properties policy).

Configuration keys:

Key	Type	Description
properties	String	How to resolve a property present on both the survivor and an absorbed node: `overwrite` (the absorbed value wins; the default), `discard` (the survivor keeps its value) or `combine` (the distinct values across the merged nodes are kept, in first-seen order, as a list - or as that single value itself when every merged node agrees on it, rather than a list of duplicates). Any other value raises an error.

Returns: `node` - The surviving node

A property present only on an absorbed node is always copied onto the survivor, whichever policy is in effect. Every incoming and outgoing relationship of an absorbed node is rewired onto the survivor; a relationship that connected two nodes both being merged becomes a self-relationship on the survivor. This includes the incoming relationships of a `UNIDIRECTIONAL` edge type, which the absorbed node itself does not store; before v26.10.2 they were deleted with it.

APOC Compatible: `apoc.refactor.mergeNodes`

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Alice'})
WHERE elementId(a) <> elementId(b)
CALL refactor.mergeNodes([a, b], {properties: 'combine'}) YIELD node
RETURN node

// Keeping the survivor's own values on conflict:
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Alice'})
WHERE elementId(a) <> elementId(b)
CALL refactor.mergeNodes([a, b], {properties: 'discard'}) YIELD node
RETURN node

// Omitting config falls back to the default 'overwrite' policy:
```

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Alice'})
WHERE elementId(a) <> elementId(b)
CALL refactor.mergeNodes([a, b]) YIELD node
RETURN node
```

21.4. Schema & Meta Procedures

21.4.1. Meta/Schema Procedures

21.4.2. meta.graph()

Get a virtual graph representing the database schema structure.

Syntax: `CALL meta.graph([config]) YIELD nodes, relationships`

Returns:

- `nodes` - One virtual node per label that has nodes, carrying that label and the properties `name` and `count`
- `relationships` - One virtual relationship per start label, relationship type and end label that occur together, with the properties `type`, `out` (relationships of the type leaving a node with the start label), `in` (relationships of the type reaching a node with the end label) and `count` (all relationships of the type)

The optional `config` map is accepted for APOC compatibility and does not narrow the answer. The call reads every relationship record once, so its cost grows with the number of relationships.

APOC Compatible: `apoc.meta.graph`

```
CALL meta.graph() YIELD nodes, relationships
RETURN nodes, relationships
```

21.4.3. meta.nodetypeproperties()

Get property information for each node type.

Syntax: `CALL meta.nodetypeproperties([config]) YIELD nodeType, nodeLabels, propertyName, propertyTypes, mandatory, propertyObservations, totalObservations`

Returns: one row per label set and property found in the data or declared in the schema:

- `nodeType` - The labels in APOC's form, for example `` :`Person` ``
- `nodeLabels` - List of the labels
- `propertyName` - Name of the property, null for a label set whose nodes have no property

- `propertyTypes` - List of the value types found, with APOC's names (`String`, `Long`, `Double`, `Boolean`, `StringArray`, `LongArray`, `Date`, `DateTime`, `LocalDateTime`, `Duration`, `Point`, ...)
- `mandatory` - Whether the schema declares the property mandatory
- `propertyObservations` - How many of the examined nodes carry the property (0 for a property only declared in the schema)
- `totalObservations` - How many nodes were examined

The nodes are sampled as APOC samples them: every `count / sample`-th node of each label. The optional `config` map takes `sample` (default 1000, -1 reads every node), `includeLabels` and `excludeLabels` (and APOC's deprecated `labels` and `excludes`). Other keys are ignored.

APOC Compatible: `apoc.meta.nodeTypeProperties`

```
CALL meta.nodetypeproperties() YIELD nodeType, propertyName, propertyTypes
RETURN nodeType, propertyName, propertyTypes
```

21.4.4. meta.reltypeproperties()

Get property information for each relationship type.

Syntax: `CALL meta.reltypeproperties([config]) YIELD relType, sourceNodeLabels, targetNodeLabels, propertyName, propertyTypes, mandatory, propertyObservations, totalObservations`

Returns: one row per relationship type, start labels, end labels and property, with the same columns as `meta.nodetypeproperties()` plus:

- `relType` - The relationship type in APOC's form, for example `:`KNOWS``
- `sourceNodeLabels` - Labels of the start nodes
- `targetNodeLabels` - Labels of the end nodes

The relationships examined are the outgoing ones of the sampled nodes, at most `maxRels` (default 100) per node. The optional `config` map takes `sample`, `maxRels`, `includeLabels`, `excludeLabels`, `includeRels` and `excludeRels` (and APOC's deprecated `labels`, `rels` and `excludes`).

APOC Compatible: `apoc.meta.relTypeProperties`

```
CALL meta.reltypeproperties() YIELD relType, propertyName, propertyTypes
RETURN relType, propertyName, propertyTypes
```

21.4.5. meta.schema()

Get detailed schema information including all types and properties.

Syntax: `CALL meta.schema([config]) YIELD value`

The optional `config` map is accepted for APOC compatibility and does not change the answer.

Returns: `value` - Map containing:

- `nodeLabels` - List of vertex types with their properties
- `relationshipTypes` - List of edge types with their properties

APOC Compatible: `apoc.meta.schema`

```
CALL meta.schema() YIELD value
RETURN value.nodeLabels AS nodeTypes
```

21.4.6. meta.stats()

Get database statistics including counts of nodes and relationships.

Syntax: `CALL meta.stats() YIELD labelCount, relTypeCount, propertyKeyCount, nodeCount, relCount, labels, relTypes, relTypesCount, stats`

Returns: APOC's columns:

- `labelCount` - Number of node labels that have nodes
- `relTypeCount` - Number of relationship types that have relationships
- `propertyKeyCount` - Number of property keys, the ones the schema declares plus the ones found on a sample of the records
- `nodeCount` - Total number of nodes
- `relCount` - Total number of relationships
- `labels` - Map of label to count
- `relTypes` - Map of relationship pattern to count, with the keys `()-[:T]→()`, `(:L)-[:T]→()` and `()-[:T]→(:L)`
- `relTypesCount` - Map of relationship type to count
- `stats` - All the above in one map

A column `value`, a map with the same counts and `relTypes` by type name, is kept for the queries written before these columns existed. The call reads every relationship record once, so its cost grows with the number of relationships.

APOC Compatible: `apoc.meta.stats`

```
CALL meta.stats() YIELD nodeCount, relCount, labels
RETURN nodeCount AS nodes, relCount AS relationships, labels
```

21.4.7. Merge Procedures

21.4.8. merge.node()

Create or match a node by labels and properties.

Syntax: `CALL merge.node(labels, matchProps, createProps, onMatchProps) YIELD node`

Parameters:

Parameter	Type	Description
labels	List of String	Labels for the node
matchProps	Map	Properties to match on
createProps	Map	Properties to set when the node is newly created (optional)
onMatchProps	Map	Properties to set when the node already exists (optional)

Returns: `node` - The merged node

APOC Compatible: `apoc.merge.node`

```
CALL merge.node(["Person"], {email: "alice@example.com"}, {name: "Alice", created:
date.currentTimestamp()}) YIELD node
RETURN node

// with onMatchProps: bump a visit counter only when the node already existed
CALL merge.node(["Person"], {email: "alice@example.com"}, {name: "Alice"}, {lastSeen:
date.currentTimestamp()}) YIELD node
RETURN node
```

21.4.9. merge.relationship()

Create or match a relationship between two nodes.

Syntax: `CALL merge.relationship(startNode, relType, matchProps, createProps, endNode [, onMatchProps]) YIELD rel`

Parameters:

Parameter	Type	Description
startNode	Node	Starting node
relType	String	Relationship type name

Parameter	Type	Description
matchProps	Map	Properties to match on
createProps	Map	Properties to set on create
endNode	Node	Ending node
onMatchProps	Map	Optional. Properties to set when the relationship already exists. Default {}

Returns: `rel` - The merged relationship

`createProps` is applied only when the relationship is created, and `onMatchProps` only when an existing one is matched. Leaving `onMatchProps` out, or passing `{}`, leaves a matched relationship untouched, which is what the five-argument form has always done. A `LIGHTWEIGHT` edge type has no record to hold properties, so a non-empty `onMatchProps` on one is refused.

APOC Compatible: `apoc.merge.relationship`

```
MATCH (a:Person {name: "Alice"}), (b:Person {name: "Bob"})
CALL merge.relationship(a, "KNOWS", {}, {since: 2024}, b) YIELD rel
RETURN rel

// Set a property on create, update another one when the relationship is already
there:
MATCH (a:Person {name: "Alice"}), (b:Person {name: "Bob"})
CALL merge.relationship(a, "KNOWS", {}, {since: 2024}, b, {confirmed: true}) YIELD rel
RETURN rel

// With batch processing:
UNWIND $batch AS row
MATCH (a), (b) WHERE elementId(a) = row.source AND elementId(b) = row.target
CALL merge.relationship(a, row.relType, {}, row.props, b) YIELD rel
RETURN count(rel)
```

Text Functions

The text functions read a string by Unicode character, as `size()` counts it, so a character outside the Basic Multilingual Plane (an emoji, a letter of Deseret or Adlam, a CJK Extension B ideograph) is one character: `text.capitalize()`, `text.decapitalize()`, `text.capitalizeAll()` and `text.decapitalizeAll()` map its case, `text.camelCase()`, `text.snakeCase()` and `text.upperCamelCase()` take a letter outside the plane for a letter and not for a word break, `text.replace()` and `text.split()` with an empty regular expression cut between characters and never inside one, and `text.random()` draws whole characters. The openCypher `ltrim()`, `rtrim()`, `btrim()` and `trim()` with a set of characters to strip do the same. *(Since v26.11.1, issues #10484 and #10485)* Before, they worked on UTF-16 units and could

return half of a character. The distance and similarity functions (`text.hammingDistance()`, `text.levenshteinDistance()`, `text.levenshteinSimilarity()`, `text.jaroWinklerDistance()` and `text.sorensenDiceSimilarity()`) compare characters too *(since v26.11.1, issue #10417)*: one emoji replaced by another is one edit, `text.hammingDistance()` compares the number of characters and not of UTF-16 units, the 10,000 character limit of `text.levenshteinDistance()` counts characters, and a bigram of `text.sorensenDiceSimilarity()` is never cut inside a character. The number for two strings is the one for the same two strings written with characters of the Basic Multilingual Plane.

21.4.10. text.byteCount()

Get the byte count of a string.

Syntax: `text.byteCount(string)`

Returns: Long - Number of bytes

APOC Compatible: `apoc.text.byteCount`

21.4.11. text.camelCase()

Convert a string to camelCase. The words are the runs of letters and digits, each one lower-cased and then capitalized, and the first character of the result is lower case. The blanks and separators around the words are dropped.

Syntax: `text.camelCase(string)`

Returns: String - camelCase string

APOC Compatible: `apoc.text.camelCase`

(Since v26.12.1, issue #12104) A separator in front of the first word no longer capitalizes it: `' padded '` is `padded`, it was `Padded`.

```
RETURN text.camelCase("hello world") AS result
// Returns: "helloWorld"

RETURN text.camelCase("hello_world") AS result
// Returns: "helloWorld"

RETURN text.camelCase("  padded  ") AS result
// Returns: "padded"
```

21.4.12. text.capitalize()

Capitalize the first letter of a string.

Syntax: `text.capitalize(string)`

Returns: String - Capitalized string

APOC Compatible: `apoc.text.capitalize`

```
RETURN text.capitalize("hello") AS result
// Returns: "Hello"
```

21.4.13. text.capitalizeAll()

Capitalize the first letter of each word. The words are separated by spaces, and the result is trimmed.

Syntax: `text.capitalizeAll(string)`

Returns: String - String with all words capitalized

APOC Compatible: `apoc.text.capitalizeAll`

(Since v26.12.1, issue #12104) Only a space separates the words (`'a\tb c'` is `A\tb C`) and the blanks around the text are removed (`' padded '` is `Padded`). Before, any whitespace separated the words and the blanks were kept.

```
RETURN text.capitalizeAll("hello world") AS result
// Returns: "Hello World"
```

21.4.14. text.charAt()

Get the Unicode code point of the character at a specific index (0-based).

Syntax: `text.charAt(string, index)`

Returns: Long - Code point of the character at index, or `null` when the index is outside the string or an argument is `null`

APOC Compatible: `apoc.text.charAt`

(Since v26.12.1, issue #12104) The answer is the code point, as in APOC, where it was the one-character string. Use `text.code()` to turn a code point back into a string.

```
RETURN text.charAt("hello", 1) AS result
// Returns: 101
```

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10378) the count was in UTF-16 code units, so such a character counted 2 and `text.charAt()` could return half of it.

21.4.15. text.code()

Get the one-character string of a Unicode code point.

Syntax: `text.code(codePoint)`

Returns: String - The character, or `null` when the argument is `null`, negative or above U+10FFFF

APOC Compatible: `apoc.text.code`

(Since v26.12.1, issue #12104) The argument is the code point and the answer the string, as in APOC. Before, the function answered the code of the first character of the text it was given (`65` gave `54`, the code of `'6'`). A text that holds a whole number is read as that number, and a text that does not (`text.code('A')`) is an error. A code point above U+FFFF is answered as one character, where APOC answers `null`.

```
RETURN text.code(65) AS result
// Returns: "A"
```

21.4.16. text.decapitalize()

Decapitalize the first letter of a string.

Syntax: `text.decapitalize(string)`

Returns: String - Decapitalized string

APOC Compatible: `apoc.text.decapitalize`

21.4.17. text.decapitalizeAll()

Decapitalize the first letter of each word. The words are separated by spaces, and the result is trimmed.

Syntax: `text.decapitalizeAll(string)`

Returns: String - String with all words decapitalized

APOC Compatible: `apoc.text.decapitalizeAll`

(Since v26.12.1, issue #12104) Only a space separates the words and the blanks around the text are removed (`' Padded '` is `padded`), as in `text.capitalizeAll()`.

21.4.18. text.format()

Format a string using printf-style formatting. The values are given as one list. Number formatting (`%f`, `%e`, `%d`, etc.) does not depend on the server's configured locale: it follows the `language` argument, `'en'` by default (`.` as decimal separator).

Syntax: `text.format(format, params, [language])`

Parameters:

Parameter	Type	Description
format	String	The format, as in Java `String.format()`
params	List	The values for the format specifiers. A value that is not a list is the one value. When it is `null`, the format is returned as it is
language	String	Optional - Language code that gives the decimal and grouping separators (default: `'en'`)

Returns: String - Formatted string

APOC Compatible: `apoc.text.format`

(Since v26.12.1, issue #12104) The values are one list, as in APOC: `text.format('%s-%d', ['a', 3])`. Before, they were the following arguments, and `text.format('%s %d', 'a', 1)` now reads `'a'` as the one value and `1` as the language. The separators of a number follow the language.

```
RETURN text.format("%s-%d", ["a", 3]) AS result
// Returns: "a-3"

RETURN text.format("%,d", [1234567], "de") AS result
// Returns: "1.234.567"
```

21.4.19. text.hammingDistance()

Calculate the Hamming distance between two strings of equal length. Returns the number of positions at which the corresponding characters differ.

Syntax: `text.hammingDistance(string1, string2)`

Returns: Long - Number of positions with different characters

APOC Compatible: `apoc.text.hammingDistance`

```
SELECT text.hammingDistance('karolin', 'kathrin') AS distance
-- Returns: 3

SELECT text.hammingDistance(name, 'John') AS dist FROM Person
```

```
RETURN text.hammingDistance("karolin", "kathrin") AS distance
// Returns: 3
```

21.4.20. text.hexValue()

Convert a number to its hexadecimal representation, in upper case and padded with zeros: 4 digits up to `0xFFFF`, 8 digits up to `0xFFFFFFFF` and for a negative number in the 32-bit range (`-1` is `FFFFFFFF`), 16 digits for the rest. A string is written one UTF-16 unit at a time (4 digits each) and a byte array one byte at a time.

Syntax: `text.hexValue(number)`

Returns: String - Hexadecimal string

APOC Compatible: `apoc.text.hexValue`

(Since v26.12.1, issue #12104) The answer is upper case and padded, as in APOC. Before it was lower case and not padded (`255` gave `ff`).

```
RETURN text.hexValue(255) AS result
// Returns: "00FF"
```

21.4.21. text.indexOf()

Find the position of a substring within a string.

Syntax: `text.indexOf(string, substring, [start])`

Parameters:

Parameter	Type	Description
string	String	The string to search in
substring	String	The substring to find

Parameter	Type	Description
start	Integer	Optional - Starting position (default: 0)

Returns: Long - Position of substring, or -1 if not found

APOC Compatible: `apoc.text.indexOf`

Positions and lengths count Unicode characters: a character outside the Basic Multilingual Plane, such as an emoji, is one character and is never cut in two. Before 26.11.1 (issue #10378) the count was in UTF-16 code units, so such a character counted 2 and `text.charAt()` could return half of it.

```
RETURN text.indexOf("hello world", "world") AS pos
// Returns: 6

RETURN text.indexOf("hello hello", "hello", 1) AS pos
// Returns: 6
```

21.4.22. text.jaroWinklerDistance()

Calculate the Jaro-Winkler distance between two strings: 1 minus their Jaro-Winkler similarity. Returns a value between 0.0 (identical) and 1.0 (nothing in common). This metric is especially effective for detecting typos and misspellings because it gives extra weight to matching prefixes.

Syntax: `text.jaroWinklerDistance(string1, string2)`

Returns: Double - Distance (0.0 to 1.0), the smaller the more similar

APOC Compatible: `apoc.text.jaroWinklerDistance`

(Since v26.12.1, issue #12104) The function answers the distance, as APOC does. Before, it answered the similarity (1.0 for equal strings), so a query that kept the names with `> 0.85` must now keep those with `< 0.15`, and one that sorted by `score DESC` must sort by `score ASC`.

```
-- Find similar names using fuzzy matching
SELECT name, text.jaroWinklerDistance(name, 'John') AS score
  FROM Person
  WHERE text.jaroWinklerDistance(name, 'John') < 0.15
  ORDER BY score ASC

-- Compare two literal strings
SELECT text.jaroWinklerDistance('hello', 'helo') AS score
-- Returns: ~0.047
```

```
RETURN text.jaroWinklerDistance("hello", "helo") AS score
```

```
// Returns: ~0.047

MATCH (p:Person)
WHERE text.jaroWinklerDistance(p.name, "John") < 0.15
RETURN p.name, text.jaroWinklerDistance(p.name, "John") AS score
ORDER BY score ASC
```

21.4.23. text.join()

Join a list of strings with a delimiter.

Syntax: `text.join(list, delimiter)`

Parameters:

Parameter	Type	Description
list	List	List of strings to join
delimiter	String	String to insert between elements

Returns: String - Joined string, or `null` when the list or the delimiter is `null`

APOC Compatible: `apoc.text.join`

(Since v26.12.1, issue #12104) A `null` element is written as `null`, as in APOC, where it was written as an empty string, and a `null` delimiter answers `null`.

```
RETURN text.join(["a", "b", "c"], ",") AS result
// Returns: "a,b,c"

RETURN text.join(["a", null, "b"], ",") AS result
// Returns: "a,null,b"
```

21.4.24. text.levenshteinDistance()

Calculate the Levenshtein (edit) distance between two strings. Returns the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other.

Syntax: `text.levenshteinDistance(string1, string2)`

Returns: Long - Number of edits required

Each argument is limited to 10,000 characters. A longer string is rejected with `String length exceeds maximum allowed`, because the computation cost grows with

the two lengths multiplied together.

APOC Compatible: `apoc.text.levenshteinDistance`

```
SELECT text.levenshteinDistance('kitten', 'sitting') AS distance
-- Returns: 3

SELECT name, text.levenshteinDistance(name, 'John') AS edits
  FROM Person
  WHERE text.levenshteinDistance(name, 'John') <= 2
```

```
RETURN text.levenshteinDistance("kitten", "sitting") AS distance
// Returns: 3
```

21.4.25. text.levenshteinSimilarity()

Calculate the normalized Levenshtein similarity (0-1) between two strings. This is computed as `1.0 - (editDistance / maxLength)`, providing a score where 1.0 means identical strings.

Syntax: `text.levenshteinSimilarity(string1, string2)`

Returns: Double - Similarity score (0.0 to 1.0)

APOC Compatible: `apoc.text.levenshteinSimilarity`

```
SELECT text.levenshteinSimilarity('hello', 'hallo') AS similarity
-- Returns: 0.8
```

```
RETURN text.levenshteinSimilarity("hello", "hallo") AS similarity
// Returns: 0.8
```

21.4.26. text.lpad()

Left-pad a string to a specified length.

Syntax: `text.lpad(string, length, padChar)`

Returns: String - Padded string

APOC Compatible: `apoc.text.lpad`

The length is counted in Unicode characters, and a pad character outside the Basic

> Multilingual Plane is one character. Before 26.11.1 (issue #10378) it was counted in UTF-16 code units, so a string holding an emoji measured one more than its characters and was not padded to its width.

```
RETURN text.lpad("42", 5, "0") AS result
// Returns: "00042"
```

21.4.27. text.random()

Generate a random string.

Syntax: `text.random(length, [chars])`

Returns: String - Random string

APOC Compatible: `apoc.text.random`

21.4.28. text.regexReplace()

Replace all matches of a regular expression.

Syntax: `text.regexReplace(string, regex, replacement)`

Returns: String - String with replacements made

APOC Compatible: `apoc.text.regexReplace`

```
RETURN text.regexReplace("hello123world", "[0-9]+", "-") AS result
// Returns: "hello-world"
```

21.4.29. text.replace()

Replace every match of a regular expression. The replacement refers to the groups of the match as `$1` or `${name}`, and `\$` is a literal dollar sign.

Syntax: `text.replace(string, regex, replacement)`

Returns: String - String with replacements made, or `null` when an argument is `null`

APOC Compatible: `apoc.text.replace`

A pattern that is not valid, or a replacement that names a group the pattern does not have, is an error. The pattern is limited to 500 characters and runs under the `command.regexTimeout` deadline, as

in `text.regexReplace()`.

(Since v26.12.1, issue #12104) The second argument is a Java regular expression, as in APOC, where it was matched as literal text, and a `null` argument answers `null`. To replace literal text, escape the characters of the pattern that a regular expression reads (`"\\."` for a dot), or use the openCypher `replace()` function, which is literal.

```
RETURN text.replace("hello world", "world", "universe") AS result
// Returns: "hello universe"

RETURN text.replace("a1b22c", "[0-9]+", "#") AS result
// Returns: "a#b#c"

RETURN text.replace("hello", "(l+)", "<$1>") AS result
// Returns: "he<ll>o"
```

21.4.30. text.rpad()

Right-pad a string to a specified length.

Syntax: `text.rpad(string, length, padChar)`

Returns: String - Padded string

APOC Compatible: `apoc.text.rpad`

The length is counted in Unicode characters, and a pad character outside the Basic Multilingual Plane is one character. Before 26.11.1 (issue #10378) it was counted in UTF-16 code units, so a string holding an emoji measured one more than its characters and was not padded to its width.

```
RETURN text.rpad("42", 5, "0") AS result
// Returns: "42000"
```

21.4.31. text.slug()

Create a URL-friendly slug from a string: the string is trimmed and each run of characters other than letters, digits and `_` is replaced with the delimiter. The case and the letters of any alphabet are kept.

Syntax: `text.slug(string, [delimiter])`

Parameters:

Parameter	Type	Description
string	String	The string to convert
delimiter	String	Optional - Character to use as word separator (default: "-"), used as it is. When it is `null`, the answer is `null`

Returns: String - URL-friendly slug

APOC Compatible: `apoc.text.slug`

(Since v26.12.1, issue #12104) The case and the non-ASCII letters are kept, as in APOC. Before, the text was lower-cased and reduced to ASCII letters (`'Hello World'` gave `hello-world`).

```
RETURN text.slug("Hello World") AS result
// Returns: "Hello-World"

RETURN text.slug("Hello, World", "_") AS result
// Returns: "Hello_World"
```

21.4.32. text.snakeCase()

Convert a string to snake case: lower-case words joined by `-`, as APOC does. A word starts at every character that is not a lower-case letter or an ASCII digit, so `"fooBarBaz"` is `foo-bar-baz`. A string made only of upper-case letters, digits and `_` (`"FOO_BAR"`) is first read as words separated by the underscores. The blanks around a word are dropped, and any other character stays at the start of its word (`"a.b"` is `a-.b`).

Syntax: `text.snakeCase(string)`

Returns: String - Snake case string

APOC Compatible: `apoc.text.snakeCase`

(Since v26.12.1, issue #12104) The words are joined by `-`, where they were joined by `_`, and `"FOO_BAR"` is `foo-bar`, it was `f_o_o_b_a_r`.

```
RETURN text.snakeCase("helloWorld") AS result
// Returns: "hello-world"

RETURN text.snakeCase("FOO_BAR") AS result
// Returns: "foo-bar"
```

21.4.33. text.sorensenDiceSimilarity()

Calculate the Sorensen-Dice similarity coefficient between two strings based on bigram (character pair) overlap. Returns a value between 0.0 (no common bigrams) and 1.0 (identical). Useful for comparing short strings and detecting near-duplicates.

Syntax: `text.sorensenDiceSimilarity(string1, string2, [languageTag])`

Parameters:

Parameter	Type	Description
string1, string2	String	The strings to compare
languageTag	String	Optional - IETF language tag the strings are upper-cased in (default: `'en'`). When it is `null`, the answer is `null`

Returns: Double - Similarity score (0.0 to 1.0)

APOC Compatible: `apoc.text.sorensenDiceSimilarity`

(Since v26.12.1, issue #12104) The strings are upper-cased, split into words at the blanks, and the bigrams of each word are compared as a multiset, as APOC does: a bigram never spans two words, and `'aa'` and `'aaaa'` share one of their four bigrams (0.5). Before, the bigrams were a case-insensitive set over the whole string (1.0 for `'aa'` and `'aaaa'`). A language tag changes the case mapping (`'İstanbul'` and `'istanbul'` are 1.0 with `'tr'`).

```
SELECT text.sorensenDiceSimilarity('night', 'nacht') AS score
-- Returns: 0.25
```

```
RETURN text.sorensenDiceSimilarity("night", "nacht") AS score
// Returns: 0.25
```

21.4.34. text.split()

Split a string around the matches of a regular expression.

Syntax: `text.split(string, regex, [limit])`

Parameters:

Parameter	Type	Description
string	String	The string to split
regex	String	The Java regular expression that separates the pieces

Parameter	Type	Description
limit	Integer	Optional - With `0` (the default) the empty pieces at the end are dropped, with a negative number they are kept, and with a positive number the list has at most that many pieces, the last one holding the rest of the string

Returns: List - List of substrings, or `null` when an argument is `null`

APOC Compatible: `apoc.text.split`

The result is the one of Java `String.split(regex, limit)`. The pattern has the limits of `text.replace()`.

(Since v26.12.1, issue #12104) The second argument is a regular expression, as in APOC, where it was matched as literal text, and the empty pieces at the end are dropped unless `limit` is negative: `text.split("a,", ",")` is `["a"]`, and `text.split("a.b.c", ".")` is `[]` because `.` matches every character (write `"\\."` for a dot). The openCypher `split()` function and the SQL `.split()` method are not changed: they are literal and keep the empty pieces.

```
RETURN text.split("a,b,c", ",") AS parts
// Returns: ["a", "b", "c"]

RETURN text.split("a1b22c", "[0-9]+") AS parts
// Returns: ["a", "b", "c"]

RETURN text.split("a,b,,c,,", ",") AS parts
// Returns: ["a", "b", "", "c"]

RETURN text.split("a,b,,c,,", ",", -1) AS parts
// Returns: ["a", "b", "", "c", "", ""]

RETURN text.split("a,b,c", ",", 2) AS parts
// Returns: ["a", "b,c"]
```

21.4.35. text.upperCamelCase()

Convert a string to UpperCamelCase (PascalCase): `text.camelCase()` with the first character upper case.

Syntax: `text.upperCamelCase(string)`

Returns: String - UpperCamelCase string

APOC Compatible: `apoc.text.upperCamelCase`

(Since v26.12.1, issue #12104) The words are the runs of letters and digits, as in `text.camelCase()`: '

`padded '` is `Padded`.

```
RETURN text.upperCamelCase("hello world") AS result
// Returns: "HelloWorld"
```

21.5. Utility Functions

21.5.1. util.compress()

Compress data using gzip or deflate.

Syntax: `util.compress(data, [algorithm])`

Parameters:

Parameter	Type	Description
data	String	String data to compress
algorithm	String	Optional - "gzip" (default) or "deflate"

Returns: String - Base64-encoded compressed data

The input is limited to 10MB. A larger value is rejected with `Input size exceeds maximum allowed`, so that a single query cannot exhaust server memory.

APOC Compatible: `apoc.util.compress`

21.5.2. util.decompress()

Decompress base64-encoded compressed data.

Syntax: `util.decompress(data, [algorithm])`

Returns: String - Decompressed string

The decompressed result is limited to 100MB. Anything larger is rejected with `Decompressed output size exceeds maximum allowed`, which protects the server against a "zip bomb": a small payload that expands to an enormous one.

APOC Compatible: `apoc.util.decompress`

```
WITH util.compress("Hello World!") AS compressed
RETURN util.decompress(compressed) AS original
// Returns: "Hello World!"
```

21.5.3. util.md5()

Compute MD5 hash of a value.

Syntax: `util.md5(value)`

Returns: String - MD5 hash (32 hex characters)

APOC Compatible: `apoc.util.md5`

```
RETURN util.md5("hello") AS hash
// Returns: "5d41402abc4b2a76b9719d911017c592"
```

21.5.4. util.sha1()

Compute SHA-1 hash of a value.

Syntax: `util.sha1(value)`

Returns: String - SHA-1 hash (40 hex characters)

APOC Compatible: `apoc.util.sha1`

21.5.5. util.sha256()

Compute SHA-256 hash of a value.

Syntax: `util.sha256(value)`

Returns: String - SHA-256 hash (64 hex characters)

APOC Compatible: `apoc.util.sha256`

```
RETURN util.sha256("hello") AS hash
// Returns: "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
```

21.5.6. util.sha512()

Compute SHA-512 hash of a value.

Syntax: `util.sha512(value)`

Returns: String - SHA-512 hash (128 hex characters)

APOC Compatible: `apoc.util.sha512`

21.5.7. util.sleep()

Pause execution for a specified time.

Syntax: `util.sleep(milliseconds)`

Returns: Boolean - true after sleep

APOC Compatible: `apoc.util.sleep`

21.5.8. util.validate()

Abort the query with an error when a condition holds, as APOC's `apoc.util.validate` does: the predicate states the condition that must NOT be true. When `params` is a non-empty list, its values fill the `%s` placeholders of the message (Java `String.format`); without it the message is used as written.

Syntax: `util.validate(predicate, message [, params])`

Returns: Boolean - true when the predicate is false (or null)

APOC Compatible: `apoc.util.validate` (also callable with `CALL`)

```
CALL apoc.util.validate(false, 'bad %s', ['x']) RETURN 1 AS ok
// Returns: one row, ok = 1

CALL apoc.util.validate(true, 'bad %s', ['x']) RETURN 1 AS ok
// Throws an error with message "bad x"

// The usual guard: abort the write when the account cannot pay
MATCH (a:Account {id: $id})
CALL apoc.util.validate(a.balance < $amount, 'insufficient balance on %s', [$id])
SET a.balance = a.balance - $amount
```

Up to 26.10.2 `util.validate` took two arguments and threw when the predicate was **false**, the opposite of APOC. It now follows APOC: it throws when the predicate is **true**. A query written for the old behavior must negate its predicate (`util.validate(NOT (x > 0), …)`).

21.5.9. util.validatePredicate()

The function form of `util.validate()`, usable in a `WHERE` or a `RETURN`: throws when the predicate is true, returns true otherwise.

Syntax: `util.validatePredicate(predicate, message [, params])`

Returns: Boolean - true when the predicate is false (or null)

APOC Compatible: `apoc.util.validatePredicate`

```
MATCH (n:Person)
WHERE apoc.util.validatePredicate(n.age IS NULL, 'person %s has no age', [n.name])
RETURN n.name
```

21.6. Vector Functions

Vector functions provide comprehensive operations for vector embeddings, similarity search, and machine learning workflows. These functions are available in both SQL and Cypher queries.

Every vector function has two equivalent names: the namespaced `vector.<name>()` form (e.g. `vector.dimension()`) and a camelCase alias kept for backward compatibility (e.g. `vectorDimension()`). They resolve to the same function and are case-insensitive.

Prefer the namespaced `vector.<name>()` form in new queries and examples: it groups the suite consistently with the other extended namespaces (`geo.`, `math.`, ...) and reads more clearly. The namespaced form can be called directly, without back-ticks (e.g. `SELECT vector.cosineSimilarity(a, b)`). All examples in this section use it.

21.6.1. Aggregation Functions

There are two distinct categories of "statistics" functions, which is easy to confuse:

- **Row aggregates that return a vector** - `vector.sum()`, `vector.avg()`, `vector.min()`, `vector.max()` fold a column of vectors **element-wise across rows**. For example `vector.avg(embedding)` returns the **centroid** (the element-wise arithmetic mean) of all the embeddings in the result set.
- **Per-vector element statistics that return a scalar** - `vector.variance()` and `vector.stddev()` operate on a **single** vector and return the variance / standard deviation of **that vector's elements**. They are intentionally **not** row aggregates (unlike the scalar SQL `variance()`/`stddev()`).

vector.sum()

Aggregate function: element-wise sum of vectors.

Syntax: `vector.sum(<field>)`

Returns: Vector - Sum of all vectors

```
SELECT vector.sum(embedding) FROM documents
```

---

vector.avg()

Aggregate function: element-wise average of vectors.

Syntax: `vector.avg(<field>)`

Returns: Vector - Average vector

```
SELECT vector.avg(embedding) FROM documents
```

---

vector.min()

Aggregate function: element-wise minimum of vectors.

Syntax: `vector.min(<field>)`

Returns: Vector - Minimum vector

```
SELECT vector.min(embedding) FROM documents
```

---

vector.max()

Aggregate function: element-wise maximum of vectors.

Syntax: `vector.max(<field>)`

Returns: Vector - Maximum vector

```
SELECT vector.max(embedding) FROM documents
```

21.6.2. Basic Operations

vector.dimension()

Returns the dimension of a vector (length of the underlying array).

Syntax: `vector.dimension(<vector>)`

Returns: Integer - Vector dimension. A `NULL` argument returns `0` (consistent with `.size()`/`.length()`).

```
SELECT vector.dimension([1.0, 2.0, 3.0])
-- Returns: 3

SELECT vector.dimension(null)
-- Returns: 0
```

```
RETURN vector.dimension([1.0, 2.0, 3.0]) AS dim
```

---

vector.add()

Returns element-wise sum of two vectors, or adds a scalar to every element (broadcasting). The scalar may be either argument since addition is commutative.

Syntax: `vector.add(<vector1>, <vector2> | <scalar>)`

Returns: Vector - Sum vector

```
SELECT vector.add([1.0, 2.0, 3.0], [2.0, 3.0, 4.0])
-- Returns: [3.0, 5.0, 7.0]

SELECT vector.add([1.0, 2.0, 3.0], 4.0)
-- Returns: [5.0, 6.0, 7.0]  (scalar broadcasting)
```

---

vector.subtract()

Returns element-wise difference of two vectors, or subtracts with a scalar (broadcasting). Order is preserved: `vector - scalar` subtracts the scalar from each element, while `scalar - vector` subtracts each element from the scalar.

Syntax: `vector.subtract(<vector1> | <scalar>, <vector2> | <scalar>)`

Returns: Vector - Difference vector

```
SELECT vector.subtract([3.0, 5.0, 7.0], [1.0, 2.0, 3.0])
```

```
-- Returns: [2.0, 3.0, 4.0]

SELECT vector.subtract([1.0, 2.0, 3.0], 1.0)
-- Returns: [0.0, 1.0, 2.0]  (vector - scalar)

SELECT vector.subtract(10.0, [1.0, 2.0, 3.0])
-- Returns: [9.0, 8.0, 7.0]  (scalar - vector)
```

---

vector.scale()

Scales vector element-wise by a scalar value.

Syntax: `vector.scale(<vector>, <scalar>)`

Returns: Vector - Scaled vector

```
SELECT vector.scale([1.0, 2.0, 3.0], 2.0)
-- Returns: [2.0, 4.0, 6.0]
```

---

vector.clip()

Clips (clamps) vector elements to a specified range [min, max]. Also available as `vector.clamp()` / `vectorClamp()`.

Syntax: `vector.clip(<vector>, <min>, <max>)`

Returns: Vector - Clipped vector

```
SELECT vector.clip([1.0, 5.0, 10.0], 2.0, 8.0)
-- Returns: [2.0, 5.0, 8.0]

SELECT vector.clamp([1.0, 5.0, 10.0], 2.0, 8.0)
-- Returns: [2.0, 5.0, 8.0]  (clamp is an alias of clip)
```

21.6.3. Nearest Neighbor Search

vector.neighbors()

Returns k nearest neighbors from a vector index.

Syntax:

```
vector.neighbors(<index-spec>, <query-vector>, <k>)
```

```
vector.neighbors(<index-spec>, <query-vector>, <k>, <efSearch>)
vector.neighbors(<index-spec>, <query-vector>, <k>, { <option>: <value>, ... })
```

Parameters:

- `index-spec` - Index specification as `'TypeName[propertyName]'`
- `query-vector` - Query vector or key to look up
- `k` - Number of neighbors to return. Must be a non-negative integer: `0` returns no rows (like `LIMIT 0`), a negative or fractional value is rejected with an error. The same rule applies to the `k` of `vector.sparseNeighbors()`, `vector.discover()`, `vector.recommend()` and `vector.rerank()`. Since 26.11.1 (a negative `k` returned no rows before).
- `efSearch` - (optional, positional) Search beam width. Controls the trade-off between recall and speed. Higher values improve recall but increase latency. When omitted, the index uses its default beam, sized from the graph.

Options map (optional, alternative to the positional `efSearch` argument). An option of type Integer accepts a whole number only (`2` or `2.0`): a fractional value such as `2.5` is rejected with an error instead of being truncated, since 26.11.1.

Key	Type	Description
`efSearch`	Integer	Same semantics as the positional form.
`filter`	List of RIDs or RID strings	Restricts the search to the provided set of records. Useful to combine a vector search with a logical filter: first select the candidate RIDs, then pass them as `filter`. An empty list (or a subquery that returns no rows) matches nothing, so the function returns no results. Omit the option, or pass `null`, to search without a restriction. Unknown keys are rejected.
`where`	Map or String	Restricts the search to the records that satisfy a predicate, tested on each candidate as the search meets it: a map of property to value, every entry of which must hold with the SQL `=` semantics (`{ embed_space: :space }`, nested paths such as `meta.source` allowed), or a SQL condition (`'year >= :minYear AND lang = :lang'`, whose parameters are the query's). Unlike a `filter` subquery it does not load every record in scope and list its RID on every call: only the candidates the search considers are loaded, once each. Combined with `filter`, a candidate must be in the list and satisfy the predicate. Also accepted by `vector.sparseNeighbors`. Since 26.11.1.

Key	Type	Description
`groupBy`	String	Diversifies the answer by a property: `k` becomes the number of distinct values of that property to return rather than the number of rows, and at most `groupSize` rows are returned per value. Supports nested paths (`meta.source`). Records whose property is missing share the single `null` group.
`groupSize`	Integer	Rows to return per group when `groupBy` is set. Defaults to `1`, so the answer is the `k` nearest records with `k` distinct values. Must be `>= 1`; ignored when `groupBy` is absent.
`maxDistance`	Number	Drops neighbors further than this distance, turning the query into a radius search bounded by `k`. The result may legitimately hold fewer than `k` rows. Defaults to no bound.
`rerank`	Integer	`PRODUCT` and `BINARY` indexes only. Candidates per requested neighbor that the search, which walks the graph on the in-memory PQ codes (or one-bit codes for `BINARY`, since 26.11.1), scores again on the stored vectors before returning the nearest `k` with their exact distances. A `BINARY` index re-ranks at least one candidate per neighbor. `0` returns the PQ scores as they are, without reading a stored vector: fastest, but the recall of PQ scores alone is low and the distances are approximate (and not comparable across buckets, each of which has its own codebook). Defaults to the `rerankOversample` of the index (10). Ignored with `groupBy` and by the other quantizations. Since 26.11.1.

Prefer `where` to a `filter` subquery when the restriction is a property of the records and keeps a sizeable share of them: the subquery loads every record in scope on each call, the predicate only the candidates the search reaches. A `filter` remains the better choice for a small, precomputed list of RIDs, which the index can scan directly. A predicate that keeps few or none of the records leaves the graph walk short of `k`, and the search then tests every indexed record once to return an exact answer: the same cost the `filter` subquery pays on every call. The query's timeout is checked before each record the predicate tests, so such a search stops at its deadline. The same applies to the `WHERE` of a Cypher `SEARCH` clause, which is tested by the index the same way.

Index the properties a selective `where` filters on. When the equalities of the predicate (the map form, or the `property = value` conditions of a SQL condition without `OR`) cover an `LSM_TREE` or `HASH` index, the search lists the matching records through that index as soon as it sees that the predicate keeps few records, and scores them directly instead of walking the graph (since 26.11.1). On 100,000 vectors, 20 matching records then cost about 0.3 ms instead of about 120 ms. A predicate that keeps a large share of the records keeps the graph walk, at no extra cost. The switch is bounded by `arcadedb.vectorIndex.prefilterMaxSelectivity`, and

> setting it to 0 disables it.

Returns: List - Nearest neighbors with distances

Each row holds the record's properties, its `@rid` and `@type`, and the `distance`. The rows of `vector.sparseNeighbors()`, `vector.fuse()`, `vector.rerank()`, `vector.mmr()`, `vector.boost()` and `vector.discover()` have the same shape, with their own score. The record itself is reachable under the `record` key (`record.name`, or `SELECT record FROM (…)`), which a re-ranker downstream uses to score the row without loading it again, but `SELECT *`, `expand()` and the JSON of the row do not list it, since its properties are the row's already. Before 26.11.1 every row also carried the whole record under `record`, so each property was serialized twice.

```
-- Default beam
SELECT vector.neighbors('Document[embedding]', [0.1, 0.2, 0.3], 5)

-- Explicit efSearch for higher recall (positional form, backward compatible)
SELECT vector.neighbors('Document[embedding]', [0.1, 0.2, 0.3], 5, 500)

-- Options map form (named, extensible)
SELECT vector.neighbors('Document[embedding]', [0.1, 0.2, 0.3], 5, { efSearch: 500 })

-- Combine a vector search with a logical filter on the same type
SELECT vector.neighbors(
         'Document[embedding]',
         [0.1, 0.2, 0.3],
         10,
         { filter: (SELECT @rid FROM Document WHERE tenantId = 'acme' AND category =
'finance') }
       )

-- Using a parameter binding for the RID list
SELECT vector.neighbors('Document[embedding]', :queryVector, 10, { efSearch: 300,
filter: :allowedRids })

-- Restrict by a property without listing every RID in scope
SELECT vector.neighbors('Document[embedding]', :queryVector, 10, { where: { tenantId:
:tenant } })

-- ... or by any SQL condition
SELECT vector.neighbors('Document[embedding]', :queryVector, 10, { where: 'year >=
:minYear AND category <> :excluded' })

-- One result per source document: 10 distinct sources, nearest chunk of each
SELECT vector.neighbors('Chunk[embedding]', :queryVector, 10, { groupBy: 'sourceFile'
})

-- 5 distinct sources, up to 3 chunks each, none further away than 0.35
SELECT vector.neighbors(
         'Chunk[embedding]',
         :queryVector,
```

```
      5,
      { groupBy: 'sourceFile', groupSize: 3, maxDistance: 0.35 }
    )
```

`groupBy` changes how rows are capped and nothing else about what the index can see: a grouped query is answered from the same corpus as the ungrouped one, including vectors written since the graph was last rebuilt.

A grouped query can return fewer than `k` groups, and two index statistics tell you which of the two reasons applies:

- `groupedSearchesShortOfLimit` - the search ran out of candidate budget before reaching the `k`-th nearest group, which happens when the nearest groups are very dense. Raising `efSearch` is what helps here.
- `groupedSearchesGroupsUnavailable` - there were no more groups to find. Usually this simply means the data holds fewer distinct values of the `groupBy` property than you asked for: a query with `k` of 10 over a corpus with 3 source files can only ever return 3 groups. That is expected and needs no action - raising `efSearch` cannot create a group that does not exist, and only makes the query slower. Look into it only if you expected more groups than you got, which can indicate a damaged index graph that `REBUILD INDEX` repairs.

The options map rejects unknown keys with a descriptive error to catch typos, for example passing `{ efsearch: 500 }` (lowercase) will fail. Keys are case sensitive.

Cypher Usage:

The recommended way to use vector search from Cypher is via `CALL`, which uses the HNSW index for fast approximate nearest neighbor search:

```
// ArcadeDB-native syntax
CALL vector.neighbors('Document[embedding]', $queryVector, 10)
YIELD name, distance
RETURN name, distance
ORDER BY distance

// Neo4j-compatible syntax (returns node + score)
CALL db.index.vector.queryNodes('Document[embedding]', 10, $queryVector)
YIELD node, score
RETURN node.title AS title, score
ORDER BY score DESC
```

`db.index.vector.queryNodes()` returns `score` in [0, 1] (1.0 = identical, as Neo4j reports it), while `vector.neighbors()` returns `distance` (0.0 = identical). For COSINE the relationship is `score = 1 - distance / 2`; for DOT_PRODUCT `score = -distance`; for EUCLIDEAN `score = 1 / (1 + distance)`.

A `WHERE` after the `YIELD` runs on the `k` rows the search already returned, so it can leave fewer than `k` rows, or none. To filter during the search, pass the `where` or `filter` option: to `vector.neighbors()` in its options map (`where` since 26.11.1), as in SQL above, or to `db.index.vector.queryNodes()` in an optional fourth argument (since 26.11.1), for example `CALL db.index.vector.queryNodes('Document[embedding]', 10, $queryVector, { where: { tenantId: 'acme' } })`. The fourth argument is an ArcadeDB extension to the Neo4j signature.

Type-Specific Search with Inheritance:

When a vector index exists on a parent type, you can search specific child types:

```
-- Search only in EMBEDDING_IMAGE records
SELECT vector.neighbors('EMBEDDING_IMAGE[vector]', :queryVector, 10)

-- Search across all types (parent + children)
SELECT vector.neighbors('EMBEDDING[vector]', :queryVector, 10)
```

21.6.4. Normalization and Norms

vector.normalize()

Normalizes vector to unit length (L2 norm = 1.0).

Syntax: `vector.normalize(<vector>)`

Returns: Vector - Normalized vector

```
SELECT vector.normalize([3.0, 4.0])
-- Returns: [0.6, 0.8]
```

---

vector.isnormalized()

Checks if vector is normalized (L2 norm ≈ 1.0). The optional `tolerance` defaults to `0.001`, which is an appropriate loose bound for ArcadeDB's `float` vectors (`sqrt(float eps)` ≈ `3.45e-4`); pass a smaller value to tighten the check.

Syntax: `vector.isnormalized(<vector>, [tolerance])`

Returns: Boolean - true if normalized

```
SELECT vector.isnormalized([0.6, 0.8])
```

```
-- Returns: true
```

---

vector.magnitude()

Computes L2 norm (Euclidean length) of vector. Also available as `vector.l2Norm()` / `vectorL2Norm()`, symmetric with `vector.l1Norm()` and `vector.lInfNorm()`.

Syntax: `vector.magnitude(<vector>)`

Returns: Double - L2 norm

```
SELECT vector.magnitude([3.0, 4.0])
-- Returns: 5.0

SELECT vector.l2Norm([3.0, 4.0])
-- Returns: 5.0  (l2Norm is an alias of magnitude)
```

---

vector.l1norm()

Computes L1 norm (Manhattan norm) of vector.

Syntax: `vector.l1norm(<vector>)`

Returns: Double - L1 norm

```
SELECT vector.l1norm([1.0, 2.0, 3.0])
-- Returns: 6.0
```

---

vector.linfnorm()

Computes L∞ norm (maximum absolute value) of vector.

Syntax: `vector.linfnorm(<vector>)`

Returns: Double - L∞ norm

```
SELECT vector.linfnorm([1.0, -5.0, 3.0])
-- Returns: 5.0
```

21.6.5. Quantization

vector.quantizeint8()

Quantizes vector to 8-bit integers using min-max scaling.

Syntax: `vector.quantizeint8(<vector>)`

Returns: ByteArray - Quantized vector

```
SELECT vector.quantizeint8([0.1, 0.5, 0.9])
```

---

vector.dequantizeint8()

Dequantizes 8-bit integers back to float vector (approximate recovery). Pass the result of `vector.quantizeint8()` directly (the `min`/`max` are read from it), or supply the bytes with explicit `min`/`max`.

Syntax: `vector.dequantizeint8(<result>)` or `vector.dequantizeint8(<quantized>, <min>, <max>)`

Returns: Vector - Dequantized vector

```
SELECT vector.dequantizeint8(vector.quantizeint8([1.0, 2.0, 3.0]))
-- Returns: ~[1.0, 2.0, 3.0]
```

---

vector.quantizebinary()

Quantizes vector to binary (1 bit per dimension) using median threshold. The median is used (rather than a fixed value like 0) so the bits split ~50/50, which maximizes the information retained per bit for arbitrarily-centered embeddings.

Syntax: `vector.quantizebinary(<vector>)`

Returns: ByteArray - Binary quantized vector

```
SELECT vector.quantizebinary([0.1, 0.5, 0.9])
```

---

vector.dequantizebinary()

Reconstructs an approximate float vector from a binary-quantized result. Binary quantization keeps only one bit per dimension (the sign relative to the median), so this is lossy: each bit maps to `highValue` (default `1.0`) when set or `lowValue` (default `-1.0`) when clear. Accepts the result of `vector.quantizebinary()` directly.

Syntax: `vector.dequantizebinary(<quantized> [, <lowValue>, <highValue>])`

Returns: Vector - Reconstructed (sign) vector

```
SELECT vector.dequantizebinary(vector.quantizebinary([0.1, 0.5, 0.9]))
-- Returns: [-1.0, 1.0, 1.0]
```

21.6.6. Scoring and Fusion

vector.fuse()

Fuses the ranked rows of two or more sources into one ranking, server-side. See Hybrid Search with `vector.fuse` for the strategies and worked examples.

Syntax: `vector.fuse(<source1>, <source2>, ⋯ [, { <option>: <value>, ⋯ }])`

Parameters:

- `source` - At least two. Each is a list of rows carrying a `@rid`: the output of `vector.neighbors` or `vector.sparseNeighbors`, a sub-SELECT such as `(SELECT @rid, $score FROM Doc WHERE SEARCH_INDEX('Doc[content]', :q) = true LIMIT 100)`, or `SEARCH_INDEX('Doc[content]', :q)` passed directly.

Options map:

Key	Type	Description
`fusion`	String	`RRF` (default), `DBSF` or `LINEAR`.
`k`	Integer	RRF constant. Defaults to `60`. `0` is allowed (each term is then `1/rank`); a negative value is refused with an error *(since v26.12.1; before, it gave `Infinity` or put the worst-ranked results first)*.
`weights`	List of numbers	One weight per source. Defaults to `1.0` for each.
`groupBy`	String	Keeps at most `groupSize` rows per value of this property.
`groupSize`	Integer	Rows per group when `groupBy` is set. Defaults to `1`.
`limit`	Integer	Maximum number of rows returned. Defaults to no limit: every fused candidate is returned, and loaded.

Returns: List - One row per fused record, with its properties and the fused `score`, best first.

Every row of every source is fused, and without `limit` every fused candidate is then loaded, so the cost grows with the number of rows the sources produce. A full-text leg without a `LIMIT` produces one row per match, so the query gets slower as the full-text match count grows, whatever `k` the other legs use. Bound both, as shown in Bounding the cost of a full-text leg.

Since 26.11.1, `SEARCH_INDEX(⋯)` passed directly as a source carries the score of each match, so

`fusion: 'DBSF'` and `'LINEAR'` accept a full-text leg written that way (before, they refused it for lack of a score).

---

vector.rerank()

Re-scores the candidates of a first, cheaper search on full-precision vectors and returns the best `k`: the "prefetch and rerank" pattern. The first stage can use a lossy index (`BINARY`, `PRODUCT`, sparse, full-text) with a high over-fetch, since the rerank corrects its ranking.

Syntax: `vector.rerank(<source>, <query-vector>, <embedding-property>, <k>)`

Parameters:

- `source` - The candidate rows: the output of `vector.neighbors`, `vector.fuse`, or a sub-SELECT returning records.
- `query-vector` - The full-precision query vector.
- `embedding-property` - The property holding each candidate's full-precision vector. A candidate without it, or whose record no longer exists, is skipped. A vector whose dimension differs from the query's is an error.
- `k` - Number of rows to return.

Returns: List - The `k` candidates with the highest cosine similarity to the query vector, as a `score`, best first.

```
SELECT expand(`vector.rerank`(
    `vector.neighbors`('Doc[binary_embedding]', :qBinary, 1000),
    :qFull,
    'embedding',
    100))
```

Since 26.11.1, a candidate row that already carries its record (the rows of `vector.fuse`, `vector.neighbors` and a plain `SELECT` do) is scored on that record without being reloaded, and only the best `k` candidates are kept while scanning. The cost still includes reading the embedding of every candidate, so bound the source (`k` of `vector.neighbors`, `limit` of `vector.fuse`) to the number of candidates worth re-scoring.

---

vector.hybridscore()

Computes weighted average of two scores.

Syntax: `vector.hybridscore(<score1>, <score2>, <alpha>)`

Returns: Double - Weighted average

```
SELECT vector.hybridscore(0.8, 0.6, 0.7)
```

```
-- Returns: 0.7 * 0.8 + 0.3 * 0.6 = 0.74
```

---

vector.multiscore()

Combines multiple scores using a fusion method.

Syntax: `vector.multiscore(<scores>, <method>, [weights])`

Methods:

- `'MAX'` - Maximum score (ColBERT style)
- `'AVG'` - Arithmetic average
- `'MIN'` - Minimum score
- `'WEIGHTED'` - Weighted average (requires weights)

Returns: Double - Combined score

```
SELECT vector.multiscore([0.9, 0.7, 0.8], 'MAX')
-- Returns: 0.9
```

---

vector.rrfscore()

Computes Reciprocal Rank Fusion (RRF) for combining multiple rankings.

Syntax: `vector.rrfscore(<rank1>, <rank2>, ⋯, [{ k: <long> }])` or `vector.rrfscore([<ranks>], [{ k: <long> }])`

Ranks may be passed as variadic arguments or grouped in a single array/list (consistent with `vector.multiscore`). Every positional numeric argument is treated as a rank. The center constant `k` (default 60) is set only via the trailing options map `{ k: <long> }` - a bare trailing number is always a rank, never `k`.

Parameters:

- `k` - Center rank constant (default: 60), via `{ k: <long> }`. It must not be negative: a negative `k` is refused with an error *(since v26.12.1)*.

Returns: Double - RRF score

```
SELECT vector.rrfscore(1, 2, 4)
-- Returns: 1/61 + 1/62 + 1/64

SELECT vector.rrfscore([1, 2, 4], { k: 100 })
-- Returns: 1/101 + 1/102 + 1/104
```

vector.normalizescores()

Normalizes scores to [0, 1] range using min-max normalization.

Syntax: `vector.normalizescores(<scores>)`

Returns: Vector - Normalized scores

```
SELECT vector.normalizescores([1.0, 2.0, 3.0])
-- Returns: [0.0, 0.5, 1.0]
```

vector.scoretransform()

Transforms scores using various functions.

Syntax: `vector.scoretransform(<score>, <method>)`

Methods:

- `'LINEAR'` - No transformation
- `'SIGMOID'` - Logistic function, maps to (0, 1)
- `'TANH'` - Hyperbolic tangent, maps to (-1, 1)
- `'LOG'` / `'LN'` - Natural logarithm (requires a positive score); `LN` is the clearer synonym
- `'EXP'` - Exponential function

Returns: Double - Transformed score

```
SELECT vector.scoretransform(0.5, 'SIGMOID')

SELECT vector.scoretransform(0.5, 'TANH')
-- Returns: 0.4621172

SELECT vector.scoretransform(2.5, 'LN')
-- Returns: 0.9162907  (same as 'LOG')
```

21.6.7. Similarity and Distance

vector.dotproduct()

Computes dot product (inner product) between two vectors.

Syntax: `vector.dotproduct(<vector1>, <vector2>)`

Returns: Double - Dot product

```
SELECT vector.dotproduct([1.0, 2.0, 3.0], [4.0, 5.0, 6.0])
-- Returns: 32.0
```

---

vector.cosinesimilarity()

Computes cosine similarity between two vectors. Returns value between -1 and 1.

Syntax: `vector.cosinesimilarity(<vector1>, <vector2>)`

Returns: Double - Cosine similarity (-1 to 1)

```
SELECT vector.cosinesimilarity([1.0, 0.0], [1.0, 0.0])
-- Returns: 1.0 (identical direction)

SELECT vector.cosinesimilarity([1.0, 0.0], [0.0, 1.0])
-- Returns: 0.0 (orthogonal)
```

---

vector.l2distance()

Computes L2 distance (Euclidean distance) between two vectors.

Syntax: `vector.l2distance(<vector1>, <vector2>)`

Returns: Double - Euclidean distance

```
SELECT vector.l2distance([0.0, 0.0], [3.0, 4.0])
-- Returns: 5.0
```

---

vector.approxdistance()

Computes approximate distance between quantized vectors without full dequantization. When you pass the result objects of `vector.quantizeint8()`/`vector.quantizebinary()`, the mode is inferred and the third argument can be omitted; the explicit mode is still required for raw byte arrays.

Syntax: `vector.approxdistance(<quantized1>, <quantized2> [, <mode>])`

Modes:

- `'INT8'` - Faster than floats; preserves the top-k ordering of the true distances (the approximate scalar distances rank in the same order, even though their absolute values differ). Accepts a `vector.quantizeint8()` result or a raw int8 byte array.

- `'BINARY'` - Very fast Hamming distance, 8x fewer operations. Requires the `vector.quantizebinary()` result object (raw packed bits are not enough - the Hamming normalization needs the original length).

Returns: Double - Approximate distance

```
-- mode inferred from the quantization result objects
SELECT vector.approxdistance(
  vector.quantizeint8([1.0, 2.0, 3.0]),
  vector.quantizeint8([1.0, 3.0, 3.0])
)
```

21.6.8. Sparse Vectors

vector.densetosparse()

Converts dense vector to sparse representation.

Syntax: `vector.densetosparse(<vector>, [threshold])`

Parameters:

- `threshold` - Values below this are considered zero (default: 0.0)

Returns: SparseVector - Sparse representation

```
SELECT vector.densetosparse([0.5, 0.0, 0.1], 0.2)
-- Only keeps elements >= 0.2
```

---

vector.sparsetodense()

Converts sparse vector back to dense representation.

Syntax: `vector.sparsetodense(<sparsevector>)`

Returns: Vector - Dense vector

```
SELECT vector.sparsetodense(vector.sparsecreate([0, 2], [0.5, 0.3]))
```

---

vector.sparsecreate()

Creates sparse vector from indices and values.

Syntax: `vector.sparsecreate(<indices>, <values>, [dimension])`

Returns: SparseVector - Sparse vector

```
SELECT vector.sparsecreate([0, 2, 5], [0.5, 0.3, 0.8], 7)
```

---

vector.sparsedot()

Computes dot product between two sparse vectors.

Syntax: `vector.sparsedot(<sparse1>, <sparse2>)`

Returns: Double - Dot product

```
SELECT vector.sparsedot(
  vector.densetosparse([1.0, 2.0, 3.0]),
  vector.densetosparse([1.0, 1.0, 1.0])
)
```

---

vector.sparsity()

Returns a sparsity measure of a vector. The threshold is optional and defaults to `sqrt(eps)` for float (~3.45e-4). An optional third argument selects the measure.

Syntax: `vector.sparsity(<vector> [, <threshold> [, <mode>]])`

Modes:

- `'FRACTION'` (default) - fraction of elements with `|x| < threshold`, in [0, 1]
- `'L0'` - the L0 pseudonorm: count of "significant" elements with `|x| >= threshold` (integer)
- `'GMEAN'` - geometric mean of the absolute values, `exp(mean(ln|x|))`; returns 0 if any element is 0

Returns: Double (FRACTION, GMEAN) or Integer (L0)

```
SELECT vector.sparsity([0.01, 0.1, 0.05, 0.02], 0.06)
-- Returns: 0.75 (3 of 4 elements are below 0.06)

SELECT vector.sparsity([0.0, 1.0, 2.0, 3.0])
-- Returns: 0.25 (uses the default threshold; one near-zero element)

SELECT vector.sparsity([0.0, 1.0, 2.0, 3.0], 0.5, 'L0')
-- Returns: 3 (three significant elements)

SELECT vector.sparsity([1.0, 2.0, 4.0], 0.0, 'GMEAN')
```

```
-- Returns: 2.0 (geometric mean)
```

21.6.9. Statistics

vector.variance()

Computes variance of vector elements.

Syntax: `vector.variance(<vector>)`

Returns: Double - Variance

```
SELECT vector.variance([1.0, 2.0, 3.0])
```

vector.stddev()

Computes standard deviation of vector elements.

Syntax: `vector.stddev(<vector>)`

Returns: Double - Standard deviation

```
SELECT vector.stddev([1.0, 2.0, 3.0])
```

vector.hasnan()

Checks if vector contains NaN values. A `NULL` element (e.g. produced by an invalid math op such as `sqrt(-1.0)` that the engine coerces to `NULL` inside a collection) is treated as NaN.

Syntax: `vector.hasnan(<vector>)`

Returns: Boolean - true if contains NaN

```
SELECT vector.hasnan([1.0, 2.0, 3.0])
-- Returns: false

SELECT vector.hasnan([1.0, sqrt(-1.0), 3.0])
-- Returns: true (the NULL element counts as NaN)
```

vector.hasinf()

Checks if vector contains infinity values. A `NULL` element is treated as NaN, which is not infinite, so it does not trigger a true result.

Syntax: `vector.hasinf(<vector>)`

Returns: Boolean - true if contains infinity

```
SELECT vector.hasinf([1.0, 10e400, 2.0])
-- Returns: true
```

---

vector.hasnull()

Checks if a vector contains any `NULL` element. Unlike `vector.hasnan()`, this distinguishes a genuine `NULL` (a missing element) from a NaN float value. Primitive `float[]`/`double[]`/`int[]`/`long[]` inputs can never hold a `NULL` and therefore always return `false`.

Syntax: `vector.hasnull(<vector>)`

Returns: Boolean - true if contains a NULL element

```
SELECT vector.hasnull([1.0, 2.0, 3.0])
-- Returns: false

SELECT vector.hasnull([1.0, sqrt(-1.0), 3.0])
-- Returns: true
```

21.6.10. Utility Functions

vector.tostring()

Converts vector to string representation. Equivalent to the `asString()` method on a vector value (e.g. `embedding.asString('PYTHON')`); both share the same formatter.

Syntax: `vector.tostring(<vector>, [format])`

Formats:

- `'COMPACT'` - Single line `[1.0, 2.0, 3.0]` (default)
- `'PRETTY'` - Multi-line with formatting
- `'PYTHON'` - Python list format
- `'MATLAB'` - space-separated row vector `[1.0 2.0 3.0]`
- `'MATLAB_COLUMN'` - semicolon-separated column vector `[1.0; 2.0; 3.0]`

- `'JULIA'` - Julia vector literal `[1.0, 2.0, 3.0]`
- `'NUMPY'` - bare comma-separated `1.0, 2.0, 3.0` (no brackets), for `numpy.fromstring(…, sep=",")`

Returns: String - Formatted vector

```
SELECT vector.tostring([0.5, 0.25, 0.75], 'PYTHON')

SELECT vector.tostring([0.5, 0.25, 0.75], 'NUMPY')
-- Returns: 0.5, 0.25, 0.75
```