ArcadeDB Manual

Version 24.3.1

Table of Contents

Chapter 1. Jump to the Hot Topics

Skip the boring parts and check this out:

- Run ArcadeDB with Open Beer demo database in 30 seconds
- What is Multi-Model?
- ArcadeDB supports the following models:
  - Graph (compatible with Gremlin and OrientDB SQL)
  - Document (compatible with the MongoDB driver and MongoDB queries)
  - Key/Value (compatible with the Redis driver)
  - Search Engine (LSM-Tree Full-Text Index)
  - Time Series (under construction)
  - Vector (based on the HNSW index)
- ArcadeDB understands multiple languages:
  - SQL (inspired from OrientDB SQL dialect that supports pattern matching on graphs)
  - Cypher
  - Apache Gremlin (Apache Tinkerpop)
  - MongoDB Query Language
- ArcadeDB can be used as:
  - Embedded from any language on top of the Java Virtual Machine
  - Remotely by using HTTP/JSON
  - Remotely by using a Postgres driver (ArcadeDB implements Postgres Wire protocol)
  - Remotely by using a MongoDB driver (only a subset of the operations are implemented)
  - Remotely by using a Redis driver (only a subset of the operations are implemented)
- Getting started with ArcadeDB:
  - Tutorials: Java Tutorial
  - Tools: Working with the Console
  - Containers: Docker, Kubernetes
  - Migrating: from OrientDB
- ArcadeDB Links:
  - Project Website: https://arcadedb.com
  - Source Repository: https://github.com/ArcadeData/arcadedb
  - Latest Release: 24.3.1

Chapter 2. Introduction

2.1. What is ArcadeDB?

Edit this section

ArcadeDB is the new generation of DBMS (DataBase Management System) that runs pretty much on every hardware/software configuration. ArcadeDB is multi-model, which means it can work with graphs, documents and other models of data.

How can it be so fast?

ArcadeDB is written in LLJ ("Low-Level-Java"), that means it's written in Java (Java8+), but without using a high-level API. The result is that ArcadeDB does not allocate many objects at run-time on the heap, so the garbage collection does not need to act regularly, only rarely. At the same time, it is highly portable and leverages the hyper optimized Java Virtual Machine*. Furthermore, the kernel is built to be efficient on multi-core CPUs by using novel mechanical sympathy techniques.

ArcadeDB is a native graph database:

- No more "Joins": relationships are physical links to records
- Traverses parts of, or entire trees and graphs of records in milliseconds
- Traversing speed is independent from the database size

Cloud DBMS

ArcadeDB was born on the cloud. Even though you can run ArcadeDB as embedded and in an on-premise setup, you can spin an ArcadeDB server/cluster in a few seconds with Docker, Kubernetes, Amazon AWS (coming soon), or Microsoft Azure (coming soon).

Is ArcadeDB FREE?

ArcadeDB Community Edition is really FREE for any purpose and thus released under the Apache 2.0 license. We love to know about your project with ArcadeDB and any contributions back to ArcadeDB's open community (reports, patches, test cases, documentations, etc) are welcome.

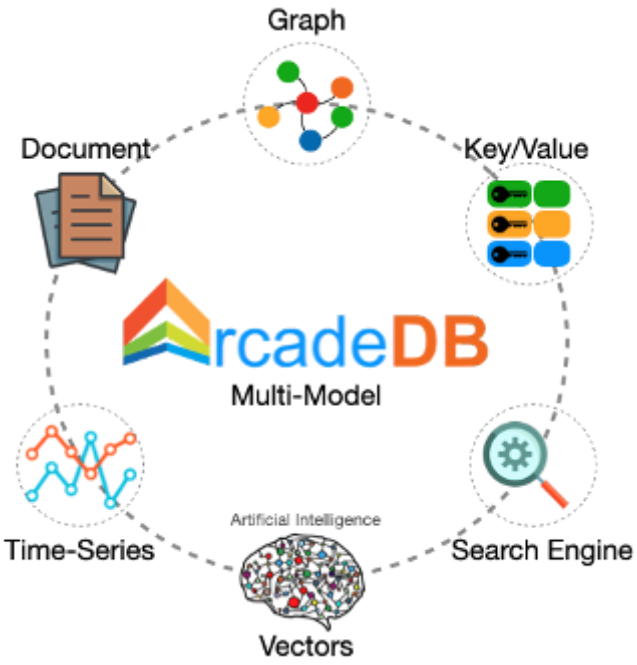
Ask yourself: which is more likely to have better quality? A DBMS created and tested by a handful of developers in isolation, or one tested by thousands of developers globally? When code is public, everyone can scrutinize, test, report and resolve issues. All things open source moves faster compared to the proprietary world.

2.2. Multi Model

Edit this section

The ArcadeDB engine supports **Graph**, **Document**, **Key/Value**, **Search-Engine**, **Time-Series** (), and

Vector-Embedding (□) models, so you can use ArcadeDB as a replacement for a product in any of these categories. However, the main reason why users choose ArcadeDB is because of its true **Multi-Model** DBMS ability, which combines all the features of the above models into one core. This is not just interfaces to the database engine, but rather the engine itself was built to support all models. This is also the main difference to other multi-model DBMSs, as they implement an additional layer with an API, which mimics additional models. However, under the hood, they're truly only one model, therefore they are limited in speed and scalability.



2.2.1. 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
- **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. These properties can be defined by users, which can make vertices and edges appear similar to documents. Furthermore, edges are sorted by the reverse order of insertion, meaning the last edge

added is the first when listed, cf. "Last In First Out".

In the table below, you can find a comparison between the graph model, the relational data model, and the ArcadeDB graph model:

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

2.2.2. 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. This provides several benefits, which we will discuss in further sections of the documentation.

ArcadeDB's document model also adds the concept of a "Relationship" between documents. With ArcadeDB, 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 to other document databases, like MongoDB or CouchDB, where the developer must handle any and all relationships between the documents herself.

The table below illustrates the comparison between the relational model, the document model, and the ArcadeDB document model:

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

2.2.3. Key/Value Model

This is the simplest model. Everything in the database can be reached by a key, where the values can be simple and complex types. ArcadeDB supports documents and graph elements as values allowing for a richer model, than what you would normally find in the typical key/value model. The usual Key/Value model provides "buckets" to group key/value pairs in different containers. The most typical use cases of the Key/Value Model are:

- POST the value as payload of the HTTP call → `/<bucket>/<key>`
- GET the value as payload from the HTTP call → `/<bucket>/<key>`
- DELETE the value by Key, by calling the HTTP call → `/<bucket>/<key>`

The table below illustrates the comparison between the relational model, the Key/Value model, and the ArcadeDB Key/Value model:

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

2.2.4. 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. Such a full-text index is created just like any index in ArcadeDB.

2.2.5. Time-Series Model

Coming Soon.

2.2.6. Vector Model

This model uses the hierarchical navigable small world (HNSW) algorithm to index the multi-dimensional vector data. Practically, an extended version of the hnswlib is used. Since the HNSW algorithm is based on a graph, the vectors are stored as compressed arrays inside ArcadeDB's vertex type, and the proximities are represented by actual edges.

The vector indexing process is configurable, i.e. the distance function, the number of nearest neighbors during construction (`efConstruction`) or search (`ef`), as well as others can be set, see Additional Settings.

Java Example

```
HnswVectorIndexRAM<String, float[], Word, Float> hnswIndex = HnswVectorIndexRAM
.newBuilder(300, DistanceFunctions.FLOAT_INNER_PRODUCT, words.size())
          .withM(16).withEf(200).withEfConstruction(200).build();
```

```
persistentIndex = hnswIndex.createPersistentIndex(database)//
.withVertexType("Word").withEdgeType("Proximity").withVectorPropertyName("vector").wit
hIdProperty("name").create();
```

```
persistentIndex.save();
```

```
persistentIndex = (HnswVectorIndex) database.getSchema().getIndexByName
("Word[name,vector]");
```

```
List<SearchResult<Vertex, Float>> approximateResults = persistentIndex.findNeighbors
(input, k);
```

SQL Example

```
import database https://dl.fbaipublicfiles.com/fasttext/vectors-crawl/cc.en.300.vec.gz
  with distanceFunction = 'cosine', m = 16, ef = 128, efConstruction = 128;
```

In this case the default vertex type used for storing vectors is `Node`.

```
SELECT vectorNeighbors('Node[name,vector]','king',3);
```

Distance Functions (Similarity Measures)

Measure	Name	Type
`cosine`	Cosine Similarity	L_2
`innerproduct`	Inner Product	L_2
`euclidean`	Euclidean Distance	L_2
`correlation`	Correlation Distance	L_2
`manhattan`	Manhattan Distance	L_1
`canberra`	Canberra Distance	L_1
`chebyshev`	Chebyshev Distance	L_∞
`braycurtis`	Bray-Curtis Similarity	/

2.3. Run ArcadeDB

Edit this section

You can run ArcadeDB in the following ways:

- In the cloud (coming soon), by running an ArcadeDB instance on Amazon AWS, Microsoft Azure, or Google Cloud Engine.
- 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 2 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 3 servers, and a load balancer in front. Run ArcadeDB with Kubernetes to have an automatic setup of servers in HA with a load balancer upfront.

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.

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 driver chapter for more information.

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.

Binaries

	Linux / Mac	Windows
Server	`bin/server.sh`	`bin\server.bat`
Console	`bin/console.sh`	`bin\console.bat`

2.3.1. Getting Started

All Platforms

1. Download latest release from Github
2. Unpack `tar -xzf arcadedb-24.3.1`
    - Change into directory: `cd arcadedb-24.3.1`
3. Launch server
    - Linux / MacOS: `bin/server.sh`
    - Windows: `bin\server.bat`

4. Exit server via **CTRL+C**
5. Interact with server
    - Studio: `http://localhost:2480`
    - Console:
        - Linux / MacOS: `bin/console.sh`
        - Windows: `bin\console.bat`

Windows via Scoop

Instead of using manual install you can use Scoop installer, instructions are available on the 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.

Chapter 3. Main Concepts

Edit this section

3.1. Record

A record is the smallest unit you can load from and store in the database. Records come in three types:

- Document
- Vertex
- Edge

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."
  }]
}
```

Vertex

In Graph databases the vertices (also named vertexes), 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.

Edge

An edge, or arc, is the connection between two vertices. Edges can be unidirectional and bidirectional. One edge can only connect two vertices.

For more information on connecting vertices in general, see Relationships below.

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 sign 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.

`#-1:-1` is a null RID.

The prefix character `#` is mandatory.

Each Record ID is immutable, universal, and is never reused. 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);
```

3.2. Types

The concept of type is taken from the Object Oriented Programming paradigm, sometimes 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 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.

As a default, ArcadeDB creates as many buckets per type as many cores (processors) the host machine has. 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`.

3.3. 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`. ArcadeDB creates additional buckets for each type, (one for each CPU core on the server), to improve performance of parallelism.

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.

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.

```
arcadeDB> 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.

```
arcadeDB> 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.

```
arcadeDB> 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.

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.

```
arcadeDB> 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.

When you're looking for customers based in Europe, you could execute this query:

```
arcadeDB> 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.

```
arcadeDB> 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.

```
arcadeDB> SELECT FROM BUCKET:[Customer_Europe_Italy,Customer_Europe_Spain]
```

3.4. Relationships

ArcadeDB supports three kinds of relationships: **connections**, **referenced** and **embedded**. It can manage relationships in a schema-full or schema-less scenario.

Graph Connections

As a graph database, spanning edges between vertices is one way to express a connections between records. This is 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

```
   Vertex A -------------> Edge X -------------> Vertex B
TYPE=Customer           TYPE=isBilled          TYPE=Invoice
   RID #5:23              RID #16:9              RID #10:2
```

In ArcadeDB's SQL, edges are created via the `CREATE EDGE` command.

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. This boosts the load speed for the entire graph of connected objects, such as in graph and object database systems.

Example

```
   Record A -------------> Record B
TYPE=Customer            TYPE=Invoice
```

```
    RID #5:23              RID #10:2
```

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.

Embedded Relationships

When using Embedded relationships, ArcadeDB stores the relationship within the record that embeds it. These relationships are stronger than Reference relationships. You can represent it as a UML Composition relationship.

Embedded records do not have their own RID, given that you can't directly reference it through other records. It is only accessible through the container record.

In the event that you delete the container record, the embedded record is also deleted. For example,

```
    Record A <>----------> Record B
   TYPE=Account          TYPE=Address
    RID #5:23               NO RID
```

Here, record `A` contains the entirety of record `B` in the property `address`. You can reach record `B` only by traversing the container record. For example,

```
arcadeDB> SELECT FROM Account WHERE address.city = 'Rome'
```

1:1 and n:1 Embedded Relationships

ArcadeDB expresses relationships of these kinds using the `EMBEDDED` type.

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.

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.

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;
```

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.

3.5. Database

Each server or Java VM can handle multiple database instances, but the database name must be unique.

Database URL

ArcadeDB uses its own URL format, of engine and database name as `<engine>:<db-name>`. The embedded engine is the default and can be omitted. To open a database on the local file system you can use directly the path as URL.

Database Usage

You must always close the database once you finish working on it.

> ArcadeDB automatically closes all opened databases, when the process dies gracefully (not by killing it by force). This is assured if the operating system allows a graceful shutdown. For example, on Unix/Linux systems using `SIGTERM`, or in Docker `exit code 143` instead of `SIGKILL`, or in Docker `exit code 137`.

3.6. Transactions

Edit this section

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 on client 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.

ACID Properties

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

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
8	commit		10 → 11 = Error, in database x already is at 11

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.

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.

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:

- HW problems, such as loss of power or disk error
- SW problems, such as a Operating System crash
- Application problem, 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 using an ArcadeDB Server connected remotely, if your application crashes the engine continue to work, but 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.

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 on client'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.

Nested transactions and propagation

ArcadeDB does support nested transaction. If further `begin()` are called after a transaction is already begun, then the new transaction is the current one until commit or rollback. When the nested transaction is completed, the previous transaction becomes the current transaction.

3.7. Inheritance

Edit this section

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);
```

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.

```
Account(10) <|--- Company (13) <|--- OrientTechnologiesGroup (27)
```

By default, ArcadeDB creates a separate bucket for each type. It indicates this bucket by the `defaultBucketId` property in the type `OType` and indicates the bucket used by default when not specified. However, the type `OType` has a property `bucketIds`, (as `int[]`), that contains all the buckets able to contain the records of that type. `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.

3.8. Indexes

Edit this section

ArcadeDB indexes are built by using the LSM Tree algorithm.

3.8.1. LSM Tree algorithm

LSM tree is a type of data structure that is used to store and retrieve data efficiently. It works by organizing data in a tree-like structure, where each node in the tree represents a certain range of

data.

Here’s how it works:

1. When you want to store a piece of data in the LSM tree, it first goes into a special part of the tree called a "write buffer." The write buffer is like a temporary storage area where new data is kept until it’s ready to be added to the tree.
2. When the write buffer gets full, the LSM tree will "flush" the data from the write buffer into the main part of the tree. This is done by creating a new node in the tree and adding the data from the write buffer to it.
3. As more and more data is added to the tree, it will eventually become too large to be stored in memory (this is known as "overflowing"). When this happens, the LSM tree will start to "compact" the data by moving some of it to disk storage. This allows the tree to continue growing without running out of memory.
4. When you want to retrieve a piece of data from the LSM tree, the algorithm will search for it in the write buffer, the main part of the tree, and any data that has been compacted to disk storage. If the data is found, it will be returned to you

3.8.2. LSM Tree vs B+Tree

B+Tree is the most common algorithm used by relational DBMSs. What are the differences?

1. LSM tree and B+ tree are both data structures that are commonly used to store and retrieve data efficiently. Here are some of the main advantages of LSM tree over B+ tree:
2. LSM tree is more efficient for writes: LSM tree uses a write buffer to temporarily store new data, which allows it to batch writes and reduce the number of disk accesses required. This can make it faster than B+ tree for inserting large amounts of data.
3. LSM tree is more efficient for compaction: Because LSM tree stores data in a sorted fashion, it can compact data more efficiently by simply merging sorted data sets. B+ tree, on the other hand, requires more complex rebalancing operations when compacting data.
4. LSM tree is more space-efficient: LSM tree stores data in a compact, sorted format, which can make it more space-efficient than B+ tree. This can be especially useful when storing large amounts of data on disk.
5. However, there are also some potential disadvantages of LSM tree compared to B+ tree. For example, B+ tree may be faster for queries that require range scans or random access, and it may be easier to implement in some cases.

If you’re interested to ArcadeDB’s LSM-Tree index implementation detail, look at LSM-Tree

Chapter 4. Server

4.1. Server

Edit this section

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

INFO  [ArcadeDBServer] <ArcadeDB_0> ArcadeDB Server v23.12.2 (build
1a7716d309bbcbfdac089d42adee56b38fe2f6d3/1705309274480/main) is starting up...
INFO  [ArcadeDBServer] <ArcadeDB_0> Running on Mac OS X 12.7 - OpenJDK 64-Bit Server
VM 17.0.9 (Homebrew)
INFO  [ArcadeDBServer] <ArcadeDB_0> Starting ArcadeDB Server in development mode with
plugins [] ...
INFO  [ArcadeDBServer] <ArcadeDB_0> - Metrics Collection Started...

+-------------------------------------------------------------------+
|                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/security.json`. 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 rule of security are pretty basic. The password length must be between 8 and 256 characters. You can implement your own security policy. Check 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
```

> 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):
***********

INFO  [HttpServer] <ArcadeDB_0> - Starting HTTP Server (host=0.0.0.0 port=2480-2489
httpsPort=2490-2499)...
INFO  [undertow] starting server: Undertow - 2.3.10.Final
INFO  [xnio] XNIO version 3.8.8.Final
INFO  [nio] XNIO NIO Implementation Version 3.8.8.Final
INFO  [threads] JBoss Threads version 3.5.0.Final
INFO  [HttpServer] <ArcadeDB_0> - HTTP Server started (host=0.0.0.0 port=2480
httpsPort=2490)
INFO  [ArcadeDBServer] <ArcadeDB_0> Available query languages: [sqlscript, mongo,
gremlin, java, cypher, js, graphql, sql]
INFO  [ArcadeDBServer] <ArcadeDB_0> ArcadeDB Server started in 'development' mode
(CPUs=8 MAXRAM=4,00GB)
INFO  [ArcadeDBServer] <ArcadeDB_0> Studio web tool available at
http://192.168.1.113:2480
```

By default, the following components start with the server:

- JMX Metrics, to monitor server performance and statistics (served via port 9999).
- HTTP Server, that listens on port 2480 by default. If port 2480 is already occupied, then the next is taken up to 2489.

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 HA configuration, it's mandatory all the servers in cluster have different names.

4.1.1. 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
```

4.1.2. Server modes

The server can be started in one of three modes, which affect the studio and logging:

Mode	Studio	Logging
`development`	Yes	Detailed
`test`	Yes	Brief
`production`	No	Brief

The mode is controlled by the `server.mode` setting with a default mode `development`.

4.1.3. 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[elon:musk]"
```

With the example above the database "Universe" will be created if doesn't exist, with user "elon", password "musk".

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[]`

Once the server is started, multiple clients can be connected to the server by using one of the supported protocols:

- HTTP/JSON
- Postgres Driver
- MongoDB Driver
- Redis Driver

4.1.4. 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 (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 (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 (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 is 100MB
- Line 10 sets the number of files to keep in the directory. By default 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.

4.1.5. 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) {
  }
}
```

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. Instead, 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, wil be invoked when the HTTP server is initializing where you can register your own HTTP commands. 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 the comma to separate them.

4.2. Changing Settings

Edit this section

To change the default value of a setting, always put `arcadedb.` as a prefix. Example:

```
~/arcadedb $ 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.

4.2.1. Environment Variables

The server script parses a set of environment variables which are summarized below:

`JAVA_HOME`	JVM location
`JAVA_OPTS`	JVM options
`ARCADEDB_HOME`	ArcadeDB location
`ARCADEDB_PID`	`arcadedb.pid` file location

`ARCADEDB_OPTS_MEMORY`	JVM memory options

For default values see the `server.sh` and `server.bat` scripts.

4.2.2. RAM Configuration

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 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 16 MB 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.

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.

4.3. High Availability

Edit this section

ArcadeDB supports a High Availability mode where multiple servers share the same database (replication).

In order to start an ArcadeDB server with the support for replication, you need to:

1. Enable the HA module by setting the configuration `arcadedb.ha.enabled` to `true`
2. Define the servers that are part of the clusters (if you are using Kubernetes, look at Kubernetes paragraph). To setup the server list, define the `arcadedb.ha.serverList` setting by separating the servers with commas (`,`) and using the following format for servers: `<hostname/ip-address[:port]>`. The default port is `2424` if not specified.
3. Define the local server name by setting the `arcadedb.server.name` configuration. Each node must have a unique name. If not specified, the default server name is "ArcadeDB_0"

Example:

```
$ bin/server.sh -Darcadedb.ha.enabled=true
                -Darcadedb.server.name=FloridaServer
                -Darcadedb.ha.serverList=192.168.0.2,192.168.0.1:2424,japan-
server:8888
```

The log should look like this:

```
[HttpServer] <FloridaServer> - HTTP Server started (host=0.0.0.0 port=2480)
[LeaderNetworkListener] <ArcadeDB_0> Listening for replication connections on
127.0.0.1:2424 (protocol v.-1)
[HAServer] <FloridaServer> Unable to find any Leader, start election (cluster=arcadedb
configuredServers=1 majorityOfVotes=1)
[HAServer] <FloridaServer> Change election status from DONE to VOTING_FOR_ME
[HAServer] <FloridaServer> Starting election of local server asking for votes from []
(turn=1 retry=0 lastReplicationMessage=-1 configuredServers=1 majorityOfVotes=1)
[HAServer] <FloridaServer> Current server elected as new Leader (turn=1 totalVotes=1
majority=1)
[HAServer] <FloridaServer> Change election status from VOTING_FOR_ME to
LEADER_WAITING_FOR_QUORUM
[HAServer] <FloridaServer> Contacting all the servers for the new leadership (turn=1
)...
```

At startup, the ArcadeDB server will look for an existent cluster to join based on the configured list of servers, otherwise a new cluster will be created. In this example we set the server name to `FloridaServer`.

Every time a server joins a cluster, it starts the process to elect the new leader. If the cluster exists and already contains a Leader, then the existent leader is kept. Every time a server leaves the cluster (because it becomes unreachable), the election process is started again. To know more about the RAFT election process, look at RAFT protocol.

> The cluster name by default is "arcadedb", but you can have multiple clusters in the same network. To specify a custom name, set the configuration `arcadedb.ha.clusterName=<name>`. Example: `bin/server.sh -Darcadedb.ha.clusterName=projectB`

All servers of a cluster serve the same databases.

4.3.1. Architecture

ArcadeDB has a Leader/Replica model by using RAFT consensus for election and replication.

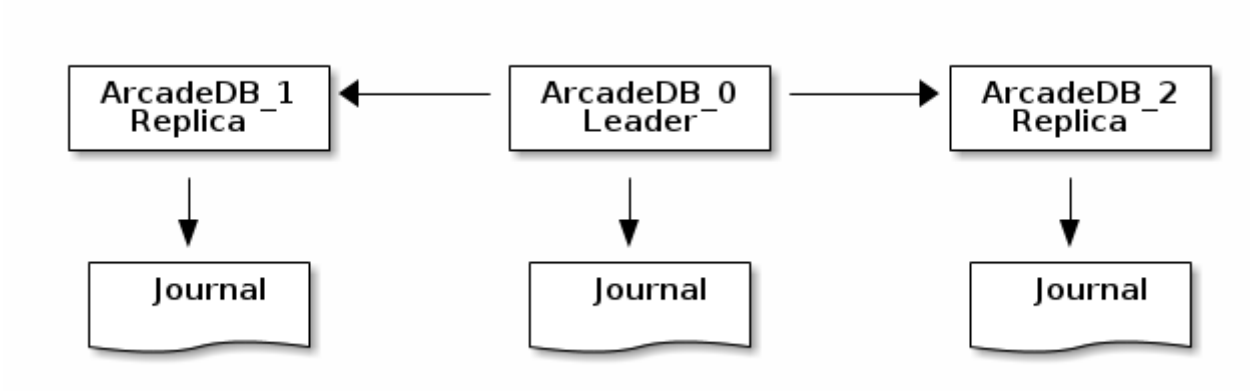


Figure 1. Cluster of Servers

Each server has its own Journal. The Journal is used in case of recovery of the cluster to get the most updated replica and to align the other nodes. All the write operations must be coordinated by the Leader node.

Any read operation, like a query, can be executed by any server in the cluster, while write operations can be executed only by the Leader server.

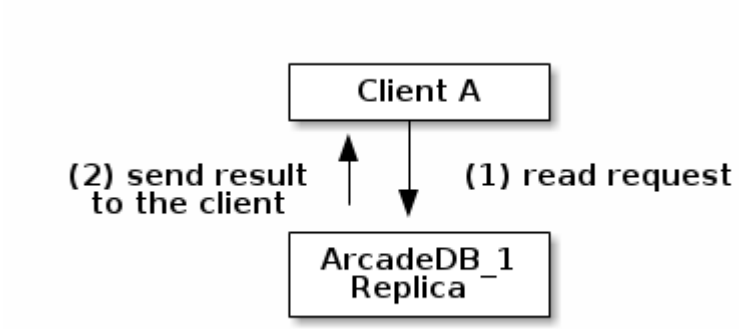


Figure 2. Read Request executed on a Replica

ArcadeDB doesn't mandate the clients to be connected directly to the leader to execute write operations, but will use the Replica server to forward the write request to the Leader server. Everything is transparent for the end user where both Leader and Replica servers can read and write, but internally only the read requests are executed on the connected server. All the write requests will be transparently forwarded to the Leader.

Look at the picture below where the client `Client A` is connected to the replica server `ArcadeDB_1` where it executes a write request.

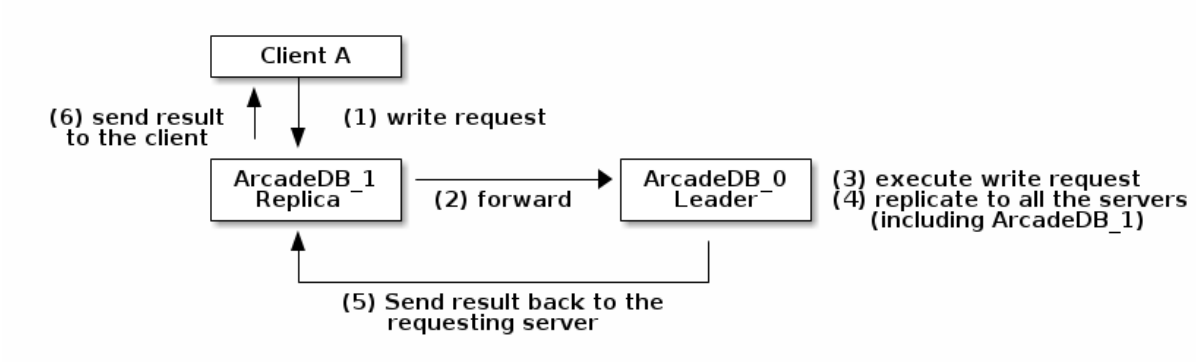


Figure 3. Write Request executed on a Replica

4.3.2. Auto fail-over

ArcadeDB cluster uses a quorum to assure the integrity of the database is maintained across all the servers forming the cluster. The quorum is set by default to `MAJORITY`, that means the majority of the servers in the cluster must return the same result to be considered accepted and propagated to all the servers.

> The quorum is `MAJORITY` by default. You can specify a different quorum by setting the number of servers or `none` to have no quorum and `all` to wait the response from all the servers. Set the configuration `arcadedb.ha.quorum=<quorum>`. Example: `bin/server.sh -Darcadedb.ha.quorum=all`

If the configured quorum is not met, the transaction is rollback on all the servers, the database returns to the previous state and a transaction error is thrown to the client.

ArcadeDB manages the fail-over automatically in most of the cases.

Server unreachable

A server can become unreachable for many reasons:

- The ArcadeDB Server process has been terminated
- The physical or virtual server hosting the ArcadeDB Server process has been shut off or is rebooting
- The physical or virtual server hosting the ArcadeDB Server process has network issues and can't reach one or more of the other servers
- Network issues that prevent the ArcadeDB Server to communicate with the rest of the servers in the cluster

4.3.3. Auto balancing clients

More coming soon.

4.3.4. Troubleshooting

Performance: insertion is slow

ArcadeDB uses an optimistic lock approach: if two threads try to update the same page, the first thread wins, the second thread throws a `ConcurrentModificationException` and forces the client to retry the transaction or fail after a certain number of retries (configurable). Often this fail/retry mechanism is totally hidden to the developer that executes a transaction via HTTP or via the Java API:

```
db.transaction( ()-> {
  // MY TRANSACTION CODE
});
```

If you are inserting a lot of record in parallel (by using the Server, or just via API multi-thread), you could benefit by allocating the bucket per thread. Example to change the bucket selection strategy for the vertex type "User":

```
alter type User BucketSelectionStrategy `thread`
```

With the command above, in insertion ArcadeDB will select the physical bucket based on the thread the request is coming from. If you have enough buckets (created by default when you create a new type, but you can manually adjust it) insertions can go truly in parallel with zero contentions in pages, meaning zero exception and retries.

4.3.5. HA Settings

The following settings are used by the High Availability module:

Setting	Description	Default Value
arcadedb.ha.clusterName	Cluster name. Useful in case of multiple clusters in the same network	arcadedb
arcadedb.ha.serverList	Servers in the cluster as a list of <hostname/ip-address:port> items separated by comma. Example: 192.168.0.1:2424,192.168.0.2:2424. If not specified, auto-discovery is enabled	NOT DEFINED (auto discovery is enabled by default)
arcadedb.ha.serverRole	Enforces a role in a cluster, either "any" or "replica"	"any"
arcadedb.ha.quorum	Default quorum between 'none', 1, 2, 3, 'majority' and 'all' servers	MAJORITY

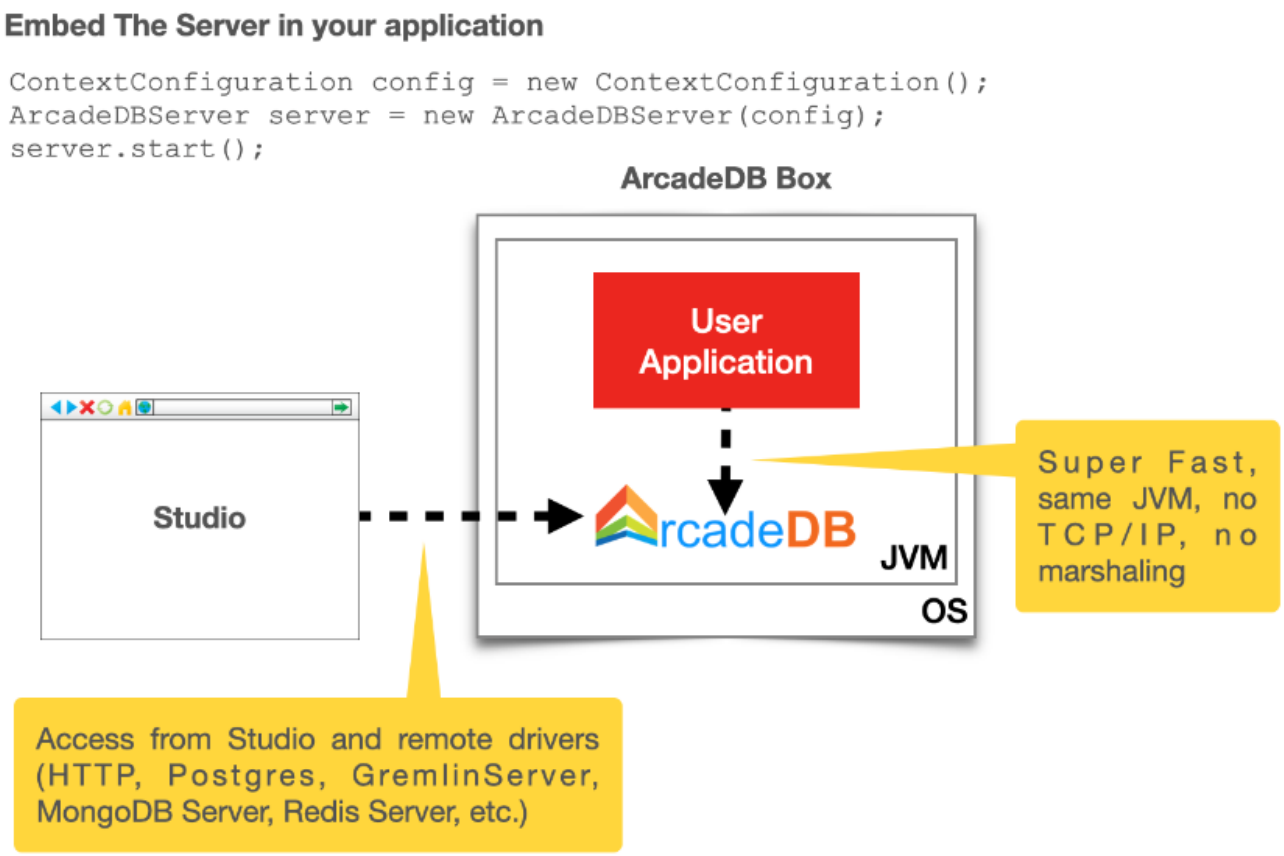
Setting	Description	Default Value
arcadedb.ha.quorumTimeout	Timeout waiting for the quorum	10000
arcadedb.ha.k8s	The server is running inside Kubernetes	false
arcadedb.ha.k8sSuffix	When running inside Kubernetes use this suffix to reach the other servers. Example: arcadedb.default.svc.cluster.local	
arcadedb.ha.replicationQueueSize	Queue size for replicating messages between servers	512
arcadedb.ha.replicationFileMaxSize	Maximum file size for replicating messages between servers"	1GB
arcadedb.ha.replicationChunkMaxSize	Maximum channel chunk size for replicating messages between servers	16777216
arcadedb.ha.replicationIncomingHost	TCP/IP host name used for incoming replication connections	localhost
arcadedb.ha.replicationIncomingPorts	TCP/IP port number used for incoming replication connections	2424-2433

4.4. Embedded Server

Edit this section

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>24.3.1</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.

4.4.1. 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);
```

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.getDatabase(<URL>);
```

Or this if you want to create a new database if not exists:

```
Database database = server.getOrCreateDatabase(<URL>);
```

4.4.2. 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 the Test handler that will called by using 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 ...
```

4.4.3. 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 under the new global setting:

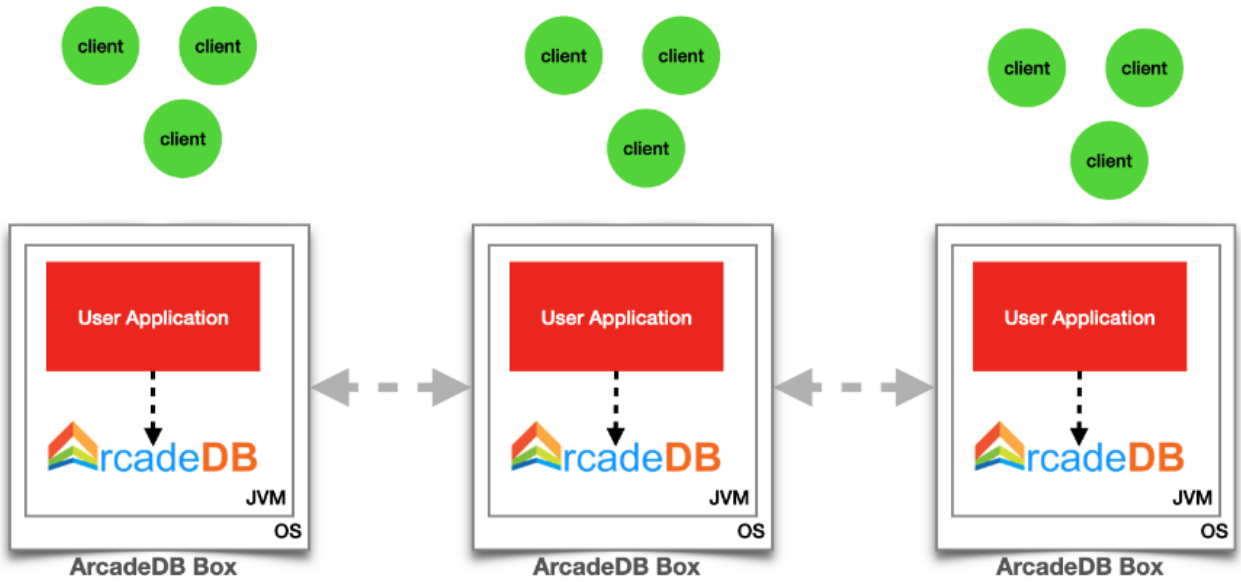
```
GlobalConfiguration.SERVER_HTTPS_INCOMING_PORT
```

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

4.4.4. Replication between boxes

You can replicate databases acros multiple boxes to have a true high availability.



Best of latency between User Application and Database Server + Replication for High Availability

4.5. Docker

Edit this section

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

```
~/arcadedb $ docker run --rm -p 2480:2480 -p 2424:2424 --name my_arcadedb \
             --env JAVA_OPTS="-Darcadedb.server.rootPassword=playwithdata" \
             --hostname my_arcadedb arcadedata/arcadedb:latest
```

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.

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

```
~/arcadedb $ docker exec -it my_arcadedb bin/console.sh
ArcadeDB Console v.23.9.1 - Copyrights (c) 2023 Arcade Data (https://arcadedb.com)

>
```

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

Quick start with the OpenBeer database

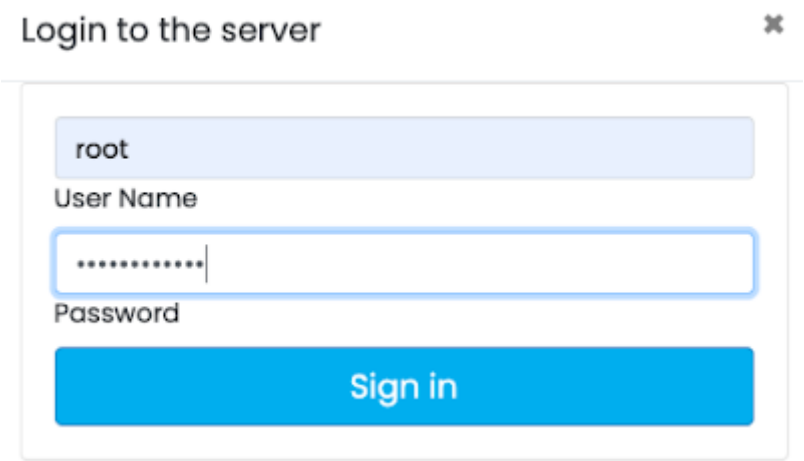
You can run ArcadeDB server with a demo database in less than 1 minute. Run ArcadeDB server with docker specifying the database to import as a parameter in the docker command.

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

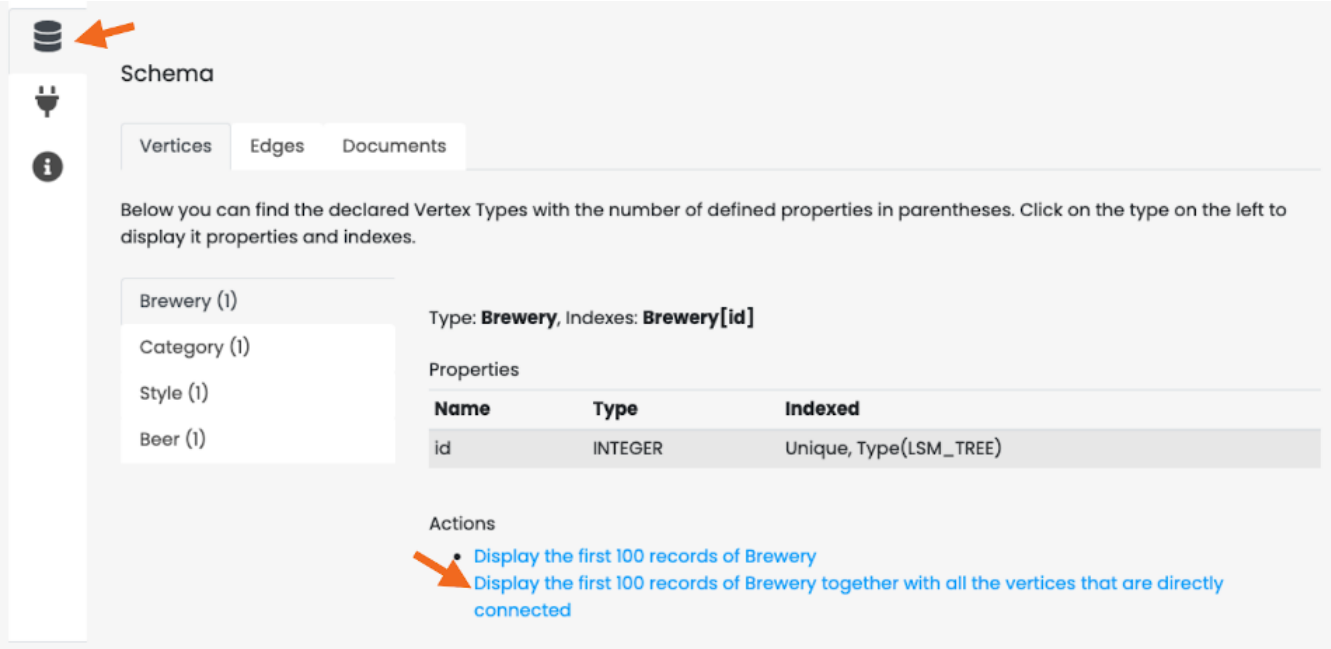
```
docker run --rm  -p 2480:2480 -p 2424:2424 -p 6379:6379 -p 5432:5432 -p 8182:8182 \
           --env JAVA_OPTS="-Darcadedb.server.rootPassword=playwithdata \
-Darcadedb.server.defaultDatabases=Imported[root]{import:https://github.com/ArcadeData
/arcadedb-datasets/raw/main/orientdb/OpenBeer.gz} \
-Darcadedb.server.plugins=Redis:com.arcadedb.redis.RedisProtocolPlugin,MongoDB:com.arc
adedb.mongo.MongoDBProtocolPlugin,Postgres:com.arcadedb.postgres.PostgresProtocolPlugi
n,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.

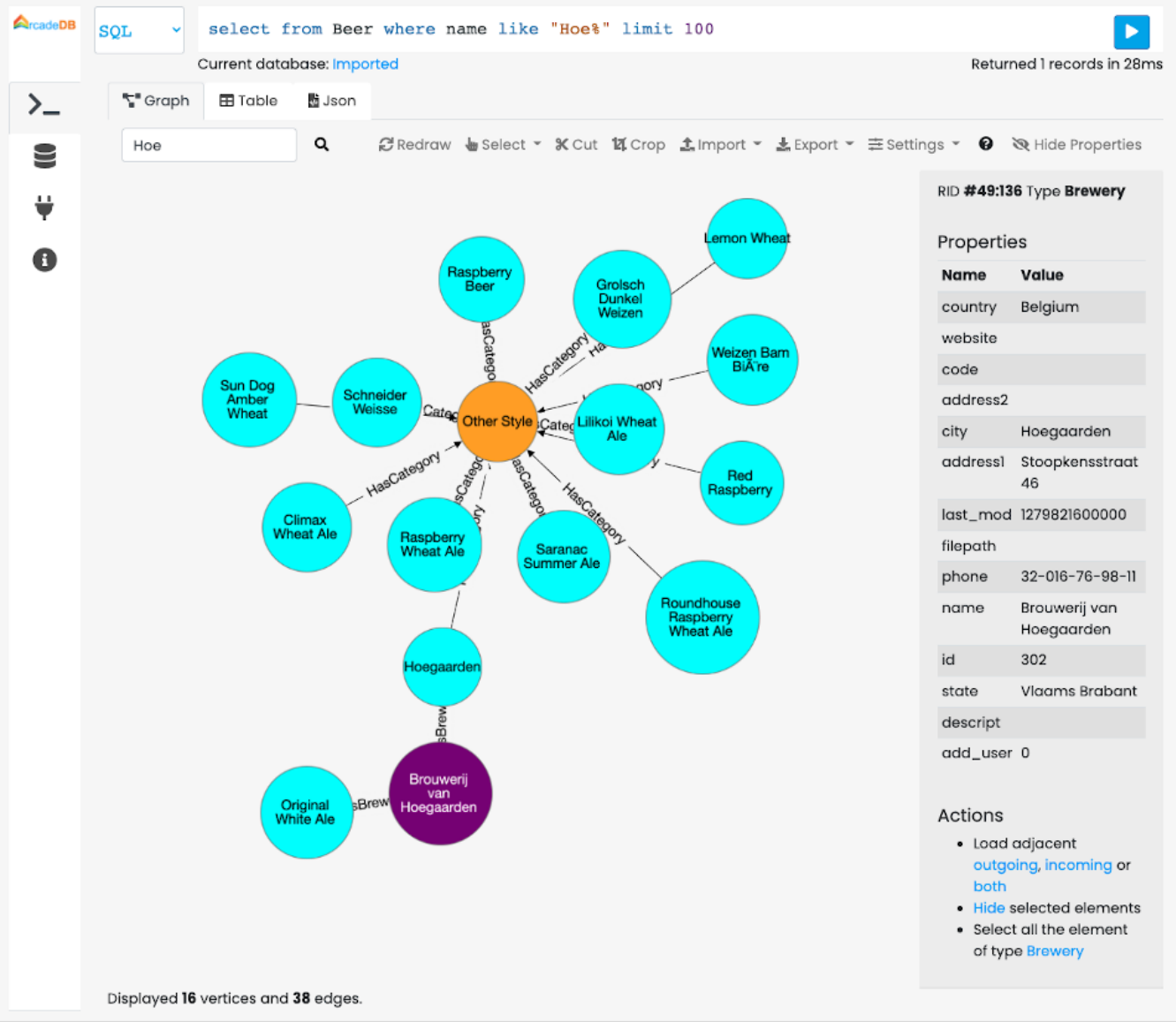
user and password are specified in the docker command above



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

Persistence

There are multiple options to make databases in an ArcadeDB container persistent, yet the practical choice is highly dependent on the application and environment.

On the Docker side there are two typical options to persist data in a container:

1. A Docker volume which is a Docker managed section of the host's filesystem. This would be the preferred option if the host machine is to store the persistent data.
2. A Bind mount which is a user-managed folder of the host's filesystem. This can be used if the target volume is itself mounted on the host like a network share.

See the Docker storage documentation on how to set up volumes or mounts.

On the ArcadeDB side there are basically two options to consider:

1. Store the database itself on an external volume. In this case all read and write operations from and to a database take place on the specified volume or mount, which can entail performance limitations. Use the setting `-Darcadedb.server.databaseDirectory=/mydatabases` to specify the

mount-point of the volume or mount as absolute path to ArcadeDB.

2. Store only backups on an external volume. In this approach all read and write operation from and to a database happen in the storage provided by Docker to the container. This is faster if the host can provide local storage. However, the user needs to take care of regular backups, see `BACKUP DATABASE`, for which the default backup folder can be set by `-Darcadedb.server.backupDirectory=/mybackup` to specify the mount-point of the volume or mount as absolute path to ArcadeDB.

> Which combination is most suitable depends on the computational environment. For example, "local storage" can have different meanings on a bare metal machine, a virtual machine or container cluster.

Tuning

In general, the RAM allocated for the JVM should be ≤80% of the container RAM. The default Dockerfile for ArcadeDB sets 2 GB of RAM for ArcadeDB (`-Xms2G -Xmx2G`), so you should allocate at least 2.3G to the Docker container running exclusively ArcadeDB.

To run ArcadeDB with 1G docker container, you could start ArcadeDB by using 800M for ArcadeDB's server RAM by setting `ARCADEDB_OPTS_MEMORY` variable with Docker:

```
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.profile=low-ram ...
```

4.6. Kubernetes

Edit this section

Before starting the cluster, set ArcadeDB Server root password as a secret (replace <password> with the root password you want to use):

```
~/arcadedb $ kubectl create secret generic arcadedb-credentials --from-literal
=rootPassword='<password>'
```

This will set the password in the `rootPassword` environment variable. The ArcadeDB server will use this password for the `root` user.

Now you can start a Kubernetes cluster with 3 servers by using the default configuration:

```
~/arcadedb $ kubectl apply -f config/arcadedb-statefulset.yaml
```

For more information on ArcadeDB Kubernetes config please check.

In order to scale up or down with the number of replicas, use this:

```
~/arcadedb $ kubectl scale statefulsets arcadedb-server --replicas=<new-number-of-
replicas>
```

Where the value of `<new-number-of-replicas>` is the new number of replicas. Example:

```
~/arcadedb $ kubectl scale statefulsets arcadedb-server --replicas=3
```

Scaling up and down doesn't affect current workload. There are no pauses when a server enters/exits from the cluster.

4.6.1. Troubleshooting

The most common issue with using K8S is with the setting of the root password for the server (see at the beginning of this section).

Follows some useful commands for troubleshooting issues with ArcadeDB and Kubernetes.

Display the status of a pod:

```
~/arcadedb $ kubectl describe pod arcadedb-0
```

Replace "arcadedb-0" with the server container you want to use.

To display the logs of the first server:

```
~/arcadedb $ kubectl logs arcadedb-0
```

Replace "arcadedb-0" with the server container you want to use.

To remove all the containers and restart the cluster again, execute these 2 commands:

```
~/arcadedb $ kubectl delete all --all --namespace default
~/arcadedb $ kubectl apply -f config/arcadedb-statefulset.yaml
```

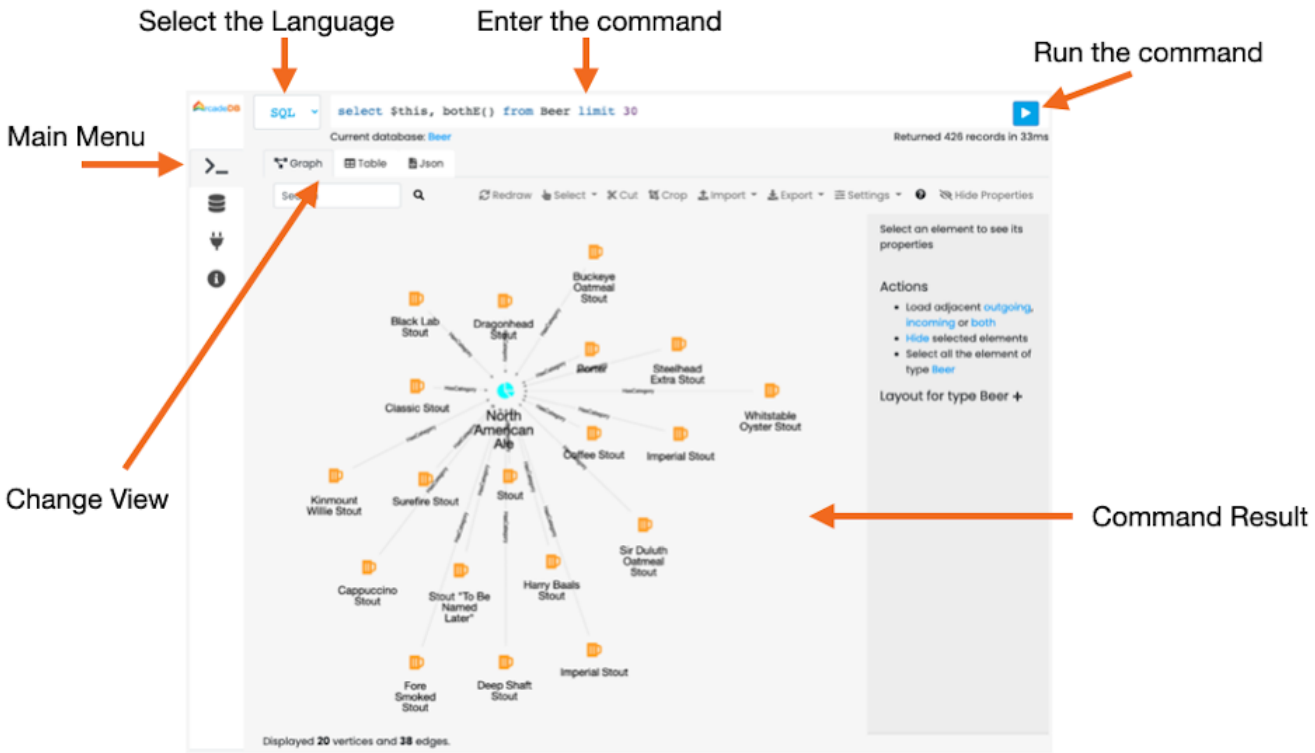
Chapter 5. Studio

Studio is the web tool that comes bundled with ArcadeDB Server. It starts automatically on port 2480. If you have installed ArcadeDB, and it's running on your local computer, then you can access Studio at http://localhost:2480. Replace "localhost" with the host name or IP address of the server where ArcadeDB server is running.

If you're running ArcadeDB server as Embedded, please make sure the jar `arcadedb-studio-<version>.jar` is included in your classpath.

Command

The first and most important panel in Studio is the Command panel. Below you can find a screenshot with the main components.

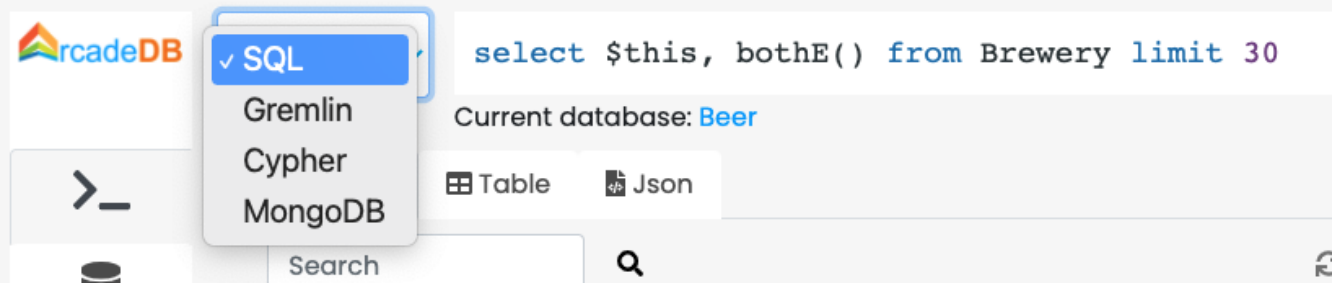


- **Main Menu** is the vertical tab with the following options:
  - **Command**, the current panel to execute commands against the database
  - **Database**, containing the information about the selected database and its schema. From this panel you can switch to a different database
  - **API**, with the description of the public HTTP API exposed on the current server
  - **Information**, containing a quick reference to the online documentation

Execute a command/query

In order to execute a command (or query), select the language first. By default is SQL, but you can choose between:

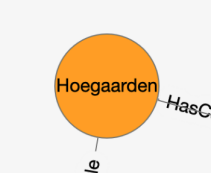
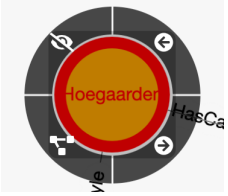
- **SQL** (for any models, including graphs and documents)
- **SQL Script** (multiple commands/queries)
- **Apache Tinkerpop Gremlin** (only for graphs)
- **Open Cypher** (only for graphs)
- **MongoDB** (only for documents)
- **GraphQL**



Based on the selected language, the command text area will adjust the syntax highlighting to simplify the writing of the command.

The result of the command will appear in the Command Result area as a Graph a Table or JSON Panel.

Graph Panel

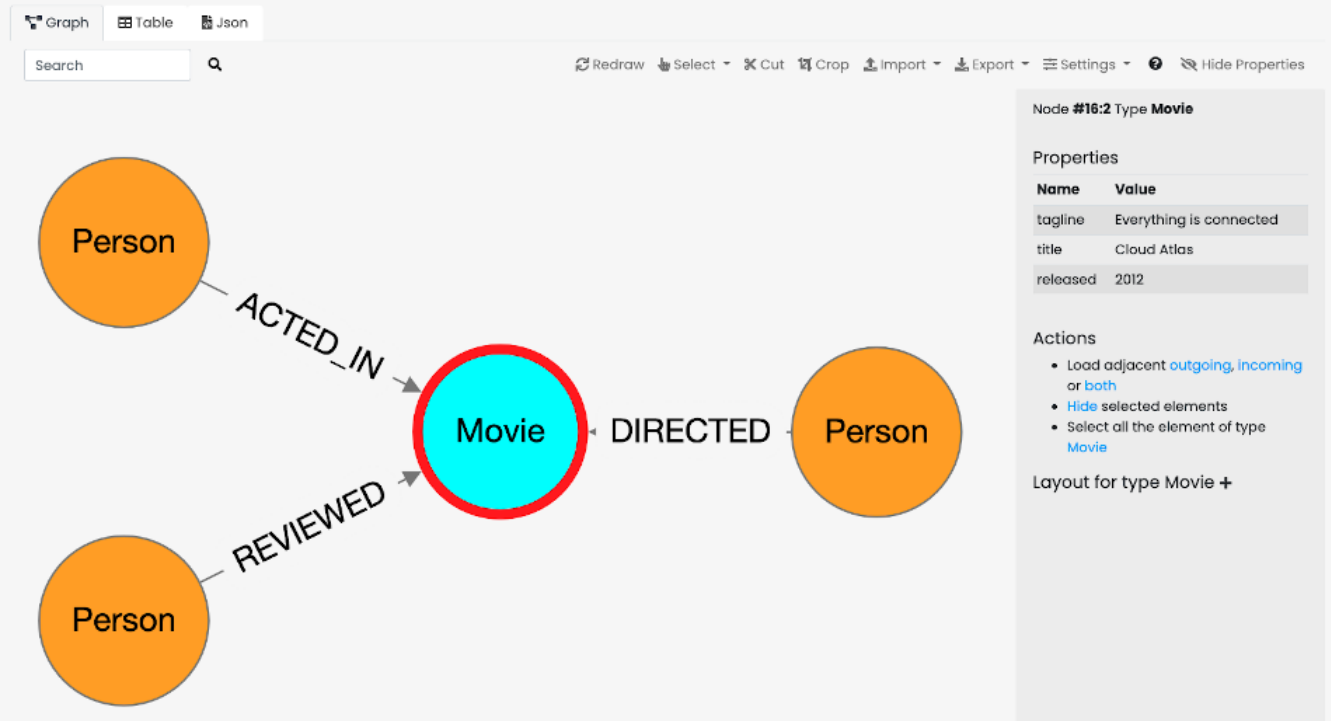
	**Hold the selection on a node to show its context menu. Then while still holding the selection, slide on the action to execute and then release the selection.**	

The context menu has the following actions:

- ← Load incoming vertices
- → Load outgoing vertices
- Load both incoming and outgoing vertices
- Hide the current node. This action will remove the node from the graph

Node Panel

When a node is selected, its property are displayed in the right panel.



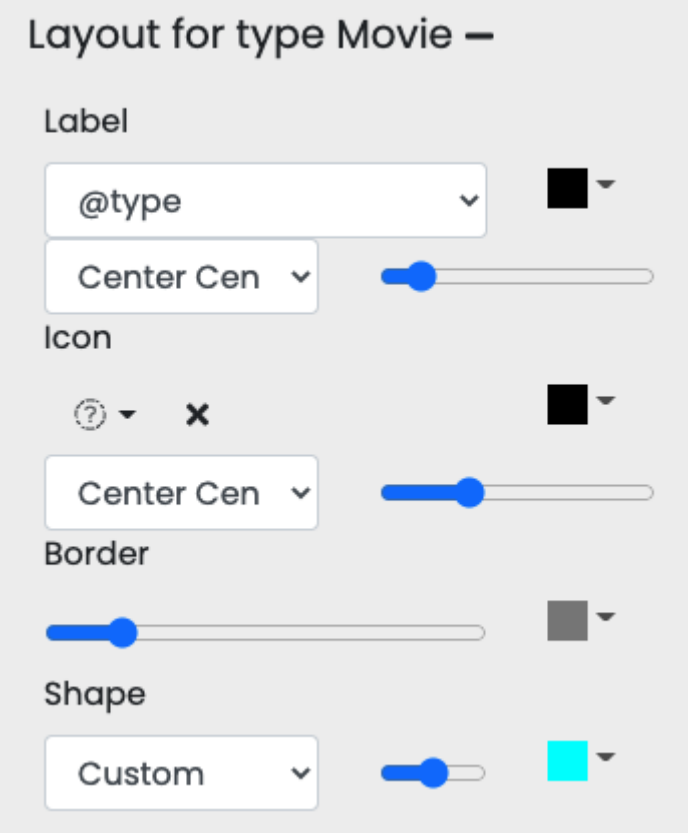
The right panel can always be hidden by clicking on `Hide Properties` button.

In the right panel you can find all the information relative to the selected node, such as:

- Element type: `Node` or `Edge`
- Record ID (RID)
- Type
- Properties table
- Actions, containing quick actions to execute against the selected node
- Layout

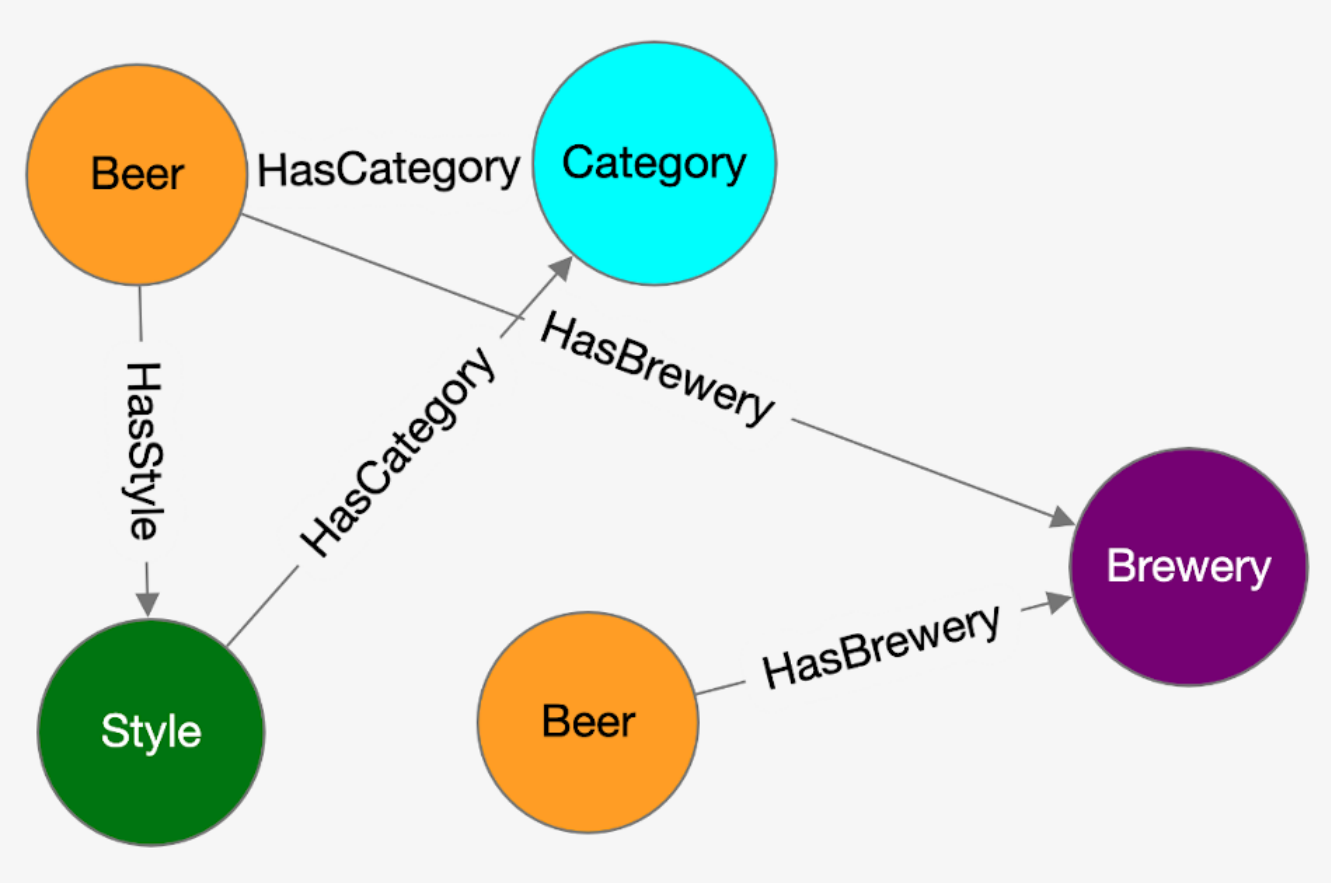
Node Layout

Click on the `+` button to expand and make visible the layout panel relative to the node type selected.

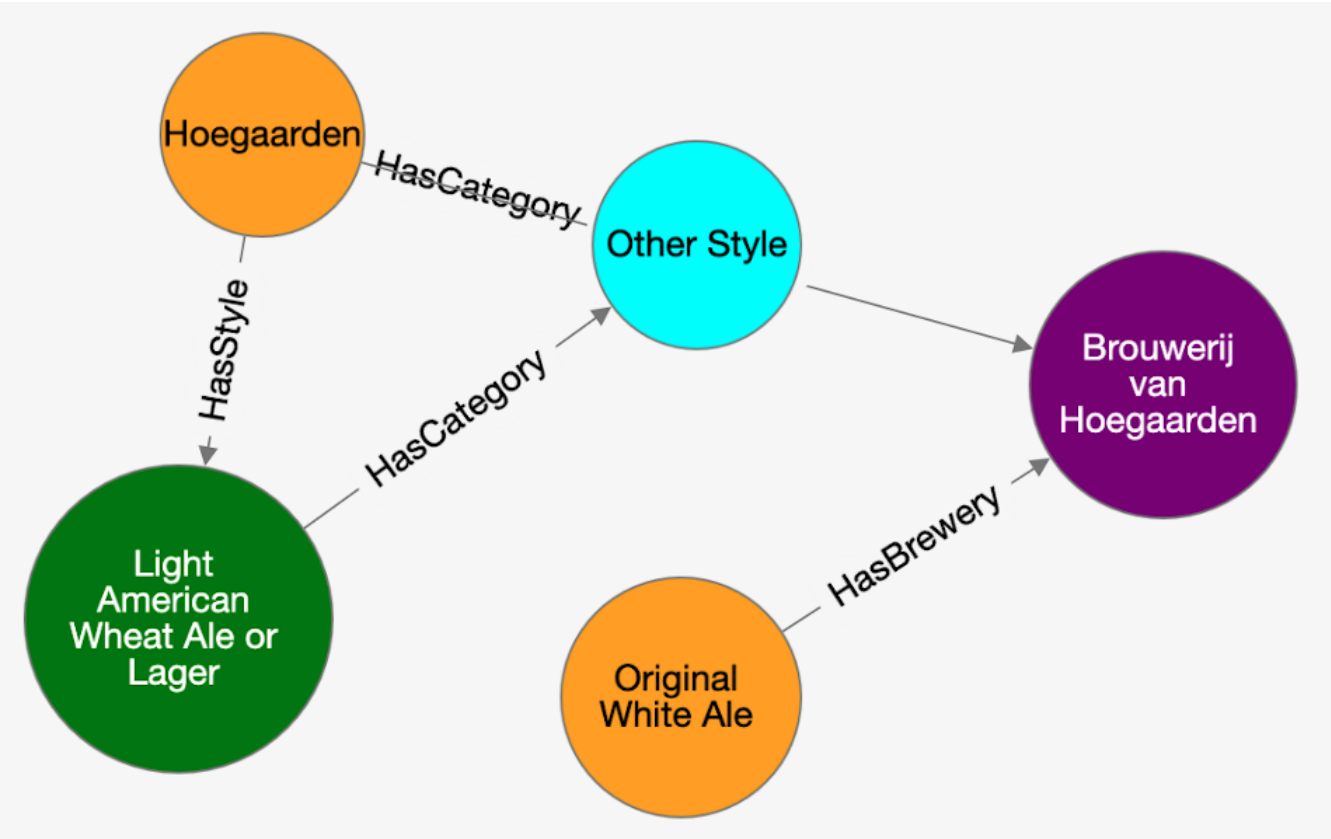


Change the label to an attribute that represents the node. In this example, selecting the title for the type `Movie` and the name for `Person`, makes the same graph much more readable and useful in terms of information.

This is the default rendering of a small graph from the OpenBeer dataset. The nodes have the type as label.



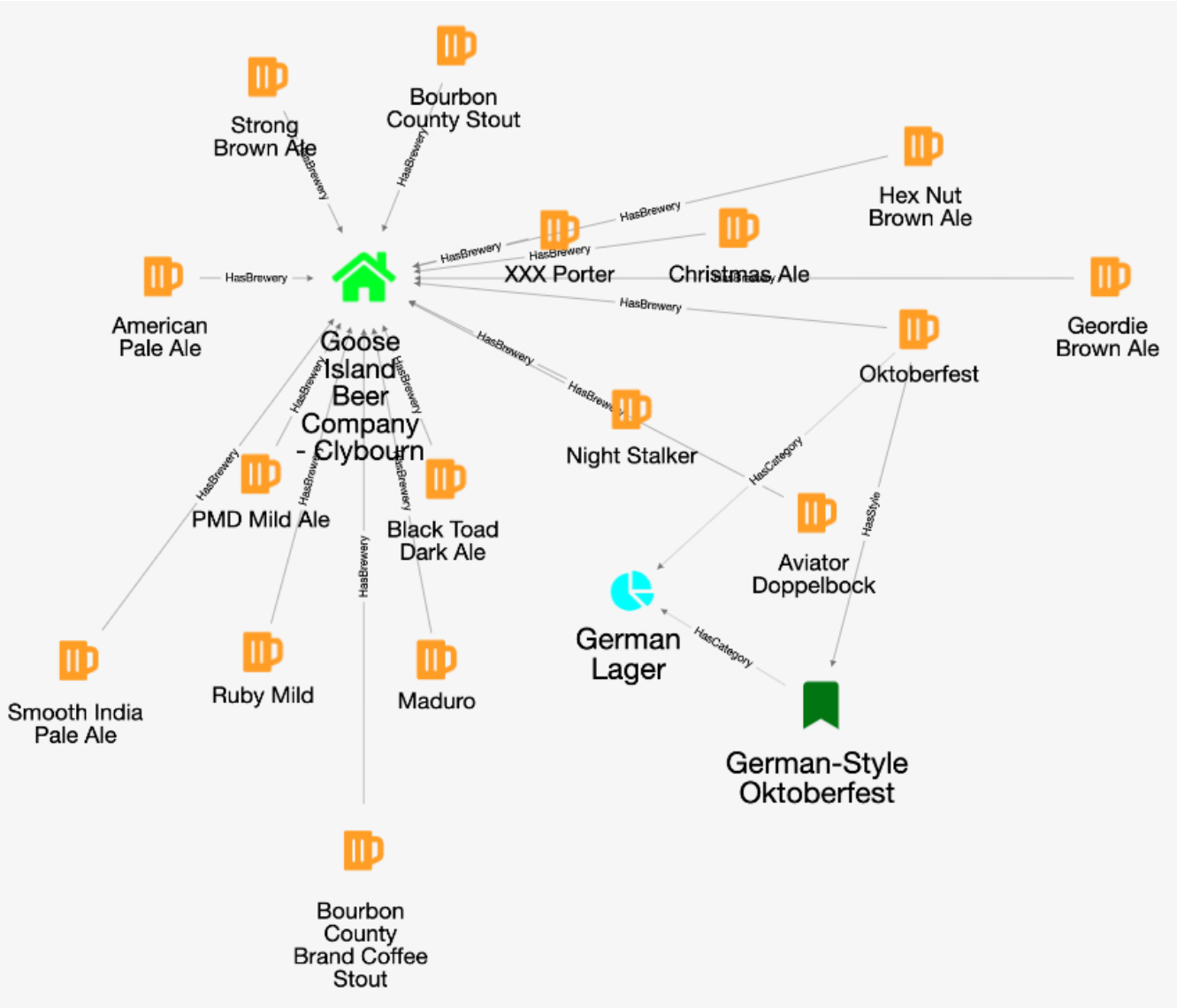
After selecting the attribute `name` on each node types, this is the result.



You can save your setting in a file and share the settings with your colleagues. To do this, click on `Export` button and select `Settings`, then download the file. You can re-apply the same style by selecting `Import` and then `Settings`. Upload the file saved before and your style settings will be

restored. You can share the setting filw with your colleagues and friends to work on the same dataset by using the same style.

Below you can find an example of customization for the OpenBeer database with custom icons, colors and labels:

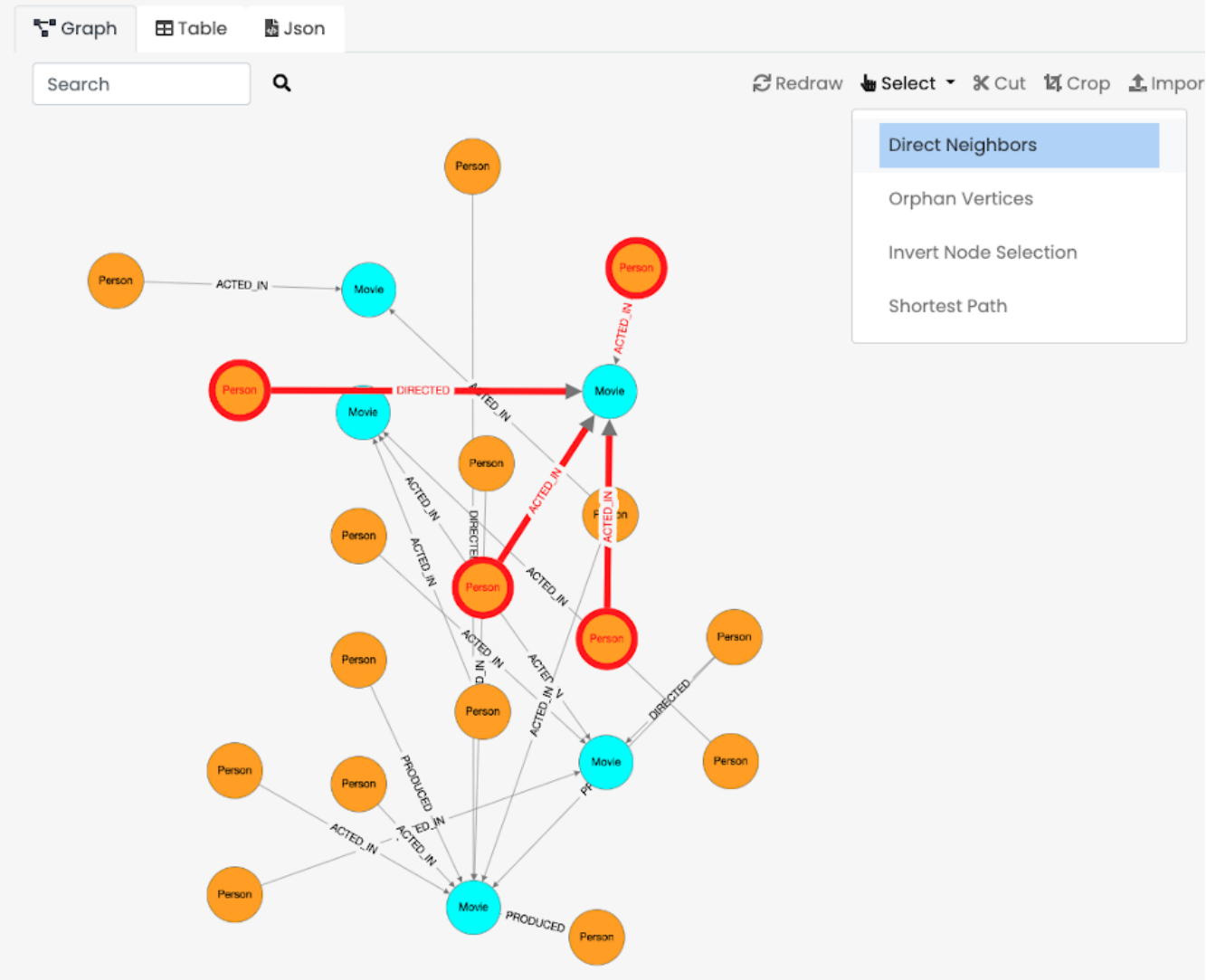


Direct Neighbors

Selects the nodes directly connected to the selected ones.

Usage

Select one or more nodes from the graph and click on `Select → Direct Neighbors`.

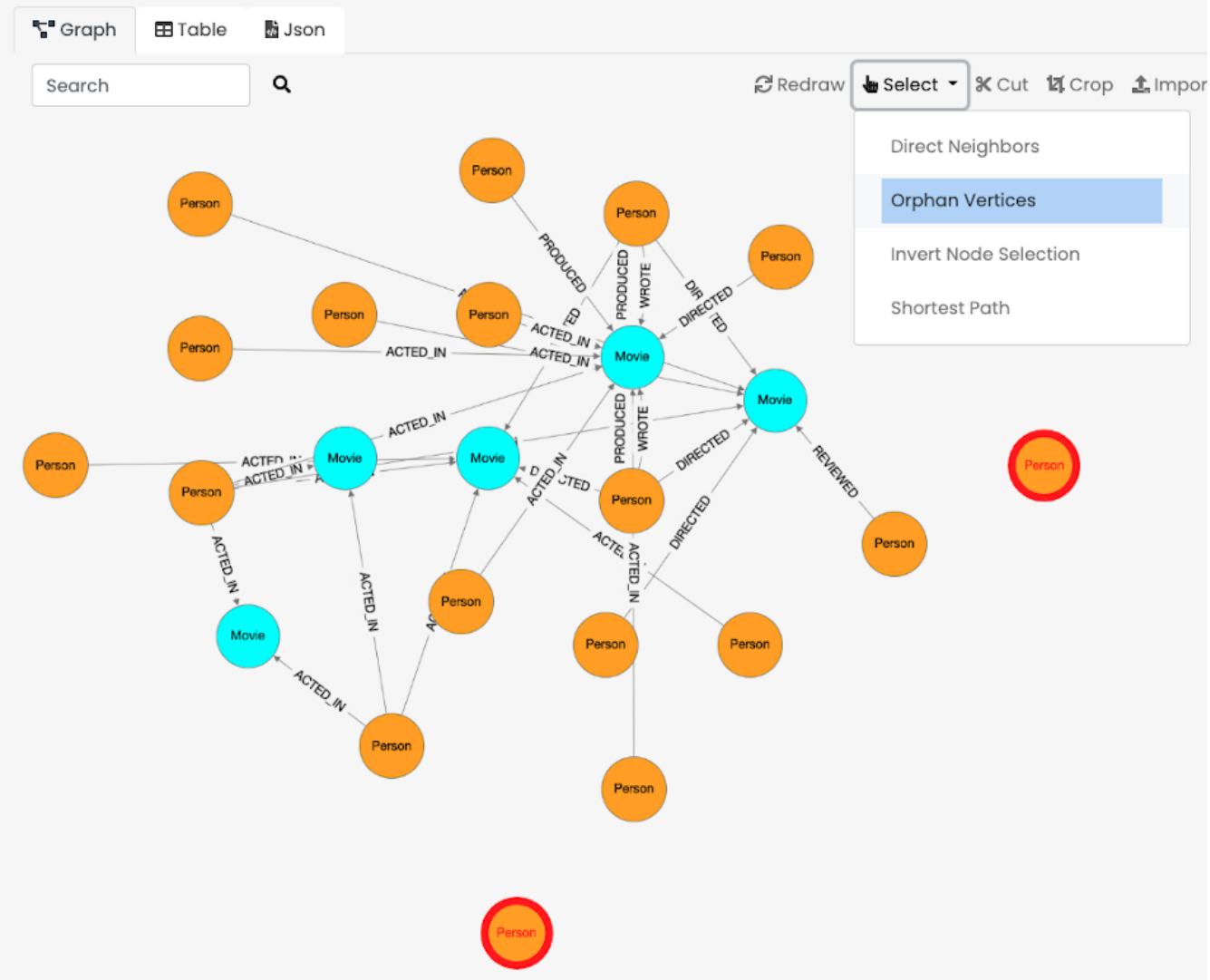


Orphan Vertices

Selects the nodes that are not connected with any other node.

Usage

Click on `Select → Orphan Vertices`.

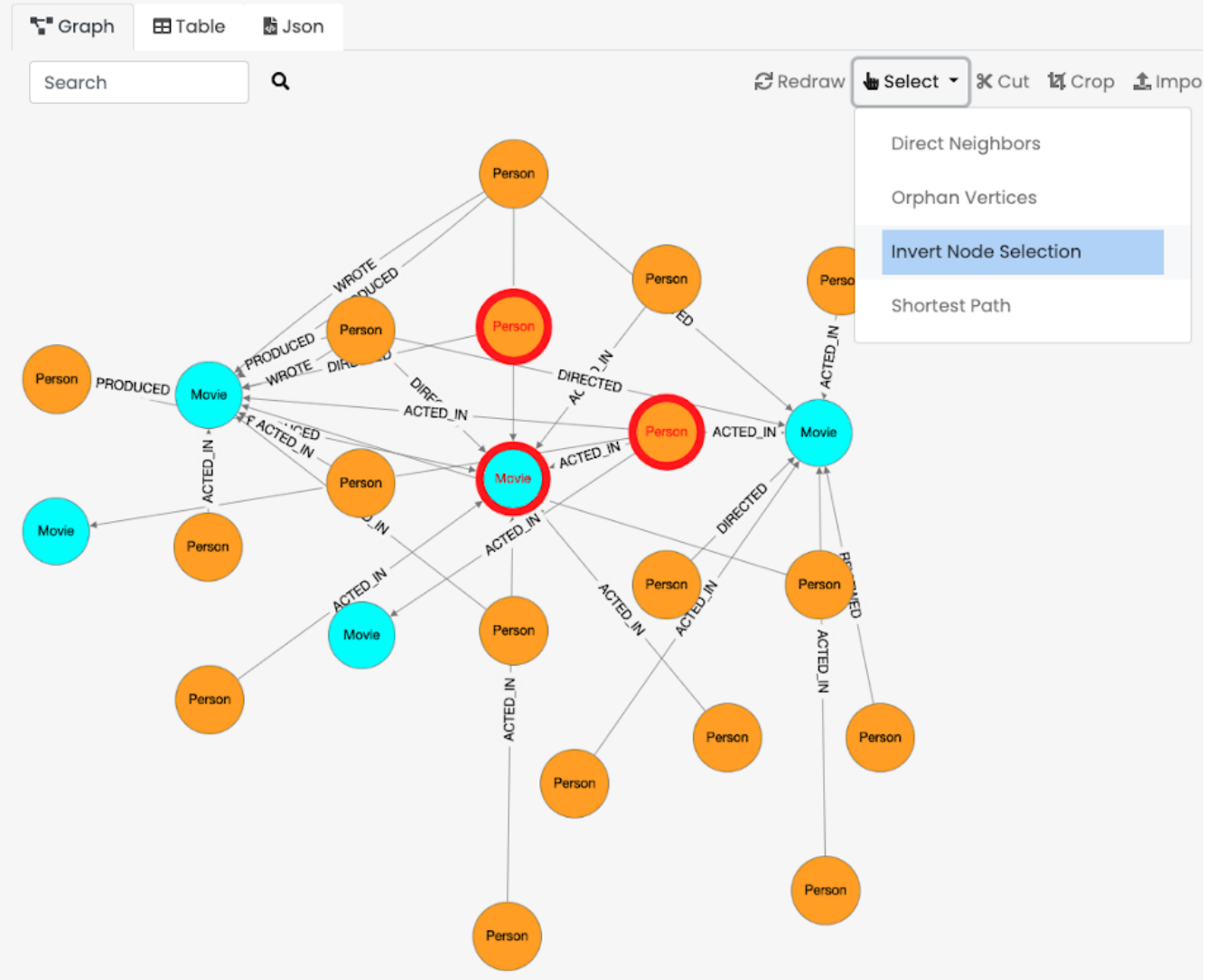


Invert Node Selection

Inverts the current selection. All the elements that are currently selected will be not selected and all the element that were not selected become selected.

Usage

Select some nodes from the graph and click on `Select → Invert Node Selection`.



Shortest Path

Displays the shortest path between 2 nodes. The Dijkstra algorithm is used (with fixed weight 1 per node). If the two nodes are connected, the entire path will be selected.

Usage

Select 2 nodes from the graph and click on `Select → Shortest Path`.

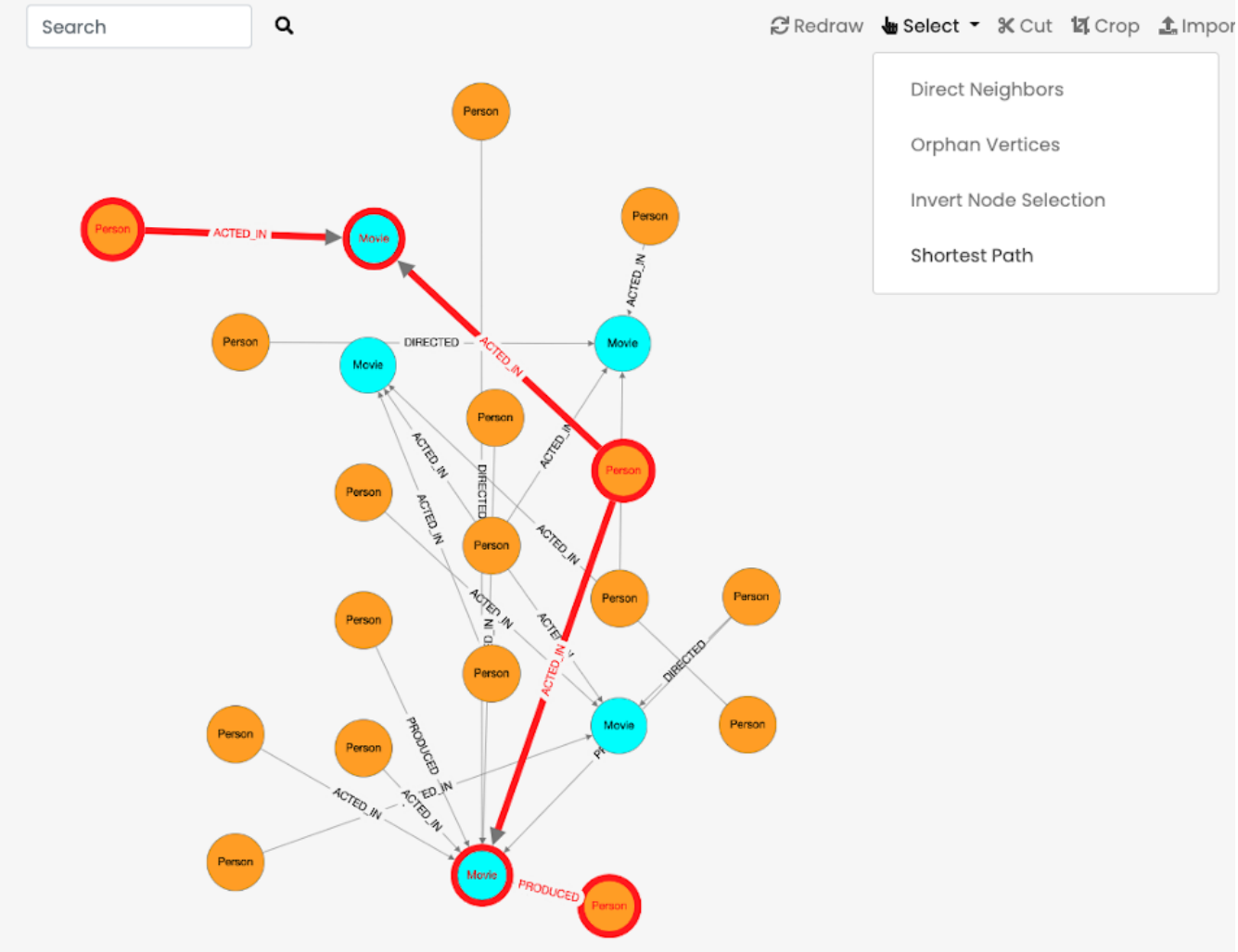
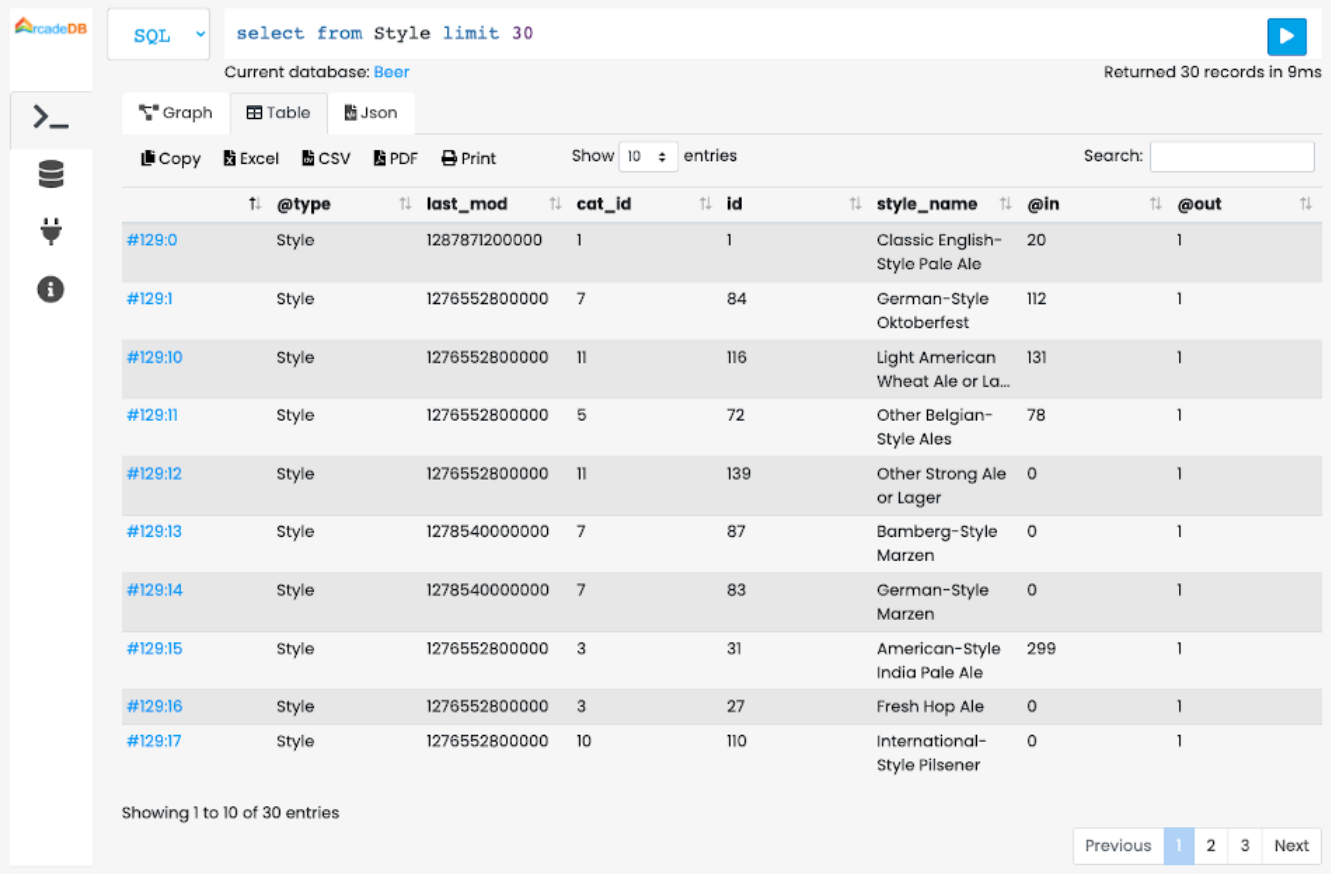


Table Panel

The Table panel renders the result set as a table. If the result of the command is a graph, then both vertices and edges will be flattened into a table. If the result has documents, they will be displayed in table format as well. Connections to other records (like edges in vertices) are not displayed in the table, but only the number of connection is reported. In the example below `@in` is the number of incoming edges for each vertex, and `@out` the number of outgoing edges.



By clicking on the RecordID (RID) (always the first column), the record will be displayed in the graph view with all its attributes.

The Table View automatically layout the records in pages. You can select the amount of records per page and moving between pages with the toolbar at the bottom of the table.

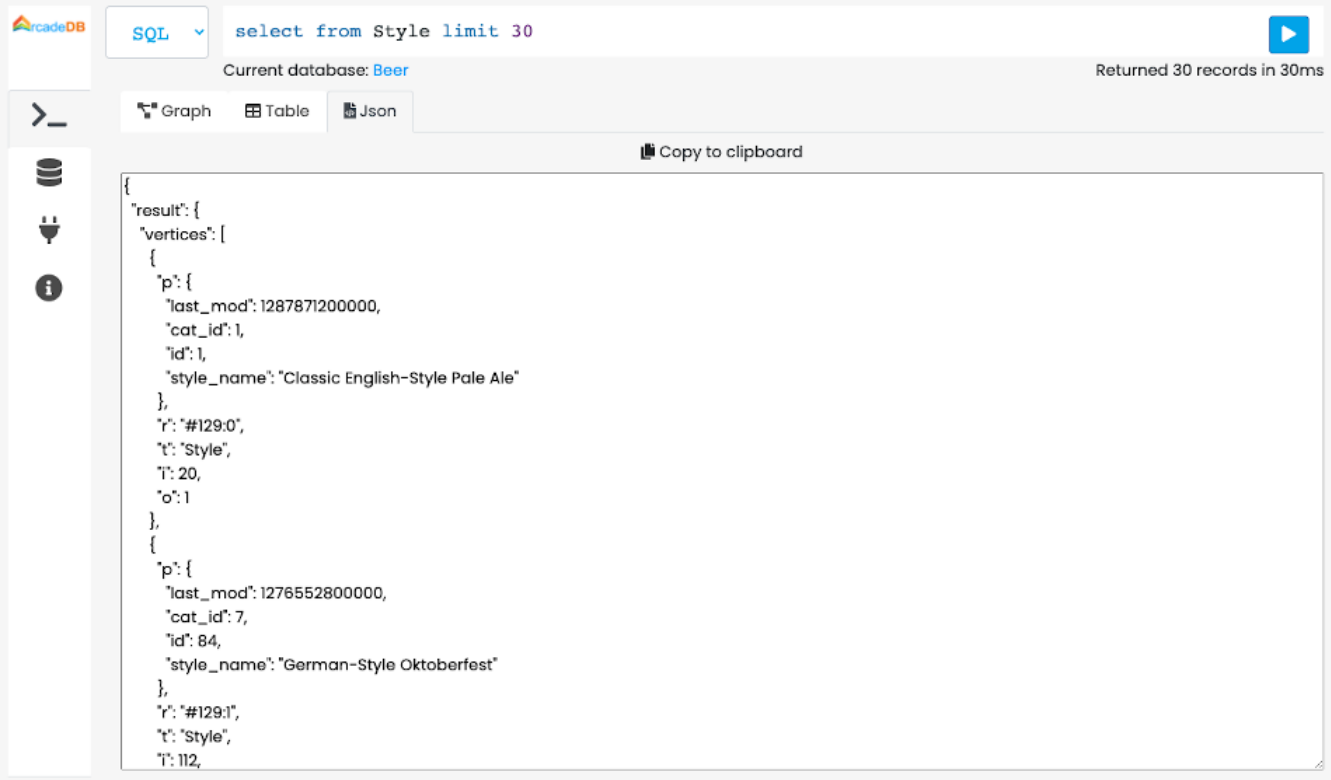
To quick search a record, type what you're looking for in the `Search` input field. The filtering works in real-time as soon as you type. The filtering only applies on the current result set.

The table can be exported in the following formats:

- **Copy**, to copy the entire content in the clipboard. You can then paste the content into your favorite editor or document with CTRL+V or CMD+V.
- **Excel**, for Microsoft® Excel format
- **CSV** (Comma Separated values)
- **PDF** to export the entire table in PDF format
- **Print** to print all the pages of the table

JSON Panel

This panel renders the command result as a JSON. The JSON returned from the HTTP API of the ArcadeDB Server.



Press the `Copy to clipboard` to copy the entire content into the clipboard.

Database Panel

The Database Panel shows the information about the selected database and its schema and allows to execute the most common operations.

The main parts of the Database Panel are:

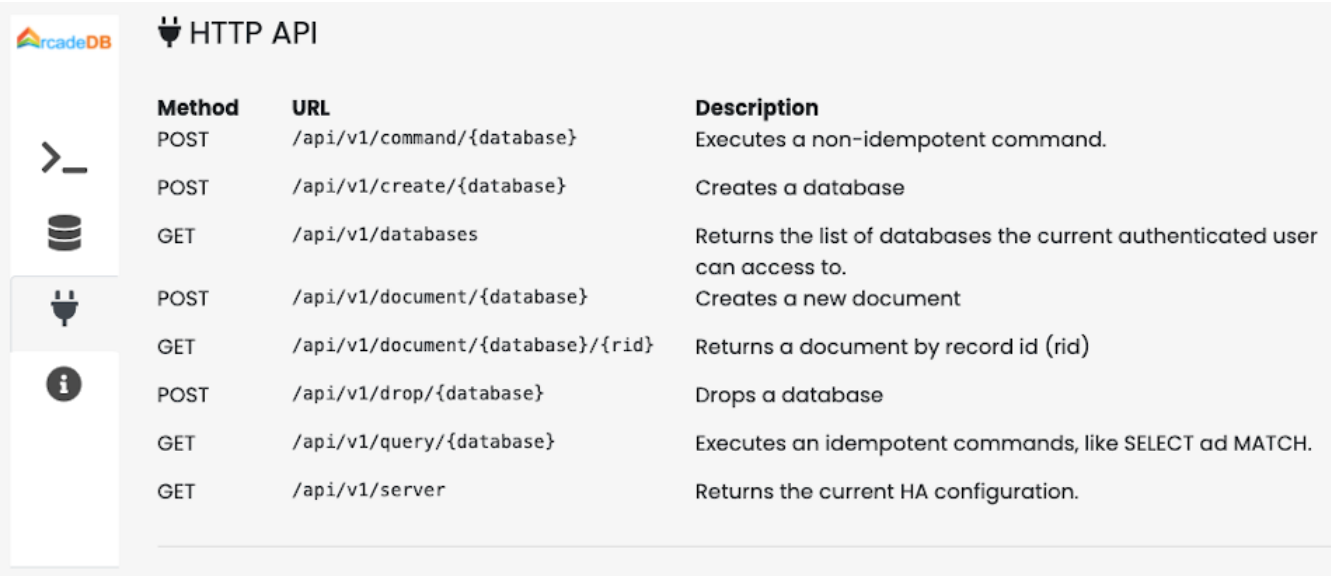
- **Server Version**, report the version you are using when you open an issue
- **User**, the user logged into the server. The list of available databases is filtered by the current

user. User the `admin` user to access to all the databases. See Users.

- **Selected Database**, the selected database. Click to select a different database from the available on the server for the current user.
- **Database Commands**:
  - **Create** to create a new database. Enter the database name in the popup and the new database will be ready to be used
  - **Drop** to drop the current database. ***NOTE: This operation cannot be undone.***
  - **Backup** to execute a backup of the selected database. The backup will be available under the directory `backups` where ArcadeDB server is installed. The generated backup filename is in the format `backups/<db-name>/<db-name>-backup-<timestamp>.tgz`, where the timestamp is expresses from the year to the millisecond. Example of backup file name `backups/TheMatrix/TheMatrix-backup-20210921-172750767.zip`. For more information look at Backup.
- **Types**, with a vertical tab you can select the type you're interested in. One a type is selected, its information are displayed, such as configured indexes and properties.
- **Actions** is a list of quick actions you can execute against the selected type. The most common actions are:
  - **Display tge first 30 records** of the selected type
  - **Display tge first 30 records with all the vertices connected** to display a graph of the first 30 records. The graph will have the 30 records and their direct neighbors.

API Panel

This panel contains the description of the public HTTP API exposed on the current server.



Information Panel

This panel contains a quick reference to the online documentation.

Useful Resources

- Open Source project on GitHub
- ArcadeDB understands multiple languages:
  - SQL (inspired from OrientDB SQL dialect that supports pattern matching on graphs)
  - Cypher
  - Apache Gremlin (Apache Tinkerpop v3.4.x)
  - MongoDB Query Language
- Tutorials: Java Tutorial
- Tools: Working with the Console Tutorial
- ArcadeDB can be used as:
  - Embedded from any language on top of the Java Virtual Machine
  - Remotely by using HTTP/JSON
  - Remotely by using a Postgres driver (ArcadeDB implements Postgres Wire protocol)
  - Remotely by using a MongoDB driver (only a subset of the operations are implemented)
  - Remotely by using a Redis driver (only a subset of the operations are implemented)
- Misc: Docker, Kubernetes

Chapter 6. Tools

6.1. Console

Edit this section

Run the console by executing `console.sh` under `bin` directory:

```
~/arcadedb $ bin/console.sh

ArcadeDB Console v.24.3.1 - Copyrights (c) 2021 Arcade Data Ltd (https://arcadedb.com)

>
```

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               -> sets console query
language
-- <comment>                                                    -> comment (no
operation)
quit|exit                                                       -> exits from the
console
```

The `close` command can be used without path or url, then it closes the currently connected database.

The comment command `--` is useful for scripts.

The multi-line comment syntax `/* … */` is not supported in the console.

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.

6.1.1. 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 explicitely set, when you start the console, like

```
~/arcadedb $ ./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 elon identified by muskmusk 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.

6.1.2. Scripting

The console can also run local SQL scripts by passing the file (path):

```
~/arcadedb $ bin/console.sh myscript.sql
```

or using the `LOAD` command:

```
~/arcadedb $ bin/console.sh "LOAD myscript.sql"
```

or passing the commands as string argument:

```
~/arcadedb $ 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.

6.1.3. 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
```

6.2. Backup of a Database

Edit this section

ArcadeDB allows to execute a non-stop backup of a database while it is used without blocking writes or affecting performance. You can execute the backup of a database from SQL.

Look at Backup Database SQL command for more information.

6.2.1. 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.

6.3. Restore a Database

Edit this section

ArcadeDB allows to restore a database previously backed up.

> If a server is running, it must be restarted in order to access to the restored database.

Example

Example for restoring the database "mydb" from the backup located in `backups/mysb/mydb-backup-20210921-172750767.zip`.

```
~/arcadedb $ bin/restore.sh -f backups/mysb/mydb-backup-20210921-172750767.zip -d
databases/mydb
```

6.3.1. 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.

6.4. Importer

Edit this section

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 CSV files
- Generic RDF files

From file of types:

- Plain text
- Compressed with ZIP (only the first file is read)
- 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

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 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://`.

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]
...
```

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,
- JSON Importer
- Neo4j Importer,
- OrientDB Importer

6.4.1. 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
```

Below you can find all the supported settings for the Importer.

Setting	Default	Description
url		url of the file to import
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
parallel	Half of the available cores - 1. If you have 8 cores, the default is 3	The number of concurrent threads used

Setting	Default	Description
typeIdProperty		Property that represents the ID of the vertex
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
analysisLimitBytes	100,000	Maximum number of bytes parsed from the source to determine the source file type
analysisLimitEntries	10,000	Maximum number of entries (if applicable) parsed from the source to determine the source file type
parsingLimitBytes		Maximum number of bytes parsed from the source to be imported
parsingLimitEntries		Maximum number of entries imported
mapping	null	
probeOnly	false	Only probe if url is requestable 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, graphml, graphson)
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
documentsSkipEntries	0	Number of rows to skip from the documents file
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)
verticesDelimiter		Delimiter used to separate vertices

Setting	Default	Description
verticesHeader		Header containing the properties in the CSV vertices. One property per column. If not defined it is parsed from the first line
verticesSkipEntries	0	Number of rows to skip from the vertices file
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)
edgesDelimiter		Delimiter used to separate edges
edgesHeader		Header containing the properties in the CSV edges. One property per column. If not defined it is parsed from the first line
edgesSkipEntries	0	Number of rows to skip from the edges file
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.
efConstruction	256	Size of dynamic neighbor candidate list of (during insert).
ef	256	Number of nearest neighbors to return (in layer search).
m	16	TODO
vectorType	`float`	The data type of a vector element, for example 'float'.

Setting	Default	Description
idProperty	`"name"`	TODO

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.

6.4.2. JSON Importer

Edit this section

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.

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:

```
~/arcadedb $ create database employees
```

Then execute the following command (assuming the file above is saved in the local directory as `employees.json`:

```
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. In case of numbers, the JSON parser always uses DOUBLE as a type. You can, for example, use LONG to force using LONG instead of DOUBLE
- `@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.

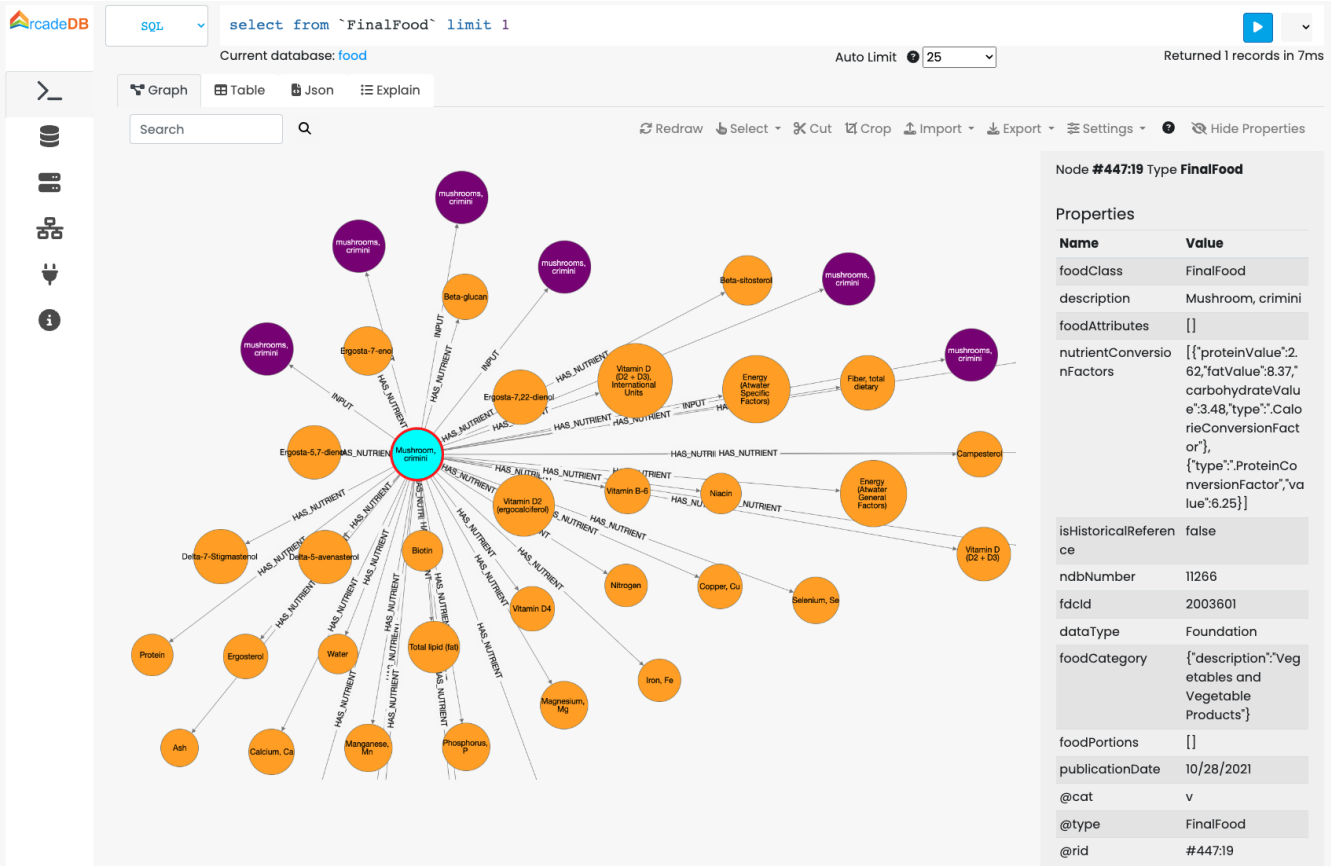
First, create a new database "food" by using the console:

```
~/arcadedb $ create database food
```

Then execute the following command:

```
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. In case of numbers, the JSON parser always uses DOUBLE as a type. You can, for example, use LONG to force using LONG instead of DOUBLE
- `@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

6.4.3. OrientDB Importer

Edit this section

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.

```
~/arcadedb $ create database MyDatabase
~/arcadedb $ import database file:///temp/orientdb-export.tgz
```

6.4.4. Neo4j Importer

Edit this section

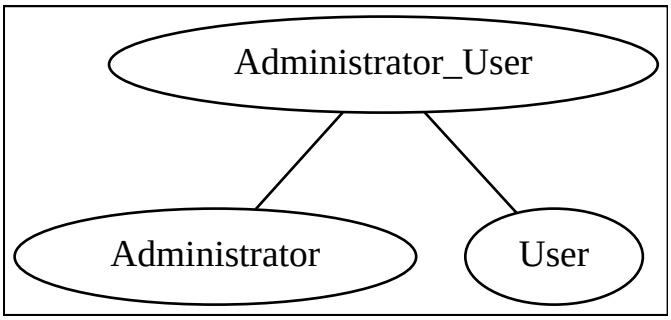
ArcadeDB is able to import a database exported from Neo4j in JSONL format (one json per line).

To export a Neo4j database follow the instructions in Export in JSON. The resulting file contains one json per line.

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.

Example

Example of importing the following mini graph exported from Neo4j. This is the example taken

from Neo4j documentation about Export to JSON.

```
{"type":"node","id":"0","labels":["User"],"properties":{"born":"2015-07-04T19:32:24"
,"name":"Adam","place":{"crs":"wgs-84","latitude":33.46789,"longitude":13.1,"height"
:null},"age":42,"male":true,"kids":["Sam","Anna","Grace"]}}
{"type":"node","id":"1","labels":["User"],"properties":{"name":"Jim","age":42}}
{"type":"node","id":"2","labels":["User"],"properties":{"age":12}}
{"id":"0","type":"relationship","label":"KNOWS","properties":{"since":1993,"bffSince":
"P5M1DT12H"},"start":{"id":"0","labels":["User"]},"end":{"id":"1","labels":["User"]}}
```

As you can see, the file contains one json per line. First all the nodes (vertices), then the relationships (edges).

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.

```
~/arcadedb $ create database PanamaPapers
~/arcadedb $ import database file:///temp/panama-papers-neo4j.jsonl

ArcadeDB 21.9.1 - 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, skipped 1 edges (0 vertices/sec
elapsed=0 secs)
- Creation of edges started: creating edges between vertices
- Creation of edged completed: created 1 edges, (0 edges/sec elapsed=0 secs)
*******************************************************************************
********
Import of Neo4j database completed in 0 secs with 0 errors and 0 warnings.

SUMMARY

- Vertices.............: 0
-- User                : 3
- Edges................: 0
-- KNOWS               : 1
- Total attributes.....: 9
*******************************************************************************
********

NOTES:
- you can find your new ArcadeDB database in 'databases/PanamaPapers'
```

6.5. Upgrade ArcadeDB

Edit this section

ArcadeDB is able to automatically upgrade a database when a newer version of ArcadeDB is used. The migration is completely automatic and transparent.

6.5.1. Steps

1. Download and extract a newer version of ArcadeDB in the local file system
2. Stop any ArcadeDB running servers (or close manually the database by using the HTTP command "close database").
3. Copy the databases directory from the old server to the new one
4. Start the new server

6.6. Downgrade ArcadeDB

In the case you need to downgrade to an older version of ArcadeDB, check the binary compatibility between the versions. ArcadeDB uses the semantic versioning with 100% compatibility for migration of databases up or down between patch version (the Z in X.Y.Z). To downgrade to a minor or major, the safest way is to export the database with the newest version and re-import the database with the older version.

Chapter 7. API

The power of a Multi-Model database is also in the way you can interact with it. ArcadeDB supports multiple languages, so it's easier to use it coming from other DBMS.

	JVM Embedded API	SQL	Apache Gremlin	Cypher	GraphQL	MongoDB Query	Redis
Speed	* * *	* *	* *	* *	* *	* *	* *
Flexibility	* * *	*	* *	* *	* *	* *	*
Support for Documents	Yes	Yes	Yes	Yes	Yes	Yes	No
Support for Graph	Yes	Yes	Yes	Yes	Yes	No	No
Embedded mode	Yes	Yes	Yes	Yes	Yes	Yes	No
Remote mode	No	Yes via HTTP/JSON and Postgres Driver	Yes via Gremlin Server, HTTP/JSON and Postgres Driver	Yes via Gremlin Server, HTTP/JSON and Postgres Driver	Yes via HTTP/JSON and Postgres Driver	Yes via MongoDB Protocol plugin, HTTP/JSON and Postgres Driver	Yes via Redis Protocol plugin

7.1. Drivers

ArcadeDB provides HTTP API to interface to the server from a remote application. You can use any programming language that supports HTTP calls (pretty much any language) or use a driver to have a little abstraction over HTTP API.

Language	Project URL	License
Java	Bundled with the project. Look at `Remote Database` class	Apache 2
CHICKEN Scheme	http://wiki.call-cc.org/eggref/5/arcadedb	zlib-acknowledgement
.NET	https://github.com/tetious/ArcadeDb.Client	Apache 2
Python	https://github.com/ExtReMLapin/pyarcade	Apache 2
Ruby	https://github.com/topofocus/arcadedb	MIT
RUST	https://crates.io/crates/arcadedb-rs	Apache 2

7.2. Java API (Local)

Edit this section

Add the following dependency in your Maven pom.xml file under the tag `<dependencies>`:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-engine</artifactId>
    <version>24.3.1</version>
</dependency>
```

ArcadeDB works in both synchronous and asynchronous modes. By using the asynchronous API you let to ArcadeDB to use all the resources of your hw/sw configuration without managing multiple threads.

Synchronous API	Asynchronous API
The Synchronous API executes the operation immediately, by the current thread, and returns when it's finished. If you use a procedural approach, using the synchronous API is the easiest way to use ArcadeDB. In order to use all the resource of your machine, you might use multi-threading in your application.	The Asynchronous API schedules jobs to be executed as soon as possible by a pool of threads. ArcadeDB optimizes the usage of asynchronous threads pool to be equals to the number of cores found in the machine (you can modify it via API). Use Asynchronous API if the response of the operation can be managed in asynchronous way. Thanks to the asynchronous API, your application doesn't need to be multi-threads to use all the available cores.

7.2.1. 10-Minute Tutorial

Edit this section

You can create a new database from scratch or open an existent one. Most of the API works in both synchronous and asynchronous modes. The asynchronous API are available from the `<db>.async()` object.

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 arcade = 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.

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. 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"));
    }
  });
}
```

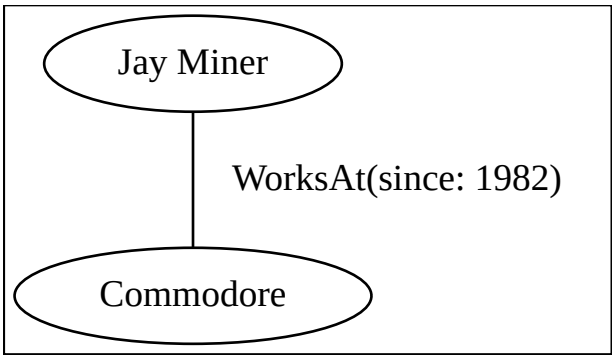
By using a map, parameters are referenced by name (`:age` and `:city` in this example).

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 elon = db.newVertex("User", "name", "Elon", "lastName", "Musk").
save();
    MutableVertex steve = db.newVertex("User", "name", "Steve", "lastName", "Jobs"
```

```
).save();
    elon.newEdge("IsFriendOf", steve, 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 "Elon" vertex to the "Steve" 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	No	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 Elon Musk's RID
    Vertex elon = 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 Steve's vertex in a flash by using this:

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

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 = ?", "Steve", "Jobs" );
    Vertex steve = 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 "Elon" to "Steve". 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 `Elon ---Friend--> Steve`. This means that if I want to retrieve all Elon's friends, I could start from the vertex "Elon" and traverse all the **outgoing** edges of type

"IsFriendOf".

Instead, if I want to retrieve all Steve's friends, I could start from Steve 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 Steve's friends, this is the example by using API:

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

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

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

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 = ?", "Steve", "Jobs" );
  });
}
```

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", "Steve",
"Jobs" );
  });
}
```

Traverse via Apache Gremlin

Since ArcadeDB is 100% compliant with Gremlin 3.6.x, you can run this query against the Apache Gremlin Server configured with ArcadeDB:

```
g.V().has('name','Steve').has('lastName','Jobs').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 = 'Steve' and me.lastName = 'Jobs'
RETURN friend.name, friend.lastName
```

For more information about Cypher see: Cypher support

7.2.2. Schema

Edit this section

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 number of available cores is used
- `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);
```

7.2.3. 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();
```

7.2.4. 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
  - `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 migh 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 thowing 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.

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).

7.2.5. 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:

```
TimeZone getTimeZone();
void     setTimeZone(TimeZone timeZone);
String   getDateFormat();
void     setDateFormat(String dateFormat);
String   getDateTimeFormat();
void     setDateTimeFormat(String dateTimeFormat);
String   getEncoding();
```

7.2.6. Embedded Documents

Edit this section

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.

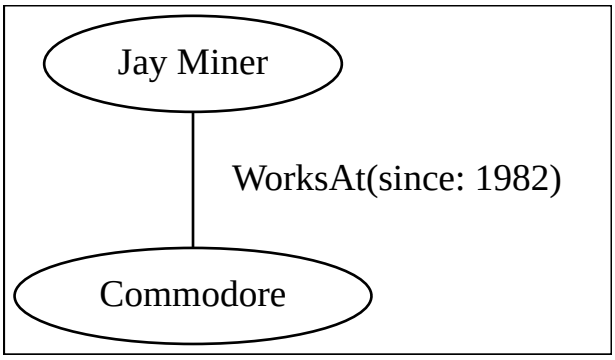
```
{
  "firstName": "Jay",
  "lastName": "Miner",
  "worksAt": {
    "companyName": "Commodore",
    "since": "1982"
  }
}
```

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( (tx) -> {
  // 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", "firstName", "Jay", "lastName",
"Miner");

  EmbeddedDocument commodore = jay.newEmbeddedDocument("Company", "worksAt").set
("compamyName", "Commodore", "since", 2010);

  // MAKE THE MAIN DOCUMENT PERSITENT
```

```
  jay.save();
});
```

Modeling with a graph it would be something like this:



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

```
db.transaction( (tx) -> {
  // 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", "firstName", "Jay", "lastName",
"Miner").save();
  MutableVertex commodore = db.newVertex("Company", "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( (tx) -> {
  // 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", "firstName", "Jay", "lastName",
"Miner").save();
  jay.newEmbeddedDocument("Address", "residenceAddress", "city", "San Francisco",
"state": "CA", "country": "USA");

  MutableVertex commodore = db.newVertex("Company", "name", "Commodore").save();
  commodore.newEmbeddedDocument("Address", "hqAddress", "city", "Palo Alto", "state":
```

```
"CA", "country": "USA");
  commodore.newEmbeddedDocument("Address", "ukAddress", "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( (tx) -> {
  ResultSet result = db.query( "SQL", "select from Employee where firstName = ? and
lastName = ?", "Jay", "Miner" );
  Vertex jay = result.next();

  EmbeddedDocument residenceAddress = (EmbeddedDocument) jay.get("residenceAddress");
  System.out.println( "Jay's lives in " + residenceAddress.getString("city") );
});
```

7.2.7. Using Vector Embeddings from Java API

Edit this section

Using the Embedded Java API is the fastest way to insert vector embeddings into the database. At the beginning it's much faster to pass through the `HnswVectorIndexRAM` implementation and then generate the persistent index after loaded.

Schema types are created automatically, but you could create them in advance if you want special settings, like a specific numbe rof buckets, or a custom page size.

Example of inserting embeddings in RAM first, and then create the persistent HNSW index.

```
if (!database.getSchema().existsIndex(indexName)) {
  // FIRST TIME: USE OPTIMIZED RAM BASED HNSW INDEX AS FIRST STEP
  final HnswVectorIndexRAM<Object, float[], Item<Object, float[]>, Float> hnswIndex =
HnswVectorIndexRAM.newBuilder(
          embeddingService.getDimensions(), distanceFunction, 100_000).withM(m).
withEf(ef).withEfConstruction(efConstruction)
      .build();

  hnswIndex.addAll((Collection<Item<Object, float[]>>) embeddings, Runtime.
getRuntime().availableProcessors(), null);

  // CREATE UNIQUE TYPE ASSOCIATED TO THE LIBRARY
  final String statementTypeName = getStatementTypeName(vertexLibrary);

  hnswIndex.createPersistentIndex(database)//
      .withVertexType(STATEMENT_TYPE_NAME).withEdgeType(PROXIMITY_TYPE_NAME)//
      .withVectorProperty(VECTOR_PROPERTY_NAME, Type.ARRAY_OF_FLOATS)//
```

```
        .withIdProperty(TEXT_PROPERTY_NAME)//
        .withDeletedProperty(DELETED_PROPERTY_NAME)//
        .create();
}
```

Once the persistent index is created, you can just add new entries into the HNSW index:

```
if (database.getSchema().existsIndex(indexName)) {
  final HnswVectorIndex index = (HnswVectorIndex) database.getSchema().getIndexByName
(indexName);

  index.addAll(embeddings, (statement, item, total) -> {
    // YOUR CALLBACK TO MANAGE CASCADING OPERATIONS WHEN VERTICES ARE CREATED AND
INDEXED, IF ANY
  });
}
```

The callback in the `addAll()` method is useful when you have a graph/tree connected to the vertices created by the index. For example, if you are indexing a book, probably you calculate a vector of embeddings per statement, and then you group the statements in paragraphs.

7.2.8. Events

Edit this section

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.

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"));
```

7.3. Java API (Remote)

ArcadeDB comes with a convenient Java library to work with a remote database.

Add the following dependencies in your Maven pom.xml file under the tag `<dependencies>`:

```
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-engine</artifactId>
    <version>24.3.1</version>
</dependency>
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-network</artifactId>
    <version>24.3.1</version>
```

```
</dependency>
```

7.3.1. 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

Let's create a RemoteDatabase instance pointing to the database "mydb" on ArcadeDB Server located on "localhost", port 5432. User and passwords are respectively "root" and "playwithdata":

```
RemoteDatabase database = new RemoteDatabase("localhost", 5432, "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.

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 (!database.exists()) {
    database.create();
}
```

For a list of all databases hosted by the server use:

```
List<String> databases = database.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.

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.

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 arry of objects of type `Result`, which can be read as Map:

```
{name=Jay, @cat=d, @rid=#1:0, lastcall=1675961747290, surname=Miner, @type=Customer}
```

7.4. Java Reference

Edit this section

7.4.1. `DatabaseFactory` Class

Edit this section

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.

Methods

Example:

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

close()

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

Syntax:

```
void close()
```

exists()

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

Syntax:

```
boolean exists()
```

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();
```

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
}
```

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();
}
```

7.4.2. `Database` Interface

Edit this section

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

Methods (Alphabetic order)

async()	begin()	close()	commit()	deleteRecord()
drop()	getSchema()	isOpen()	iterateBucket()	iterateType()

query() positional parameters	query() (parameter map)	command() positional parameters	command() (parameter map)	lookupByKey()
lookupByRID()	newDocument()	newEdgeByKeys()	newVertex()	rollback()
scanBucket()	scanType()	transaction() default	transaction() with retries	

Methods (By category)

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()		
		iterateBucket()		
		scanBucket()		
		scanType()		

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 boolean onNext(Result result) {
    System.out.println( "Name = " + result.getProperty("name"));
    return true;
  }

  @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");
```

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 `IllegalArgumentException` exception.

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.

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"));
}
```

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 `IllegalArgumentException` 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

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.

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 tather 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();

  Integer age = (Integer) record.get("age");
  if (age =! null && age >= 21 )
    System.out.println("Found customer: " + record.get("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 tather 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).

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(db, "#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
```

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();
```

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.

Syntax:

```
rollback()
```

Example:

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

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) -> {
  Integer age = (Integer) record.get("age");
  if (age =! null && age >= 21 )
    System.out.println("Found customer: " + record.get("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;
});
```

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, the transaction is repeated up to `retries` times

Syntax:

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

Examples:

Example by using Java8+ 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(Database database) {
    final MutableVertex v = database.newVertex("Author");
    v.set("name", "Jay");
    v.set("surname", "Miner");
    v.save();
  }
});
```

`DatabaseAsyncExecutor` Interface

Edit this section

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.

Methods

command(...) positional parameters	**command(Map<K,V>)** parameter map	**query(...)** positional parameters	**query(Map<K,V>)** parameter map

transaction()	transaction(retries)	transaction(retries, callbacks)	createRecord()
updateRecord()	deleteRecord()		

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 taking a map for 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,
            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 a command that could change the database. This method returns immediately. This is the equivalent to `query()`, but allows the command to modify the database.

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 a command that could change the database. This method returns immediately. This is the equivalent to `query()`, but allows non-idempotent commands to modify the database.

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);
```

createRecord(record, newRecordCallback [,errorCallback])

Creates a record (document or vertex) asynchronously. 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", "Elon");
database.async().createRecord(vertex,
                              v -> { System.out.println("Record " + v.toJSON() + "
created") });
```

updateRecord(record, updateRecordCallback [,errorCallback])

Updates a record (document or vertex) asynchronously. 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 (document or vertex) asynchronously. 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") });
```

7.5. Native-Select

The native-select API is an alternative approach to query a database from Java. This API combines an object-oriented syntax with SQL semantics and enables faster query processing than SQL.

Compare the following SQL and native-select queries:

```
SELECT FROM mytype WHERE age > 20 AND name != 'Joe' ORDER BY age LIMIT 10;
```

```
database.select().fromType("mytype").where().property("age").ge().value(20).property("
name").neq().value("Joe").orderBy("age").limit(10).compile();
```

Note that the projections happen after the query is compiled.

An important part of the query is the filtering. Filtering in native-select is done by chaining compatible objects. A filter is composed of a `SelectWhereLeftBlock`, `SelectWhereOperatorBlock`, and `SelectWhereRightBlock`. Filters are chained or filtering is completed by the `SelectWhereAfterBlock`.

In the following tables are listed which group methods belonging to objects related to a native

select query. The columns of these tables list the method signature, consisting of name, arguments and return type, a comment on its effect, and a corresponding SQL element.

`Select` Object

Signature	Comment	SQL
`select() → Select`	Initializes a "native select" query.	`SELECT`

`Select` Methods

Signature	Comment	SQL
`property(String) → Select`	Inserts a property by its name.	-
`value(Object) → Select`	Inserts a literal value.	-
`parameter(String) → Select`	Inserts a named placeholder to be specified in the compiled query.	-
`fromType(String) → Select`	Sets type to select from.	`FROM`
`fromBuckets(String, …) → Select`	Sets bucket(s) to select from by name.	`FROM`
`fromBuckets(Integer, …) → Select`	Sets bucket(s) to select from by number.	`FROM`
`polymorphic(Boolean)`	Allow or disallow polymorphic queries.	-
`timeout(Long,TimeUnit,Boolean)`	Set a timeout for the query.	-
`json(JSONObject)`	paste native-select query as JSON object.	-
`where() → SelectWhereLeftBlock`	Initializes a filter.	`WHERE`

`SelectWhereLeftBlock` Methods

Signature	Comment	SQL
`property(String) → SelectWhereOperatorBlock`	Inserts a property by its name.	-
`value(Object) → SelectWhereOperatorBlock`	Inserts an explicit value.	-
`parameter(String) → SelectWhereOperatorBlock`	Inserts a named placeholder to be specified in the compiled query.	-

`SelectWhereOperatorBlock` Methods

Signature	Comment	SQL
`eq()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is equal to `SelectWhereRightBlock`.	`=`
`neq()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is unequal to `SelectWhereRightBlock`.	`!=`
`lt()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is less than `SelectWhereRightBlock`.	`<`
`le()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is less or equal than `SelectWhereRightBlock`.	`⇐`
`gt()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is greater than `SelectWhereRightBlock`.	`>`
`ge()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is greater or equal than `SelectWhereRightBlock`.	`>=`
`like()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is case-sensitive "like" `SelectWhereRightBlock`.	`LIKE`
`ilike()` → `SelectWhereRightBlock`	Tests if `SelectWhereLeftBlock` is case-insensitive "like" `SelectWhereRightBlock`.	`ILIKE`

`SelectWhereRightBlock` Methods

Signature	Comment	SQL
`property(String)` → `SelectWhereAfterBlock`	Inserts a property by its name.	-
`value(Object)` → `SelectWhereAfterBlock`	Inserts an explicit value.	-
`parameter(String)` → `SelectWhereAfterBlock`	Inserts a named placeholder to be specified in the compiled query.	-

`SelectWhereAfterBlock` Methods

`and()` → `SelectWhereLeftBlock`	**Combines two filters using the "and" relation.**	`AND`
`or()` → `SelectWhereLeftBlock`	Combines two filters using the "or" relation.	`OR`

`Select` and `SelectWhereAfterBlock` Methods

Signature	Comment	SQL
`orderBy(String,Boolean)` → `Select`	Orders projections by argument.	`ORDER BY ⋯ DESC ASC`
`limit(Integer)` → `Select`	Limits the number of results.	`LIMIT`
`skip(Integer)` → `Select`	Skips a number of results.	`SKIP`

Signature	Comment	SQL
`compile() → SelectCompiled`	Compiles select query.	-

`SelectCompiled` Methods

Signature	Comment	SQL
`parameter(String,Object) → SelectCompiled`	Inserts a value at a named placeholder.	-
`document() → SelectIterator<Document>`	Return document iterator.	-
`vertices() → SelectIterator<Vertex>`	Return vertex iterator.	-
`edges() → SelectIterator<Edge>`	Return edge iterator.	-
`json() → JSONObject`		-

7.6. Python

Work in progress

7.7. HTTP/JSON Protocol

Edit this section

Overview Endpoints

Action	Method	Endpoint
Get server status	GET	`/api/v1/ready`
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}`
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}`

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.

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 30 seconds, see `server.httpTxExpireTimeout`), the server automatically rolls back and purges expired transactions.

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.

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`.
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.

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.

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
```

Reference

Check if server is ready (GET)

Returns a header-only (no content) status about if the ArcadeDB server is ready.

URL Syntax: `/api/v1/ready`

This endpoint accepts (GET) requests without authentication, and is useful for remote monitoring of server readiness.

Response:

- `204` OK

Example:

```
curl -I -X GET "http://localhost:2480/api/v1/ready"
```

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.

Responses:

- `200` OK
- `403` invalid credentials

Example:

```
curl -X GET "http://localhost:2480/api/v1/server?mode=basic" \
     --user root:arcadedb-password
```

Return:

```
{"user":"root", "version":"24.3.1", "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`
- `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` kills the server gracefully.
- `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

> 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.

Responses:

- `200` OK
- `400` invalid command
- `403` invalid credentials
- `500` invalid JSON request body

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"}
```

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"}
```

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"}
```

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"}
```

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.flushOnlyAtClose true" }' \
     -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"}
```

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 Databases (GET)

Returns a list of available databases for the requesting user.

URL Syntax: `/api/v1/databases`

Responses:

- `200` OK
- `403` invalid credentials

Example:

```
curl -X GET http://localhost:2480/api/v1/databases \
     --user root:arcadedb-password
```

Return:

```
{"user":"root","version":"24.3.1","serverName":"ArcadeDB_0","result":["school","mydata
base"]}
```

Does database exist (GET)

Returns boolean answering if database exists.

URL Syntax: `/api/v1/exists/{database}`

Responses:

- `200` OK
- `400` no database passed

Example:

```
curl -X GET http://localhost:2480/api/v1/exists/school \
     --user root:arcadedb-password
```

Return:

```
{"user":"root","version":"24.3.1","serverName":"ArcadeDB_0","result":true}
```

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 `:`.

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.

Responses:

- `200` OK
- `400` invalid language, invalid query
- `403` invalid credentials
- `500` database does not exist, cannot execute query

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
```

Execute database command (POST)

Executes a non-idempotent command.

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
- `params` (optional), is the map of parameters to pass to the query engine, where parameters are introduced with a colon `:`.
- `serializer` (optional) specify the serializer used for the result:
  - `graph`: returns as a graph separating vertices from edges
  - `record`: returns everything as records
  - 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.

Responses:

- `200` OK
- `202` Accepted
- `400` invalid language, invalid command
- `403` invalid credentials

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
```

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, accepts the following parameters:

- `isolationLevel` is the isolation level for the current transaction, either `READ_COMMITTED` (default) or `REPEATABLE_READ`.

Responses:

- `204` OK
- `400` invalid value
- `401` transaction already started
- `403` invalid credentials
- `500` invalid database, invalid JSON, invalid body

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 expired, then an internal server error is returned.

Response:

- `204` OK
- `403` invalid credentials
- `500` transaction expired, not found, not begun

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.

Response:

- `204` OK
- `403` 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
```

7.8. C#/.NET (HTTP/JSON)

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);
}
```

7.9. NodeJS/JavaScript (HTTP/JSON)

Sample Code

In NodeJS Javascript, HTTP/JSON requests can be made by installing the Axios HTTP package.

1. Ensure NodeJS and Node Package Manager are installed and accessible from the command prompt. If installed, `node -v` and `npm -v` will report their version numbers for confirmation.
2. Create a folder to contain a NodeJS project and a file in this folder called `test.js`.
3. From this folder, run `npm install axios` to add the Axios HTTP package to the folder/project.
4. Open `test.js` and add the needed code to be run.
5. Run the script from the command line with `node test.js`.

The following demonstrates a simple function which can be added to the javascript file. It will return records of the type `Profile` from a database named `mydb` and debug them to the command line.

```
const axios = require('axios')

// Make request
const userPass = 'root:arcadedb-password';
const authString = 'Basic ' + btoa(userPass);
axios.post('http://serveraddress:2480/api/v1/command/mydb', //customize with your
server address (or local host) and db name
    { "language": "sql", "command": "SELECT from Profile" }, //customize with your
required command
    { headers : { 'Authorization' : authString, 'Content-Type': 'application/json' }}
  )

// Show response data
```

```
.then(res => console.log(res.data))
.catch(err => console.log(err))
```

7.10. Elixir (HTTP/JSON)

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(>`.

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
```

7.11. Postgres Protocol Plugin

Edit this section

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>24.3.1</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 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 2424:2424 -p 5432:5432 \
       --env JAVA_OPTS="-Darcadedb.server.rootPassword=playwithdata \

-Darcadedb.server.plugins=Postgres:com.arcadedb.postgres.PostgresProtocolPlugin " \
          arcadedata/arcadedb:latest
```

The Server output will contain this line:

```
2021-07-08 19:05:06.081 INFO  [ArcadeDBServer] <ArcadeDB_0> - Postgres Protocol 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.

7.11.1. 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()
```

7.11.2. 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.

> Particularly, ArcadeDB does only support "simple" query mode and does not support SSL!

7.11.3. Transactions

Enabling auto commit to false is not 100% supported. With JDBC, leave the default settings or set:

```
conn.setAutoCommit(true);
```

7.11.4. Postgres Tools Known to Work

> Some tools compatible with Postgres may execute queries on internal Postgres tables to retrieve the schema. Those tables are not present in ArcadeDB, so it may return errors at startup. 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. 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.

7.12. Connect with JDBC Driver

Edit this section

If you're using Java you can use the Postgres JDBC driver.

```
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));
      }
    }
  }
}
```

7.13. Open Cypher

Edit this section

ArcadeDB partially supports Open Cypher as query engine. It uses Cypher for Gremlin under-the-hood which has stopped being supported a few years ago. There are a lot of things that work and a lot of nuances which don't.

ArcadeDB also doesn't support Neo4j's BOLT protocol. This means you can't use a Neo4J driver with ArcadeDB server.

To use Cypher queries you can do directly from the Java API, by using HTTP Api or the Postgres driver.

> Consider that Cypher queries are translated into Gremlin. As much as ArcadeDB's Gremlin implementation is optimized, queries run slower than using native SQL or Java API. Based on some internal benchmarks, ArcadeDB's native SQL (SELECT or MATCH) is much faster than Cypher (even 2,000% faster!). The difference is larger with complex queries that work on many records. A simple Cypher MATCH with a lookup with a simple condition will be closer to native SQL performance, but a scan or huge traversal will be much slower with Cypher, because of the underlying usage of Gremlin is not optimized for extreme performance.

Cypher from Java API

In order to execute a Cypher query, you need to include the relevant jars in your class path. To execute a Cypher query, use "cypher" as first parameter in the query method. Example:

```
ResultSet result = database.query("cypher", "MATCH (p:Person) WHERE p.age >= $p1
RETURN p.name, p.age ORDER BY p.age", "p1", 25);
```

You can use ArcadeDB's RecordIDs (RID) in a cypher query to start from a specific vertex. RIDs in Cypher are always strings, therefore they must always be contained between single or double quotes. Example of returning the graph connected to the vertex with RID `#1:0`:

```
MATCH (m:Movie)<-[a:ACTED_IN]-(p:Person) WHERE id(m) = '#1:0' RETURN *
```

Cypher through Postgres Driver

You can execute a Cypher query against ArcadeDB server by using the Postgres driver and prefixing the query with `{cypher}`. Example:

```
"{cypher} MATCH (p:Person) WHERE p.age >= 25 RETURN p.name, p.age ORDER BY p.age"
```

ArcadeDB server will execute the query `MATCH (p:Person) WHERE p.age >= 25 RETURN p.name, p.age ORDER BY p.age` using the Cypher query language.

Cypher through HTTP/JSON

You can execute a Cypher 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/cypher/MATCH (p:Person) WHERE p.age >= 25
```

```
RETURN p.name, p.age ORDER BY p.age"
```

Example of executing a non-idempotent query (updates the database):

```
curl -X POST "http://localhost:2480/command/graph" -d "{'language': 'cypher',
'command': 'MATCH (p:Person) WHERE p.age >= 25 RETURN p.name, p.age ORDER BY p.age'}"
```

Known Limitations with the Cypher Implementation

- ArcadeDB automatically handles the conversion between compatible types, such as strings and numbers when possible. Cypher does not. So if you define a schema with the ArcadeDB API and then you use Cypher 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 Cypher implementation is based on the Cypher For Gremlin Open Source transpiler. This project is not actively maintained by Open Cypher anymore, so issues in the transpiler are hard to fix. Please bear this in mind if you're moving a large project in Cypher into ArcadeDB. The best way to address such issues is to rewrite the faulty cypher query into ArcadeDB SELECT or MATCH statement.

For more information about Cypher:

- Open Cypher
- The Neo4j Cypher Manual
- Learn Cypher in Y Minutes

7.14. Gremlin API

Edit this section

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>24.3.1</version>
</dependency>
```

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','Steve').has('lastName','Jobs').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 steveFriend = graph.traversal().V().has("name","Steve").has("lastName",
"Jobs").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 steveFriend = graph.traversal().V().has("name","Steve").has("lastName"
,"Jobs").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.

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','Steve').has('lastName','Jobs').out('IsFriendOf')"
```

ArcadeDB server will execute the query `g.V().has('name','Steve').has('lastName','Jobs').out('IsFriendOf')` using the Gremlin query language.

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','Steve').has('lastName','J
obs').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\",\"Steve\").has(\"lastName\",\"Jobs\").out(\"
IsFriendOf\")'}"
```

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.

By default, the database "graph" will be available through the Gremlin Server. You can edit the database name or add more databases under the Gremlin Server by editing the file `config/gremlin-server.groovy`

If you're importing a database, use "graph" as the name of the database to be

> available through the Gremlin Server

Start the Gremlin Server with `OpenBeer` as imported database with name `graph`, so it can be used through the Gremlin Server.

```
docker run -d --name arcadeDb -p 2424:2424 -p 2480:2480 -p 8182:8182 \
       --env JAVA_OPTS="-Darcadedb.server.rootPassword=playwithdata \

-Darcadedb.server.defaultDatabases=Imported[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
```

> In case you're running ArcadeDB with Docker, open the port `8182` and use `-e` to pass the settings

To connect to a remote GremlinServer use this:

```
var cluster = Cluster.build()
            .port(8182)
            .addContactPoint("localhost")
            .credentials("root", "password")
            .create();
var connection = DriverRemoteConnection.using(cluster);
var g = new GraphTraversalSource(connection);
```

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.

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

7.14.1. Recommended Tools with Gremlin

If you're using Gremlin with ArcadeDB, check out G.V() graphic tool. It is compatible with ArcadeDB and provides a powerful visual debugger, advanced graph analytics, and much more.

[debugging queries short]

7.15. GraphQL

Edit this section

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>24.3.1</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 } }`

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.

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)
```

@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 Record ID.

Example:

```
{ bookById(id: "book-1")
  {
    rid @rid
    id
    name
    authors {
```

```
      firstName
      lastName
    }
  }
}
```

7.16. MongoDB API

Edit this section

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>24.3.1</version>
</dependency>
```

7.16.1. 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("Heros");

// CREATE A NEW DOCUMENT
database.transaction((tx) -> {
  database.newDocument("Heros").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: 'Heros', 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: 'Heros', query: { $and: [ { name: { $eq: 'Jay' } }, { lastName:
{ $exists: true } }, { lastName: { $eq: 'Miner' } } ] } }"
```

ArcadeDB server will execute the query `{ collection: 'Heros', 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: 'Heros', 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: \"Heros\", query: { $and: [ { name: { $eq: \"Jay\" } }, { lastName: {
$exists: true } }, { lastName: { $eq: \"Miner\" } } ] } }\"}"
```

7.16.2. 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 -p 2424:2424 -p27017:27017 \
       --env JAVA_OPTS="-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]
```

7.17. Redis API

Edit this section

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>24.3.1</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.

7.17.1. 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 2424:2424 -p 6379:6379 \
        --env JAVA_OPTS="-Darcadedb.server.rootPassword=playwithdata \
           -Darcadedb.server.plugins=Redis:com.arcadedb.redis.RedisProtocolPlugin " \
           arcadedata/arcadedb:latest
```

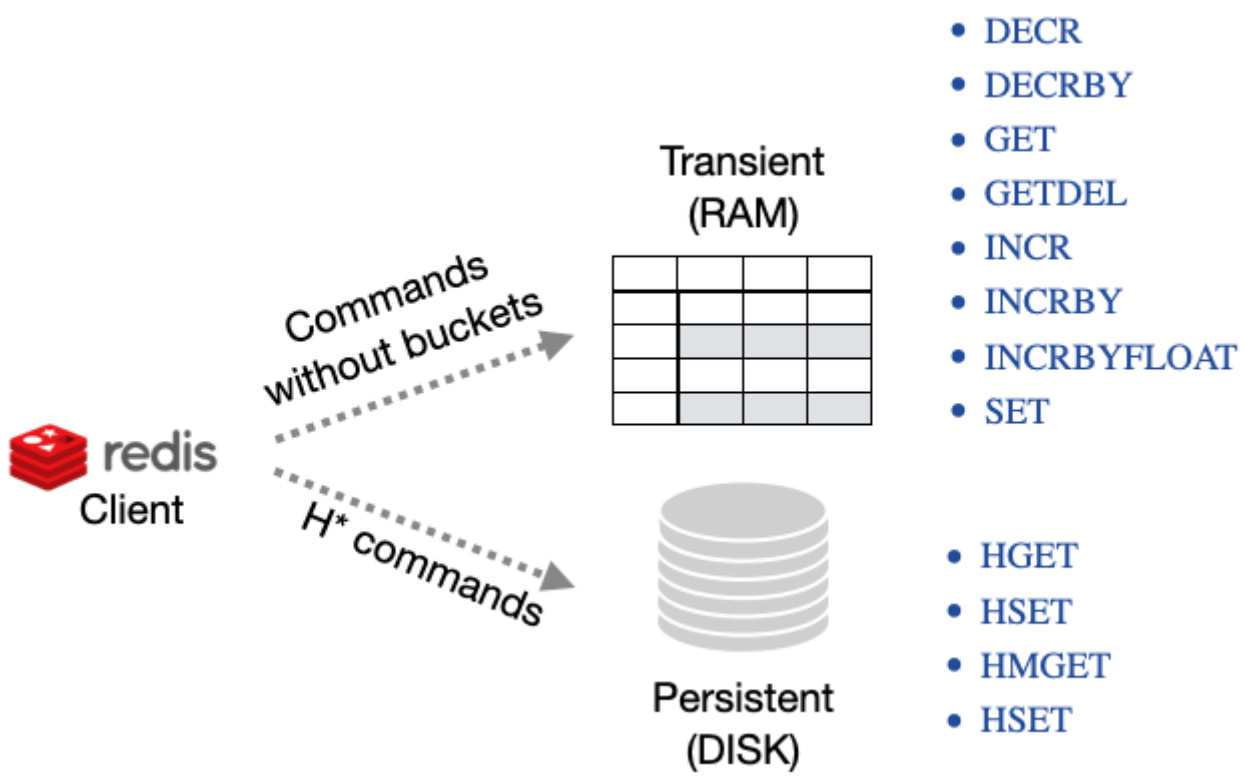
The Server output will contain this line:

```
2018-10-09 18:47:58:395 INFO  <ArcadeDB_0> - Redis Protocol plugin started
[ArcadeDBServer]
```

7.17.2. 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



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.

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

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, 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 `Account[id]` ON Account (id) UNIQUE

CREATE PROPERTY Account.email STRING
CREATE INDEX `Account[email]` ON Account (email) UNIQUE

CREATE PROPERTY Account.firstName STRING
CREATE PROPERTY Account.lastName STRING
CREATE INDEX `Account[firstName,lastName]` ON Account (firstName,lastName) UNIQUE
```

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'}"
```

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'}"
```

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 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
- 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`. 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`.

7.18. Compatible Tools

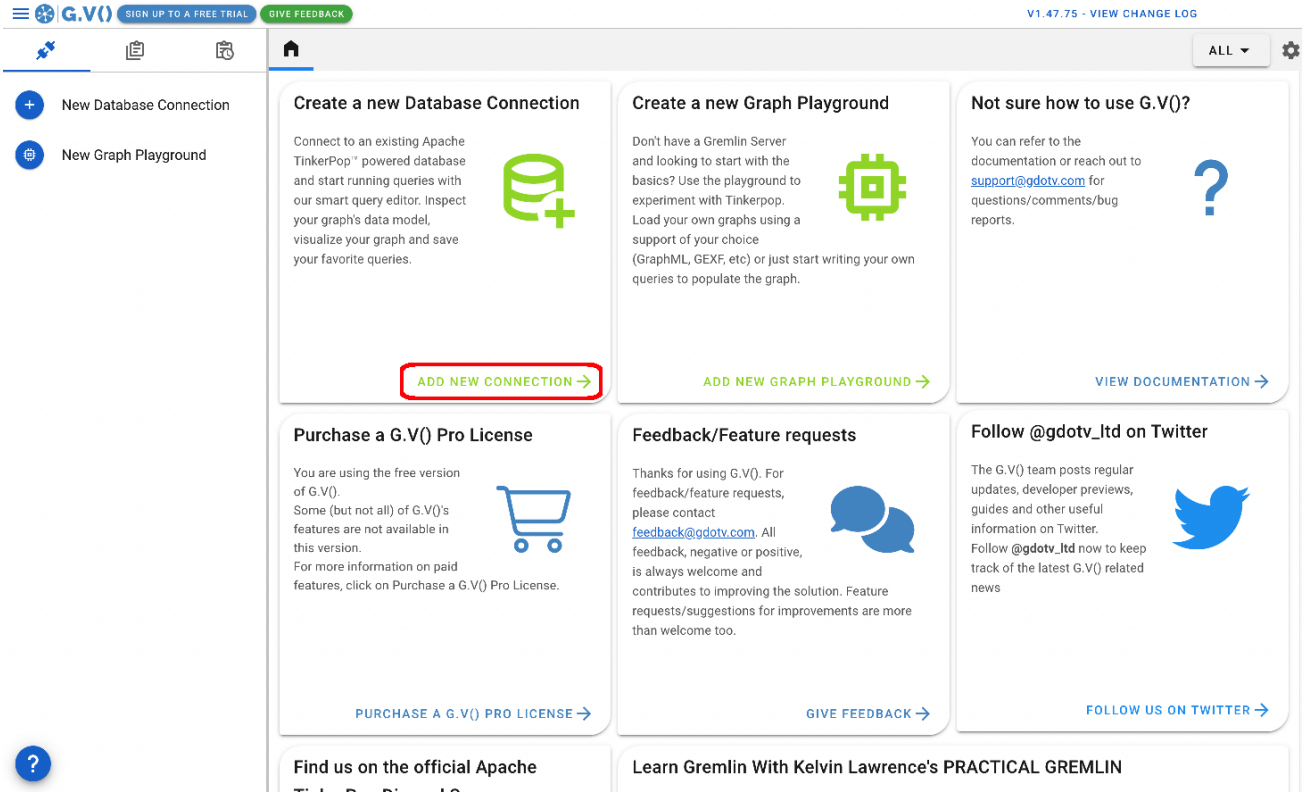
7.18.1. G.V() [gdotv]

ArcadeDB is fully compatible to G.V(), thus no particular configurations need to be made. The only requirement is loading the Gremlin plugin via the server setting:

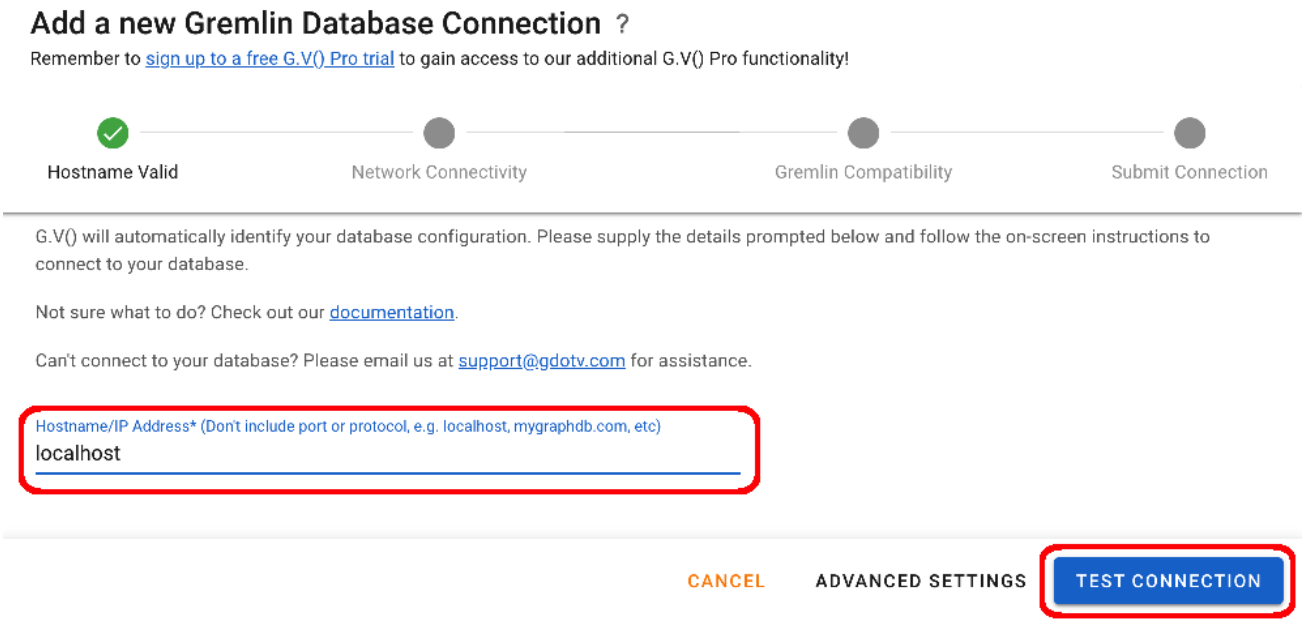
```
"-
Darcadedb.server.plugins=GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugin
"
```

▼ *Details*

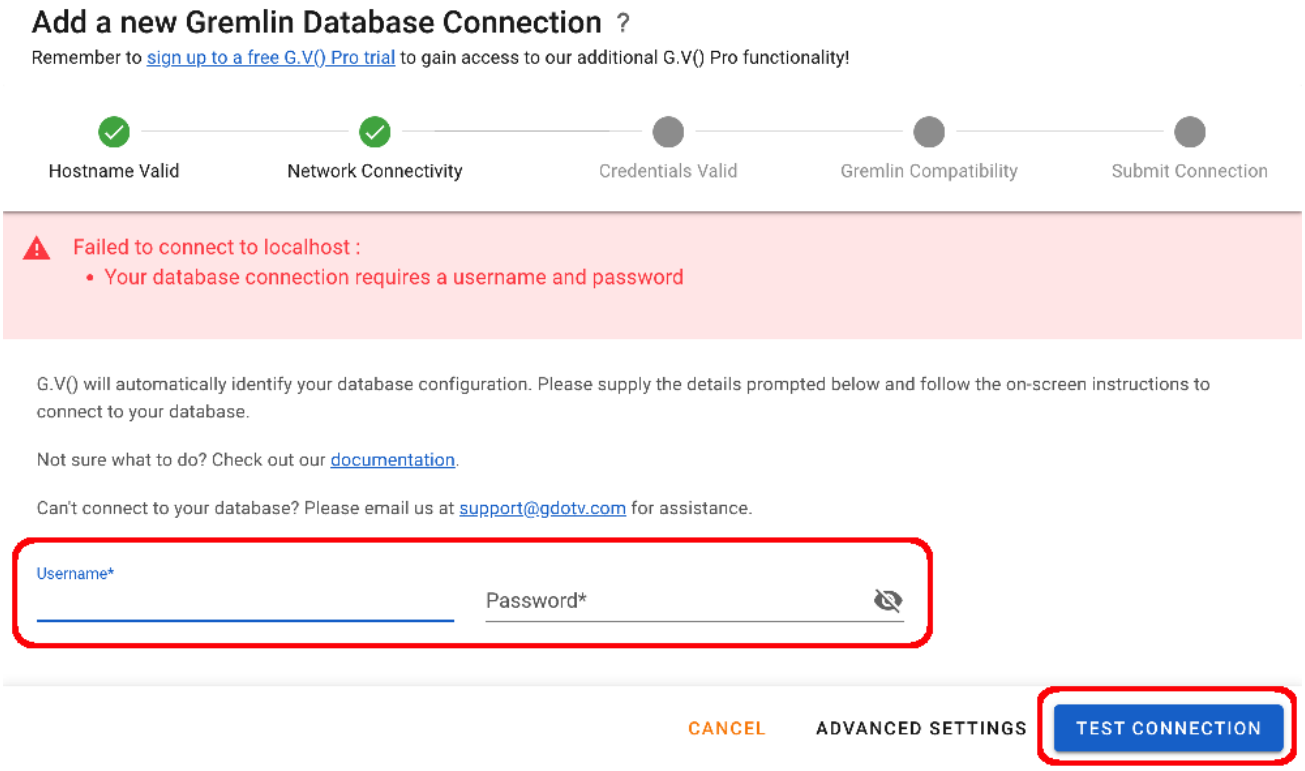
On the title screen, create a new connection:



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.

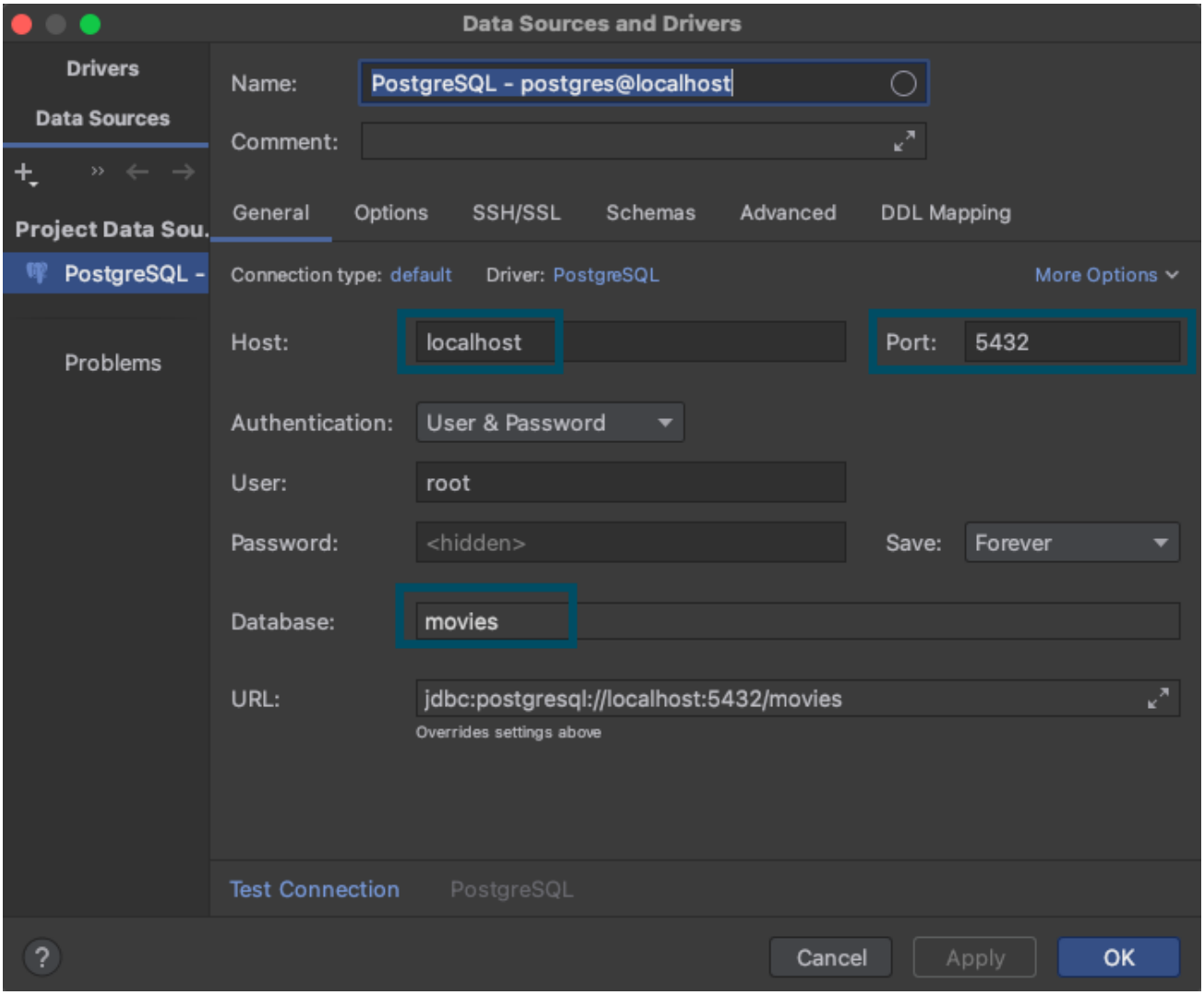
7.18.2. JetBrains DataGrip/Database Plugin

Connecting via JetBrains DataGrip database plugin is relatively straightforward. The introspection features aren't working yet, but the basics seem to work well.

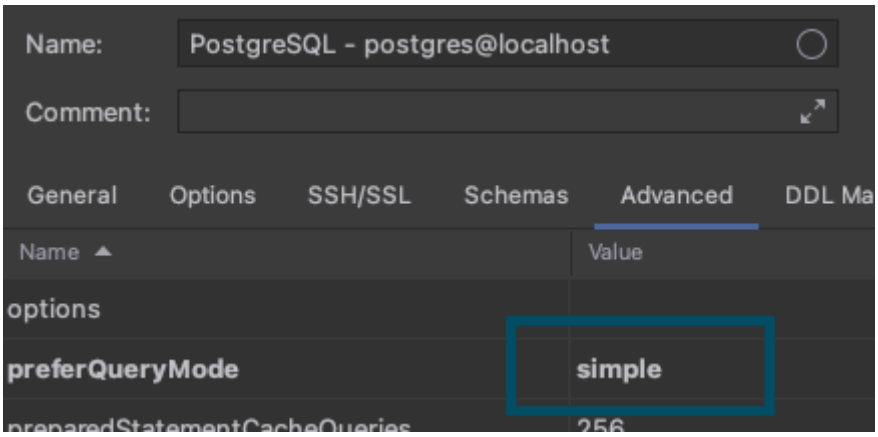
▼ *Details*

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!

console

Tx: Auto movies.<schema> default

```
select from Movie where title like 'The %';

{cypher} match (m:Movie {title: "The Matrix"}) return m;
```

Services

Output Result 5

1 row CSV

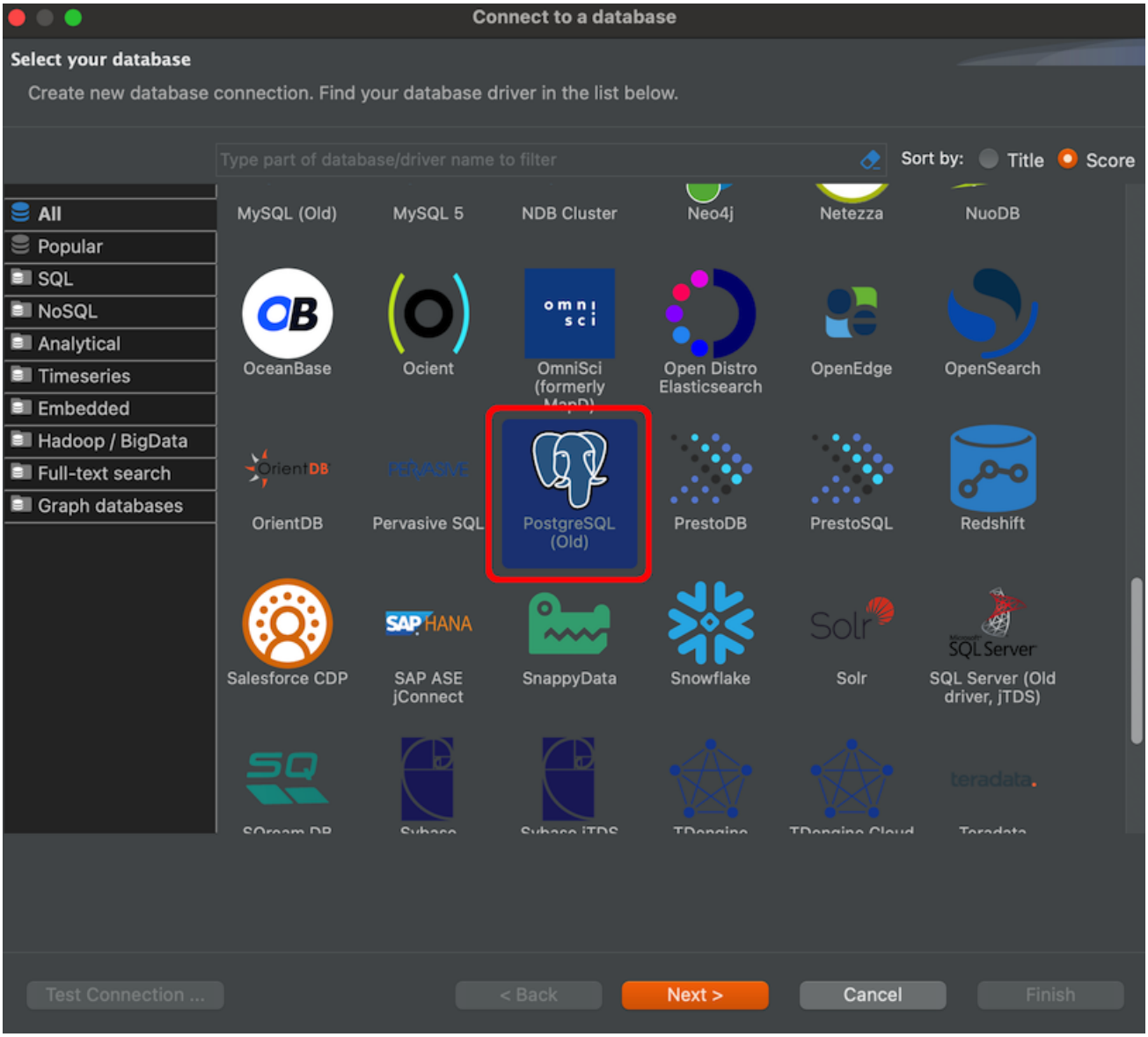
	tagline	@rid	title	@type	released
1	Welcome to the Real World	#1:0	The Matrix	Movie	1999

7.18.3. DBeaver

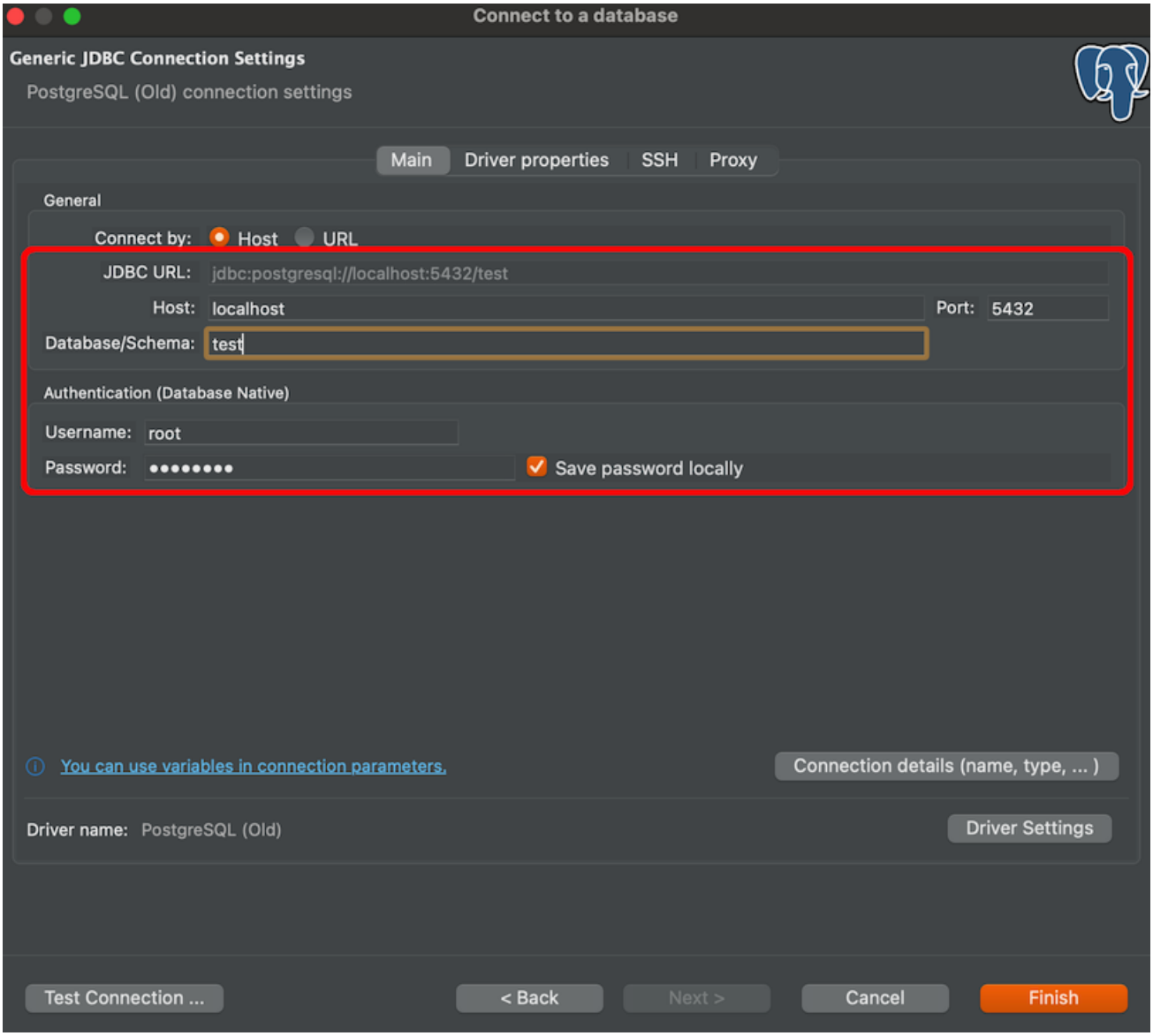
The universal database tool DBeaver has basic compatibility via the legacy Postgres connector.

▼ *Details*

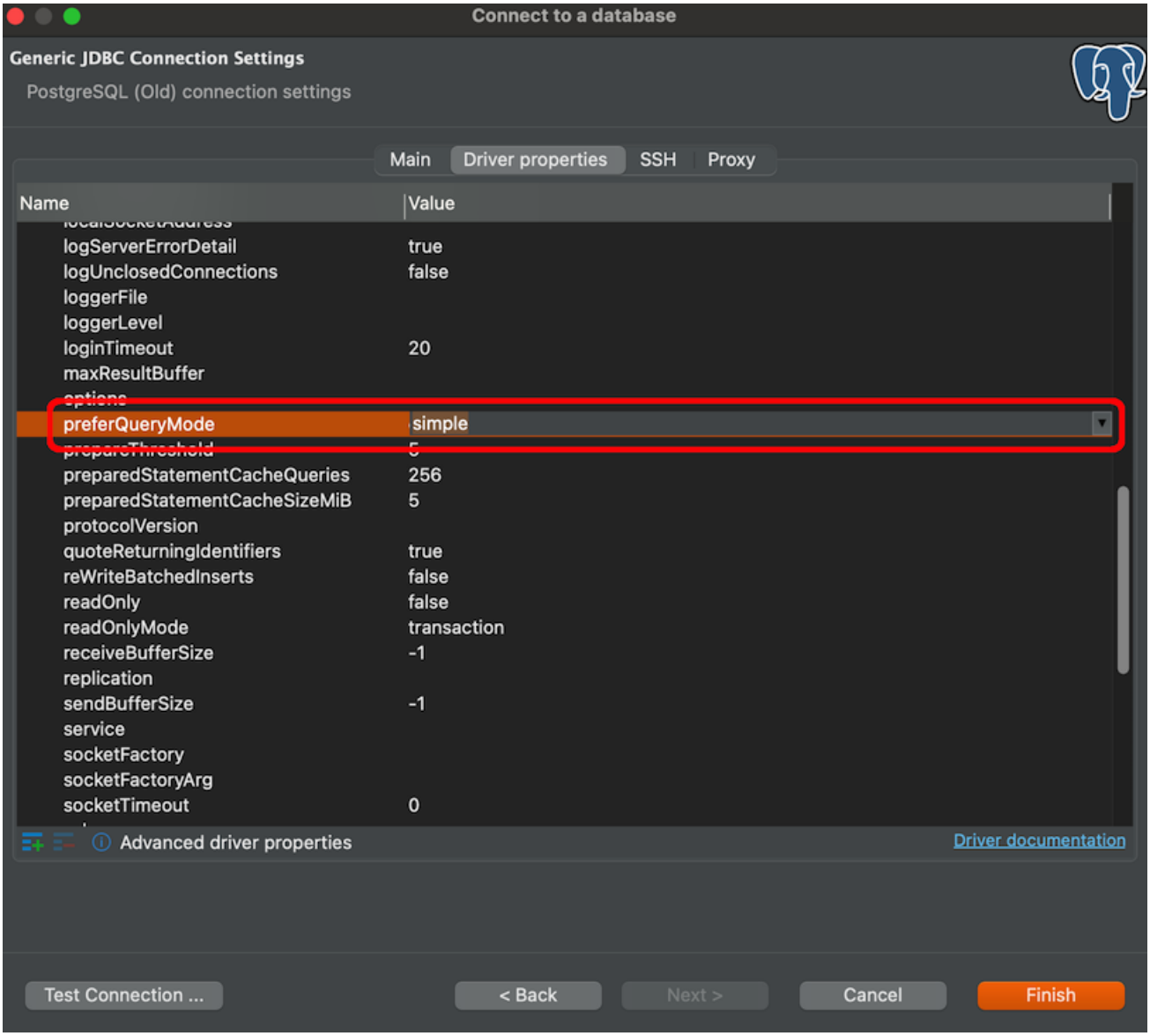
Create a new connection with the "PostgreSQL (Old)" driver:



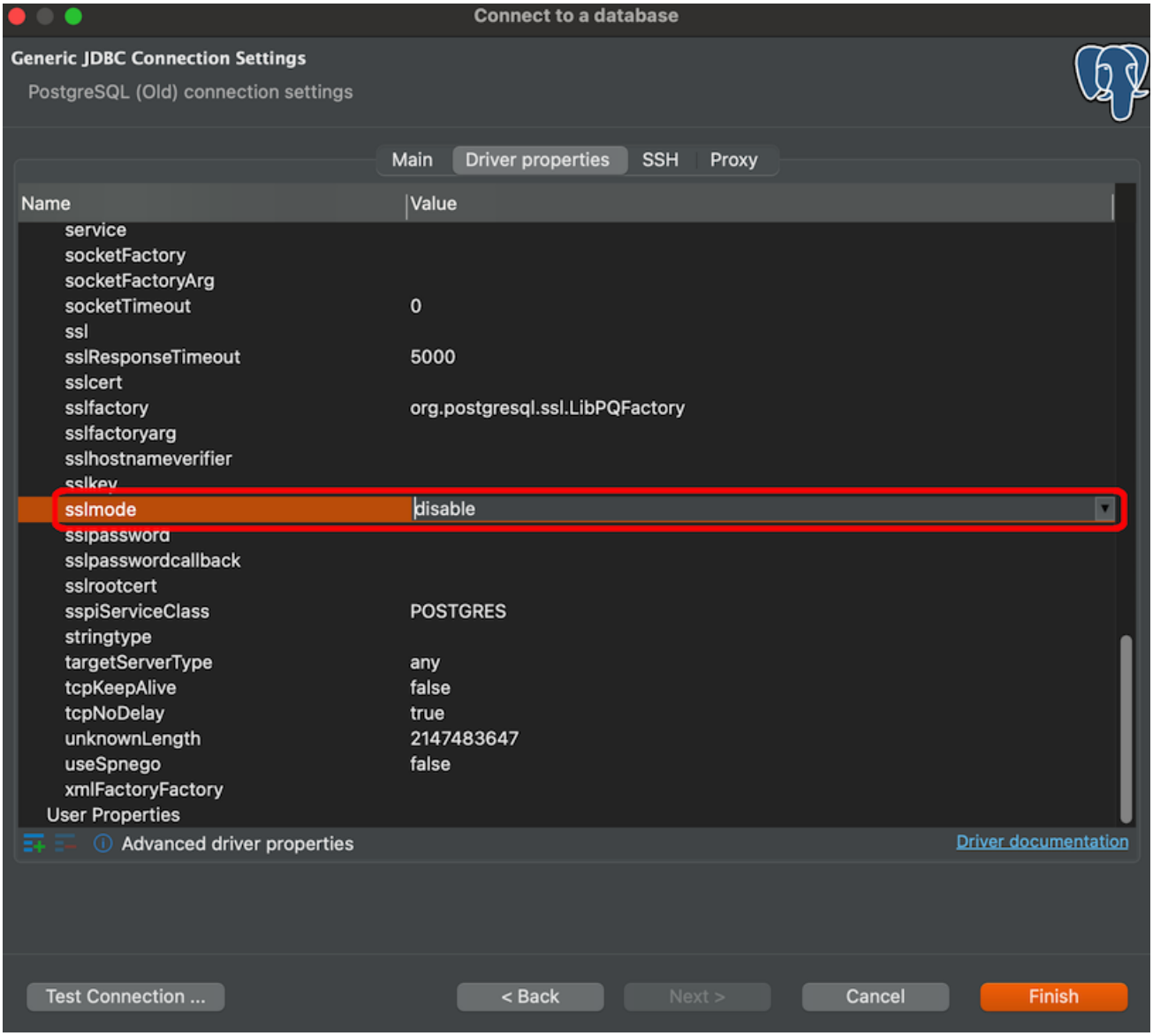
Add your host, port, database, username and password to the general connection settings:



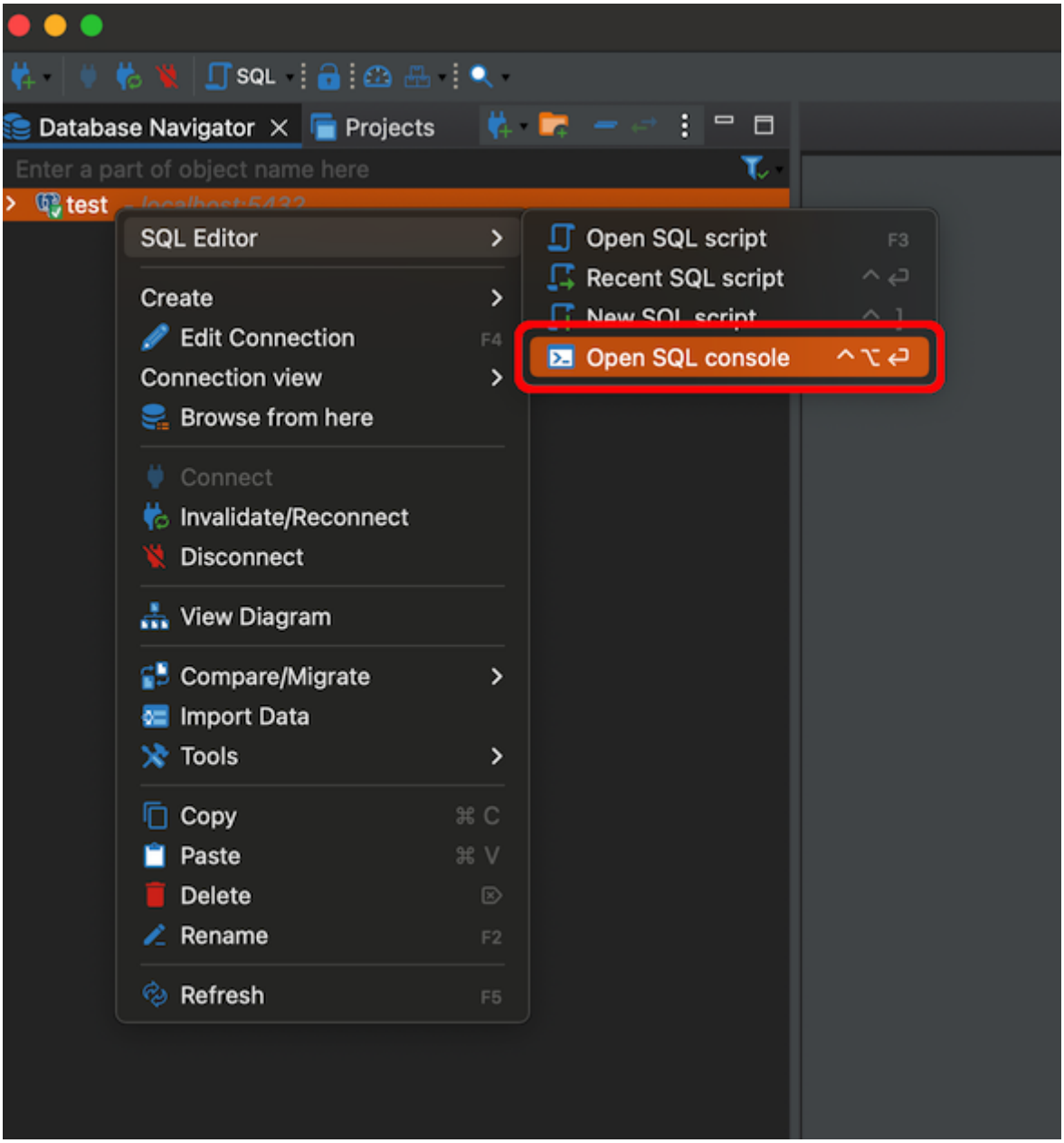
Set the `preferQueryMode` option to `simple` on "Driver Properties" tab:



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.

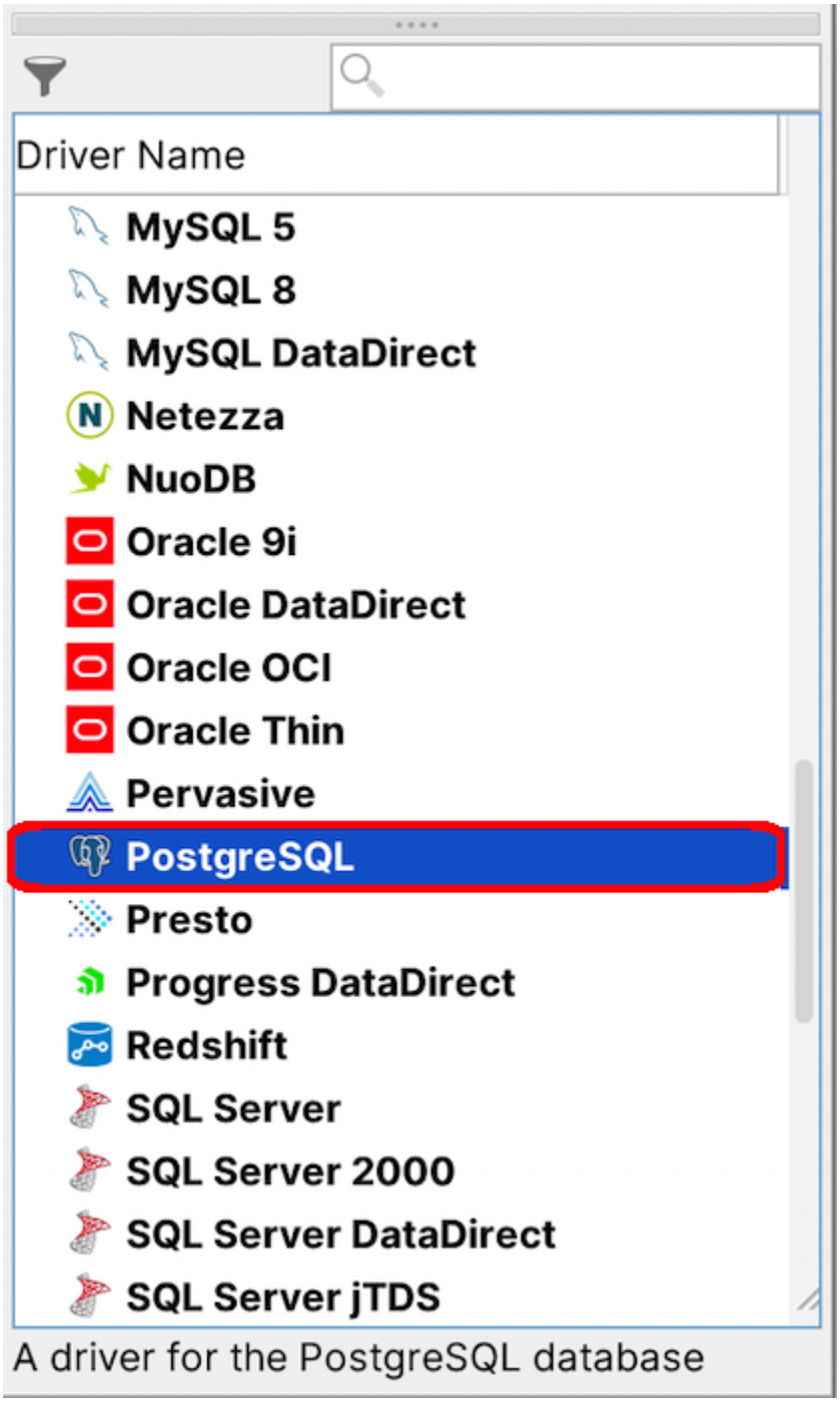
Note that this is only a basic support using a generic relational driver for a NoSQL database, so various functionalities can result in errors.

7.18.4. DbVisualizer

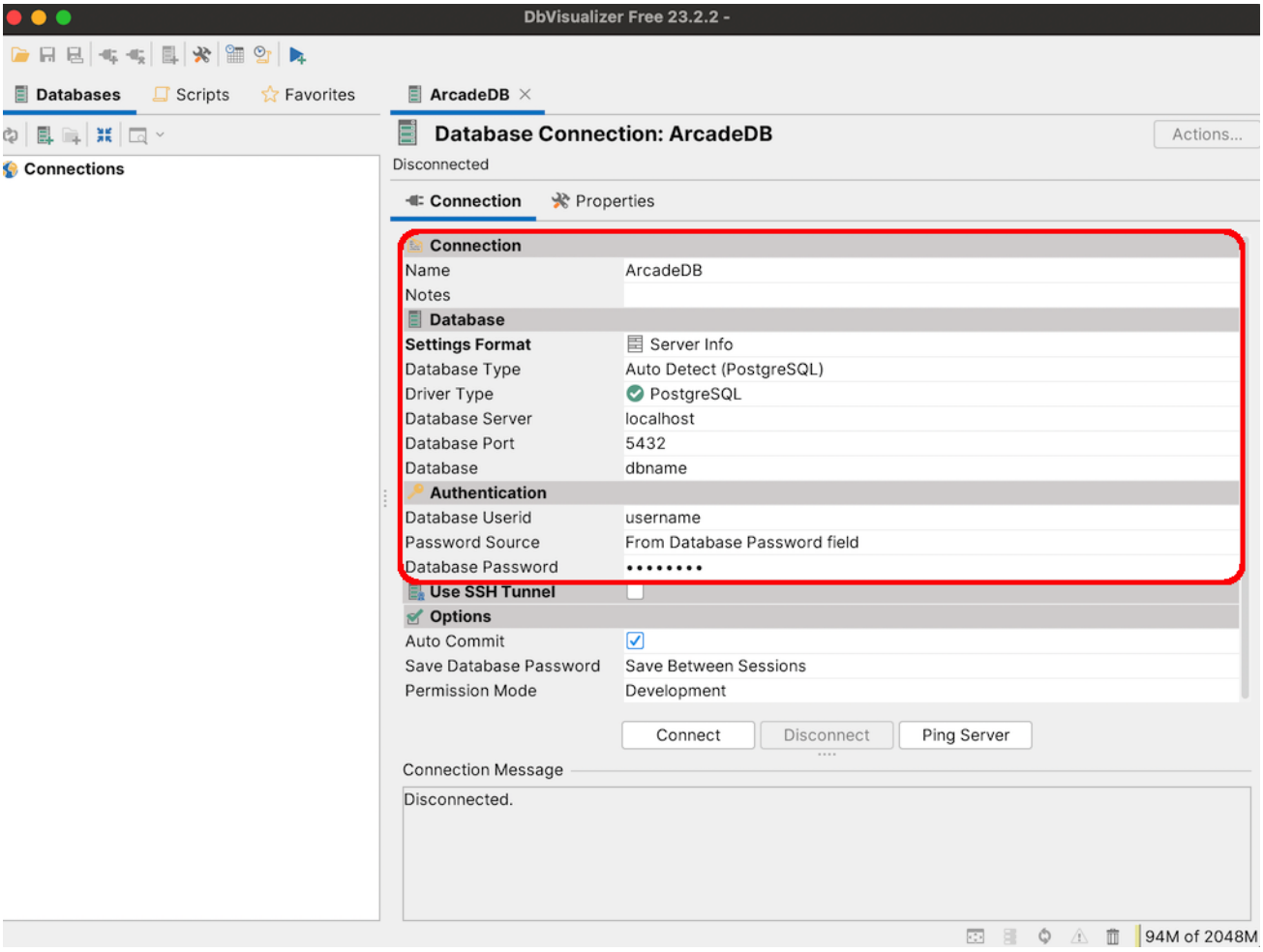
The database client DbVisualizer can also be used via its PostgreSQL driver.

▼ *Details*

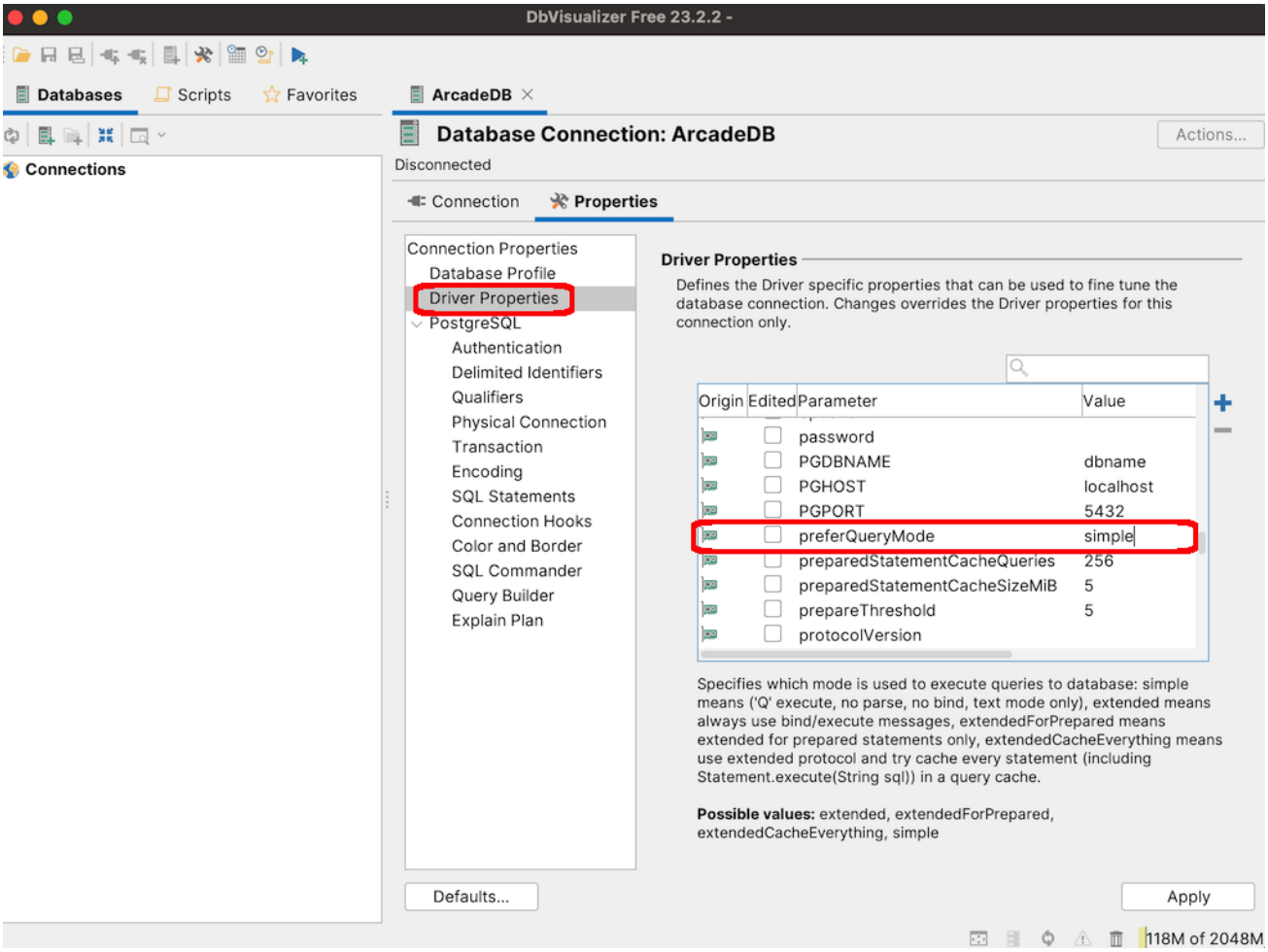
Create a new connection and select "PostgreSQL":



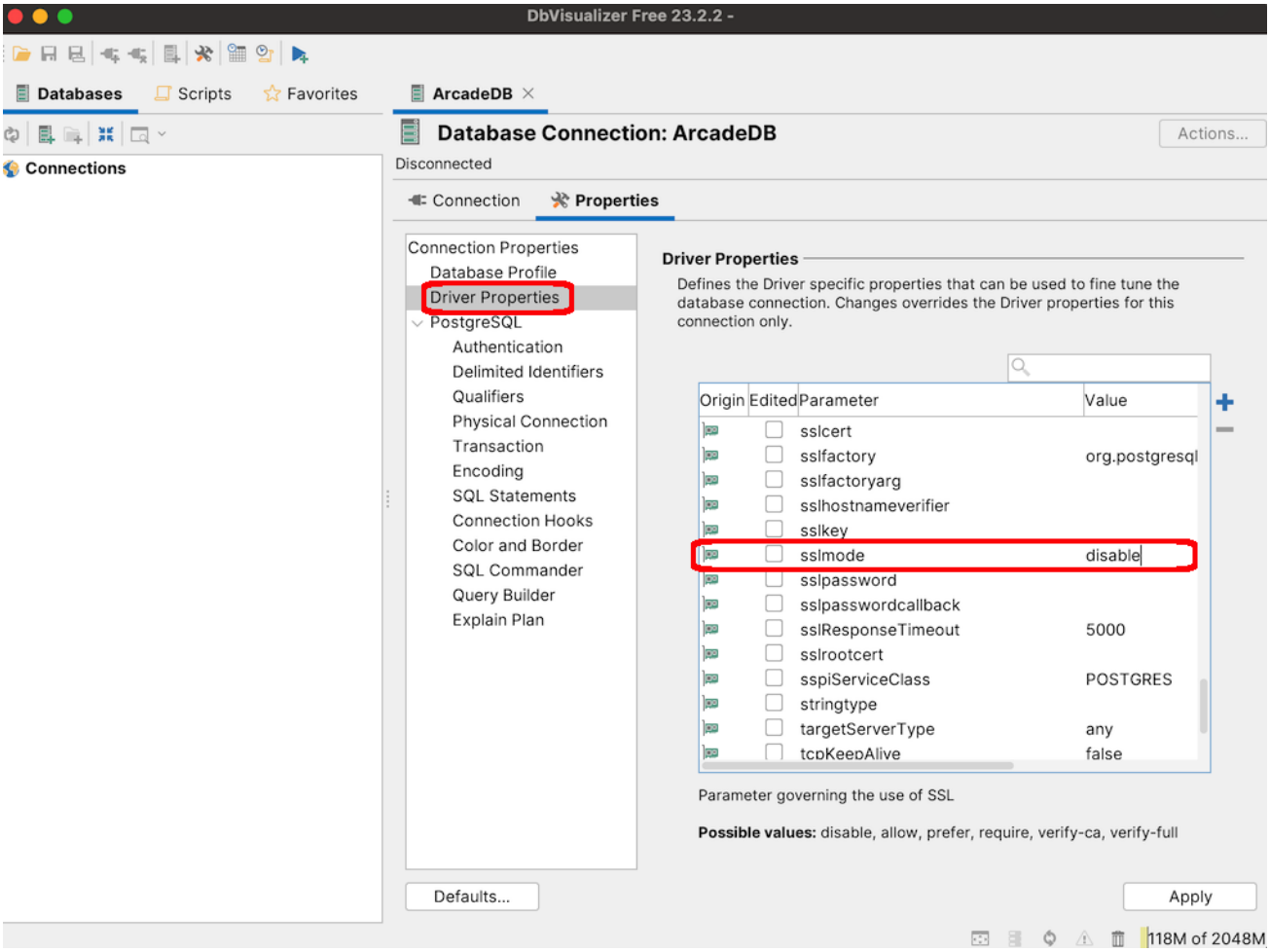
Enter server, port, database, userid, and password:



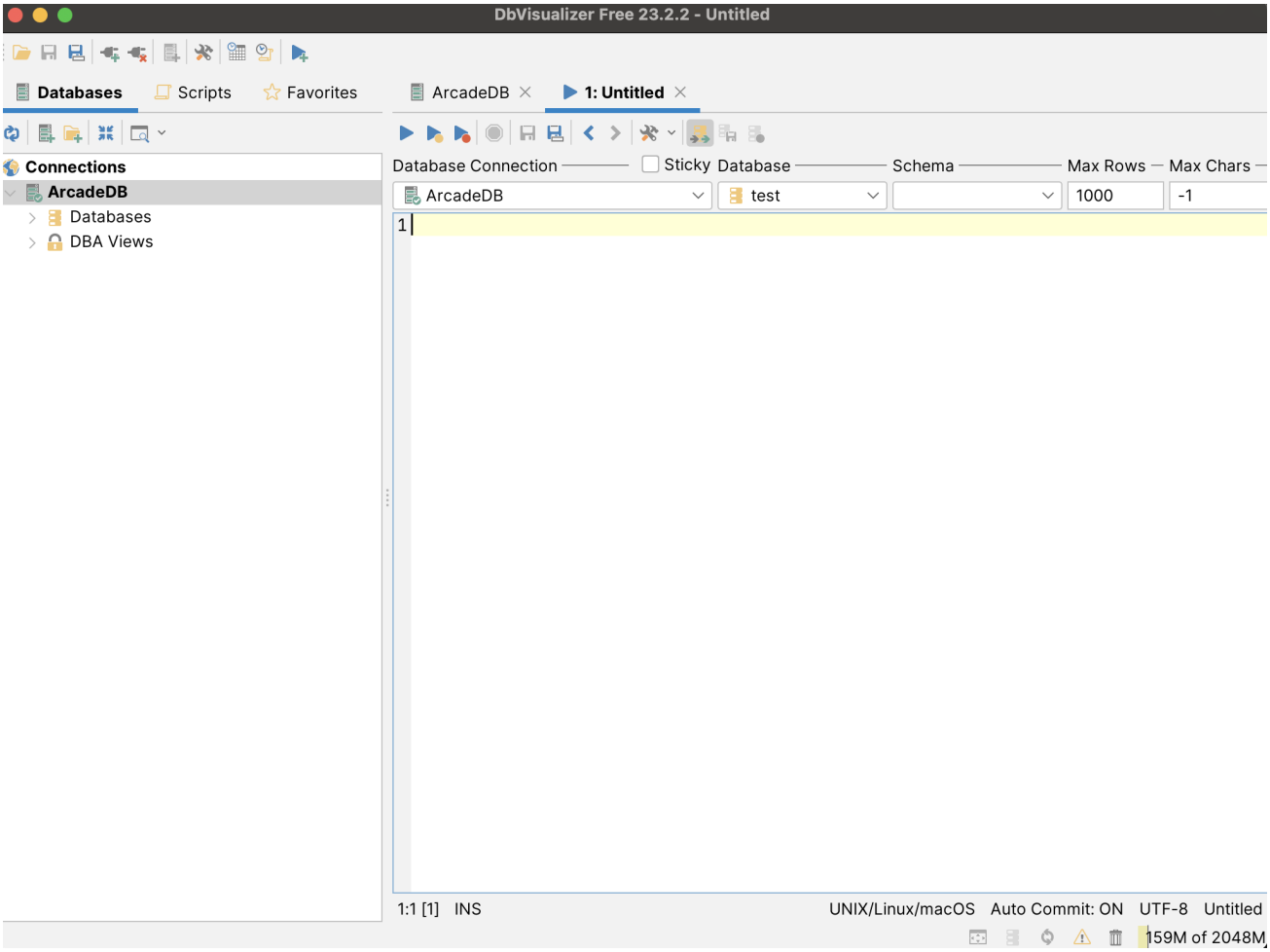
Go to the "Properties" tab and set `preferQueryMode` to `simple`:



Also set `sslmode` to `disable`:



After applying the changes and connecting the SQL commander is available:

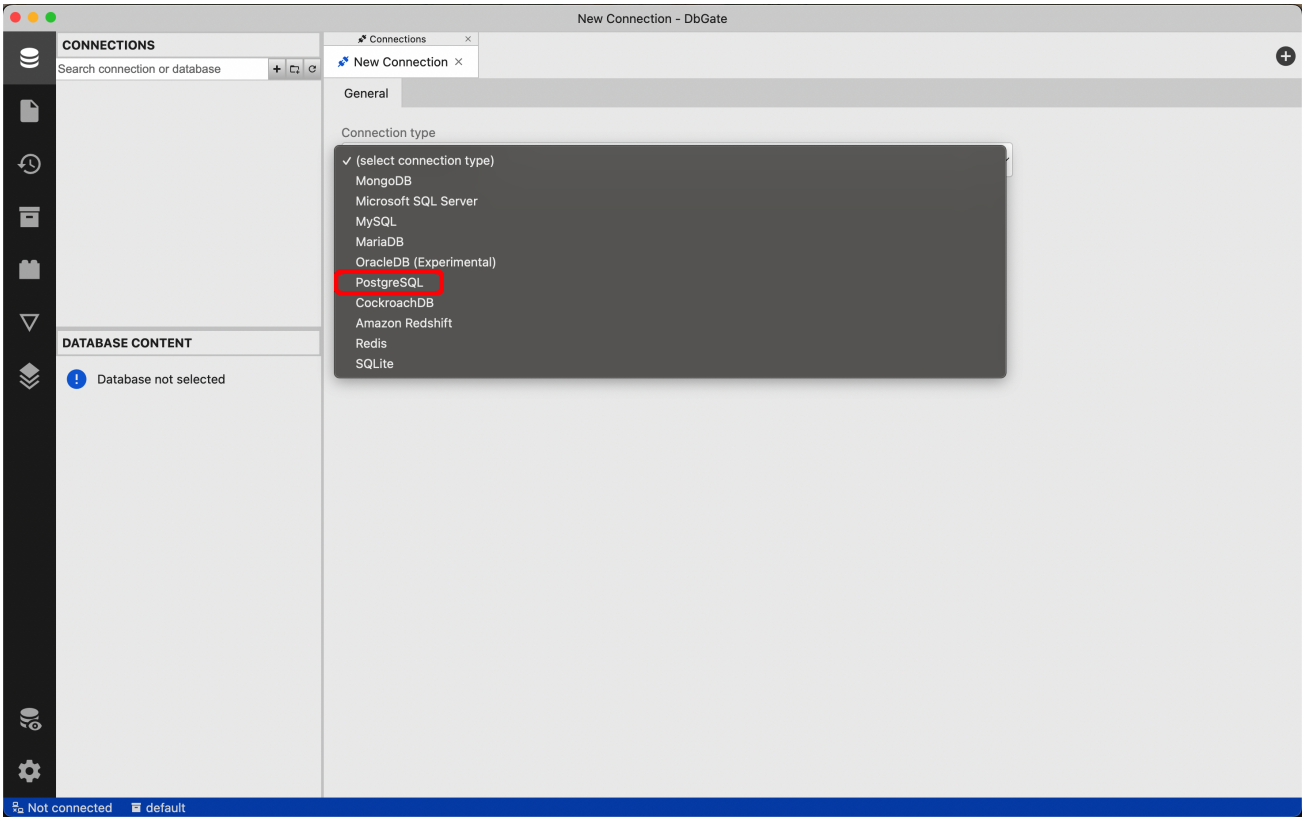


7.18.5. DbGate

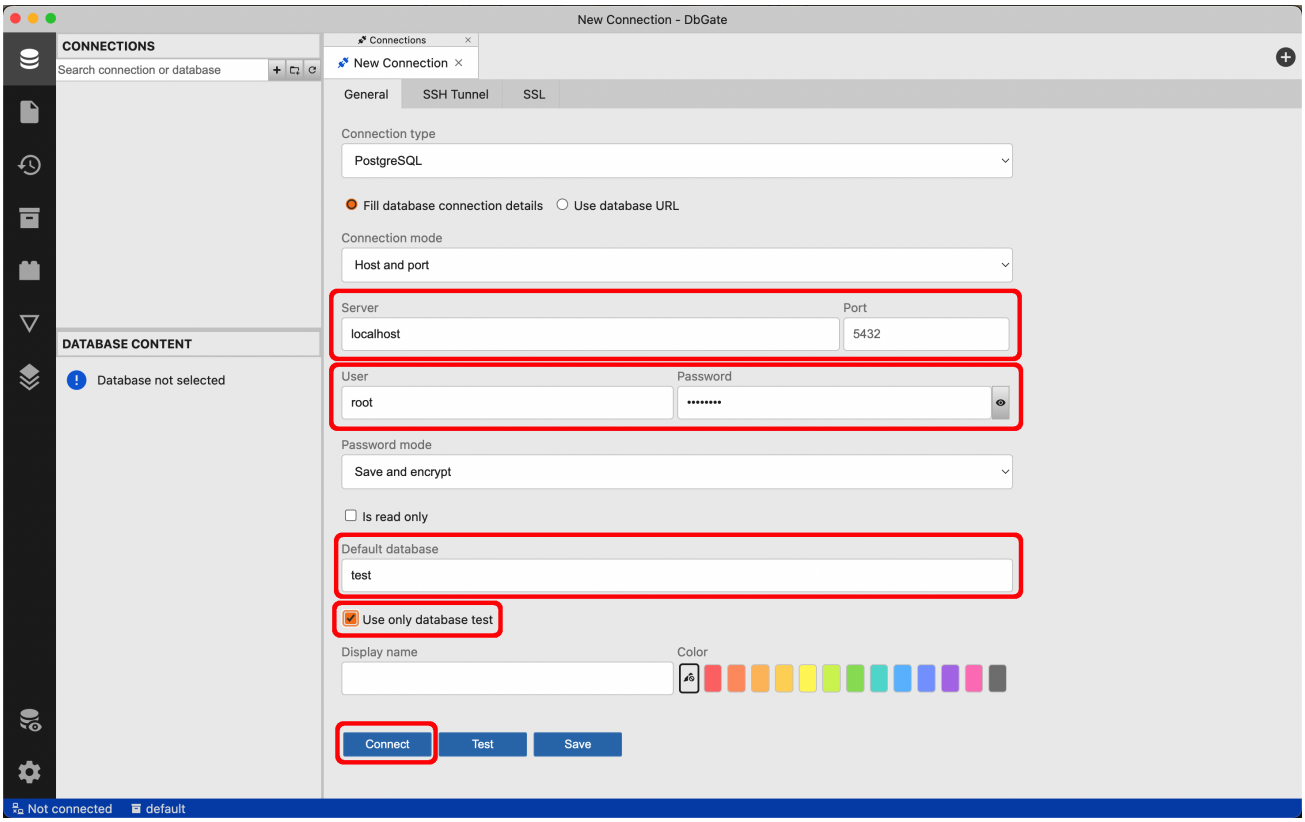
The SQL+noSQL database client DbGate is compatible via the PostgreSQL driver.

▼ *Details*

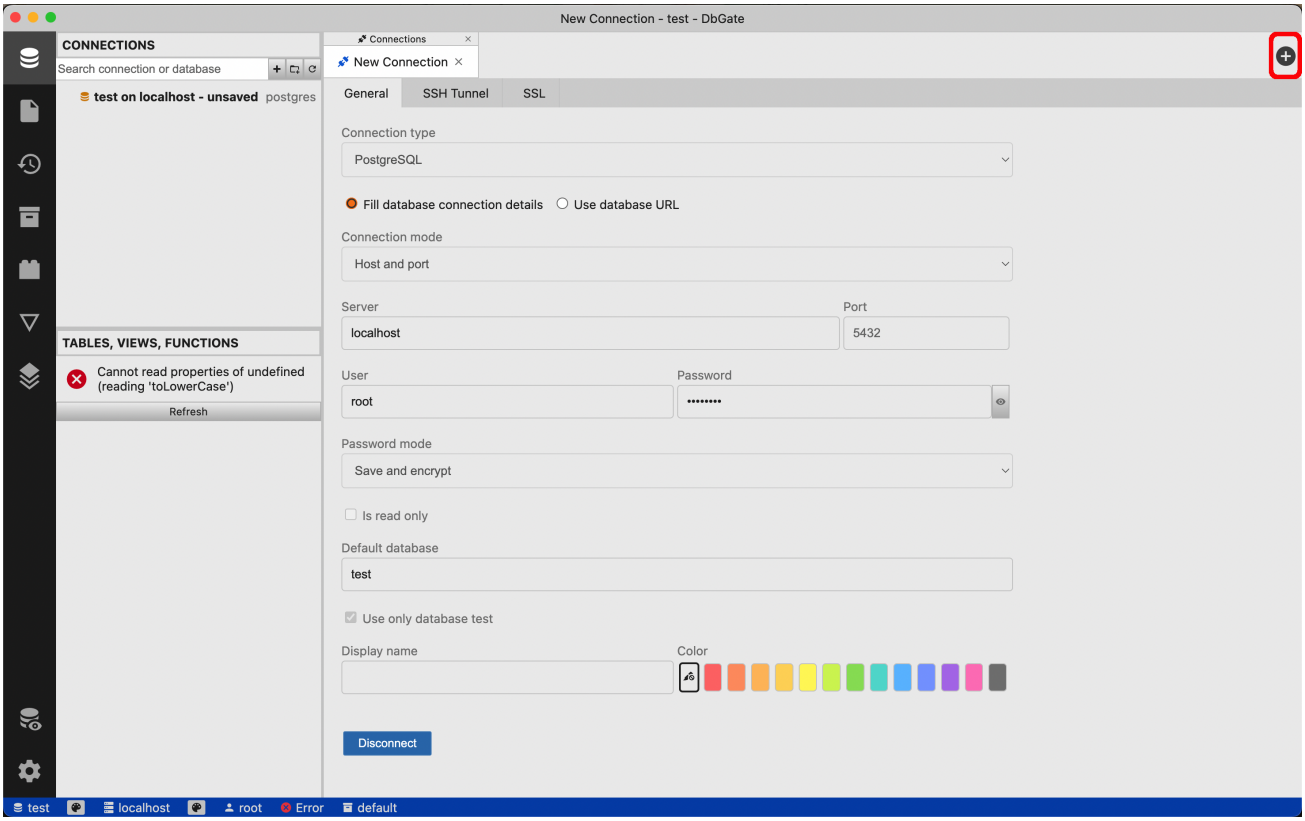
Add a new connection of type PostgreSQL:



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:

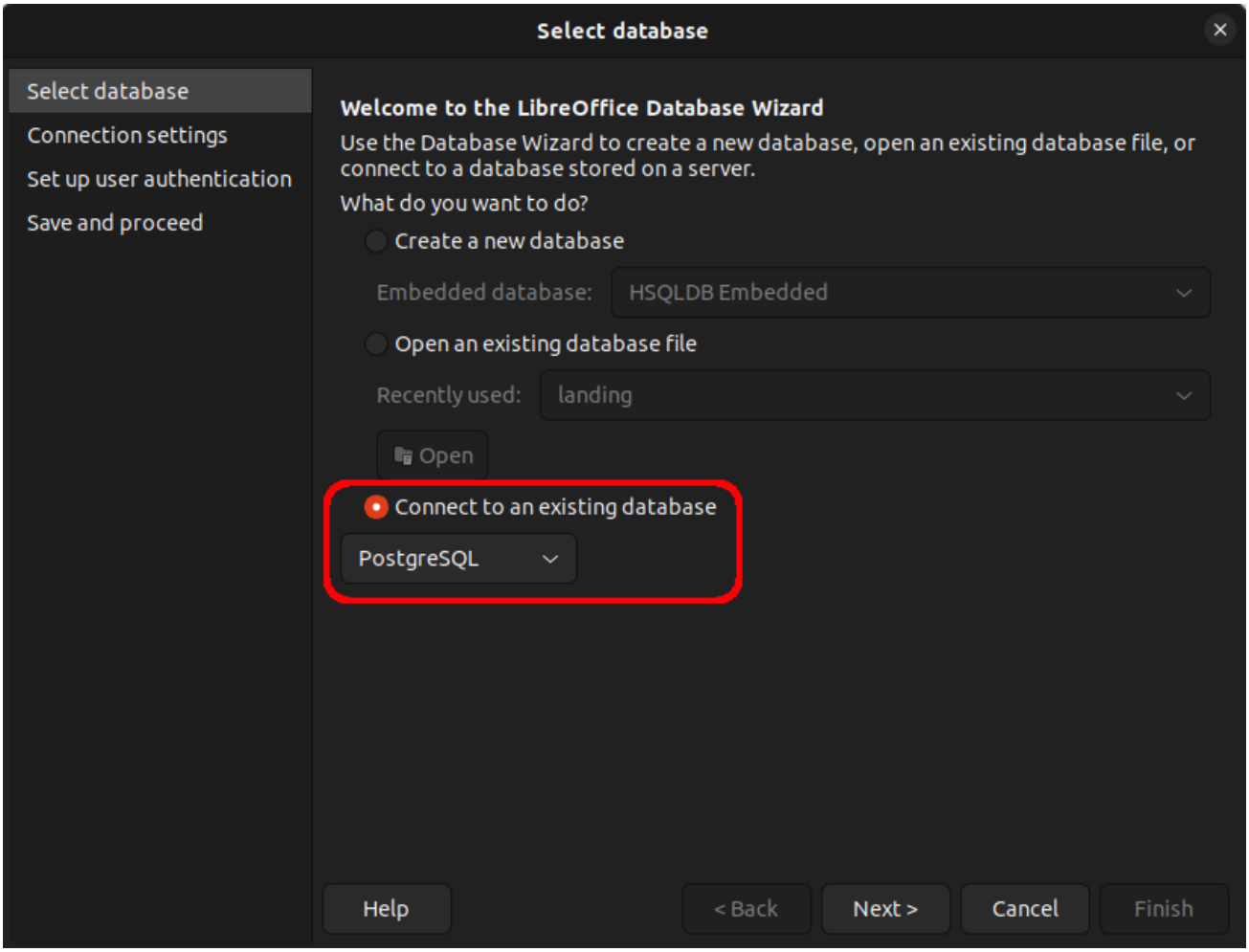


7.18.6. LibreOffice Base

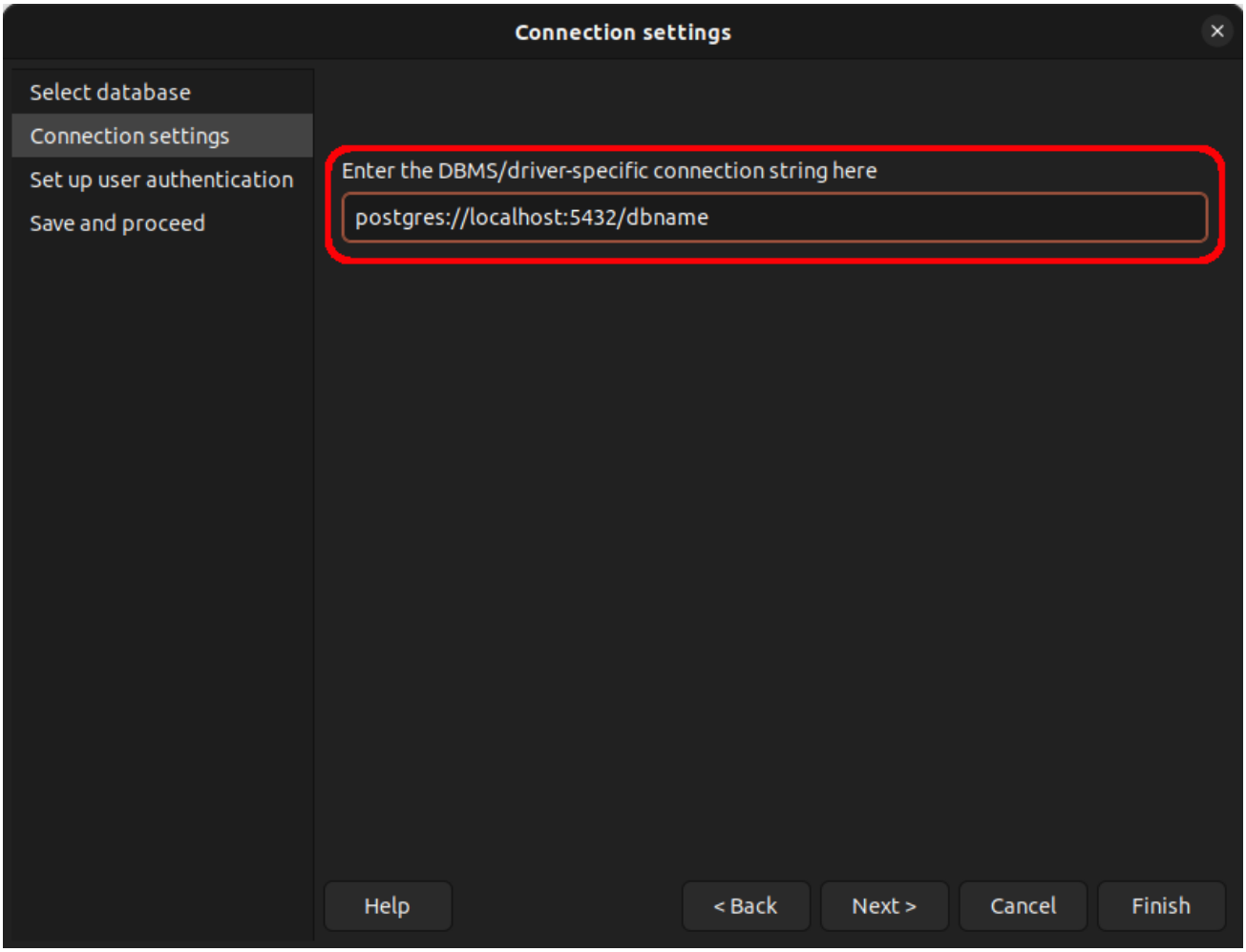
There is minimal support for ArcadeDB in LibreOffice Base via a PostgreSQL connection:

▼ *Details*

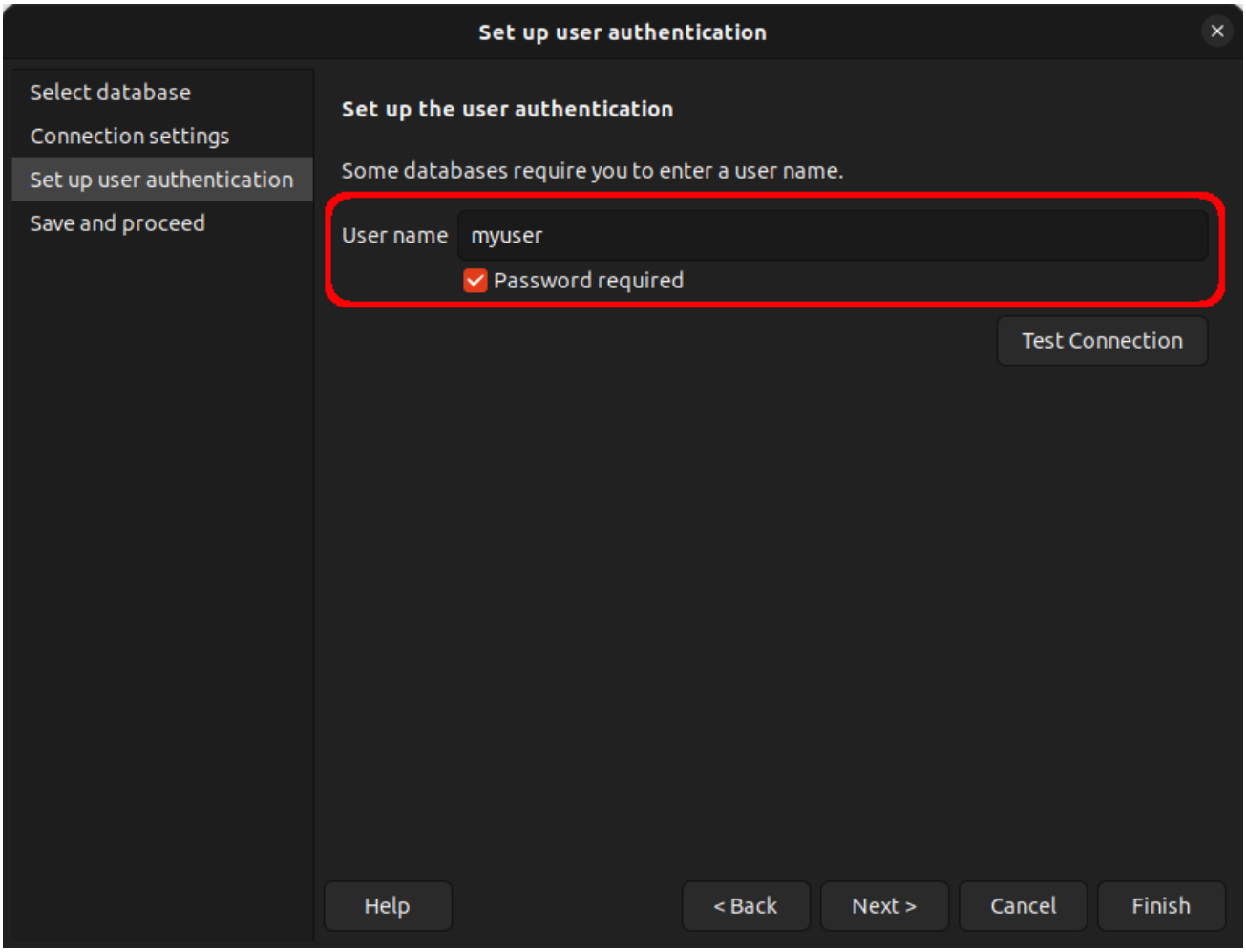
Select "Connect to existing database" and choose "PostgreSQL"



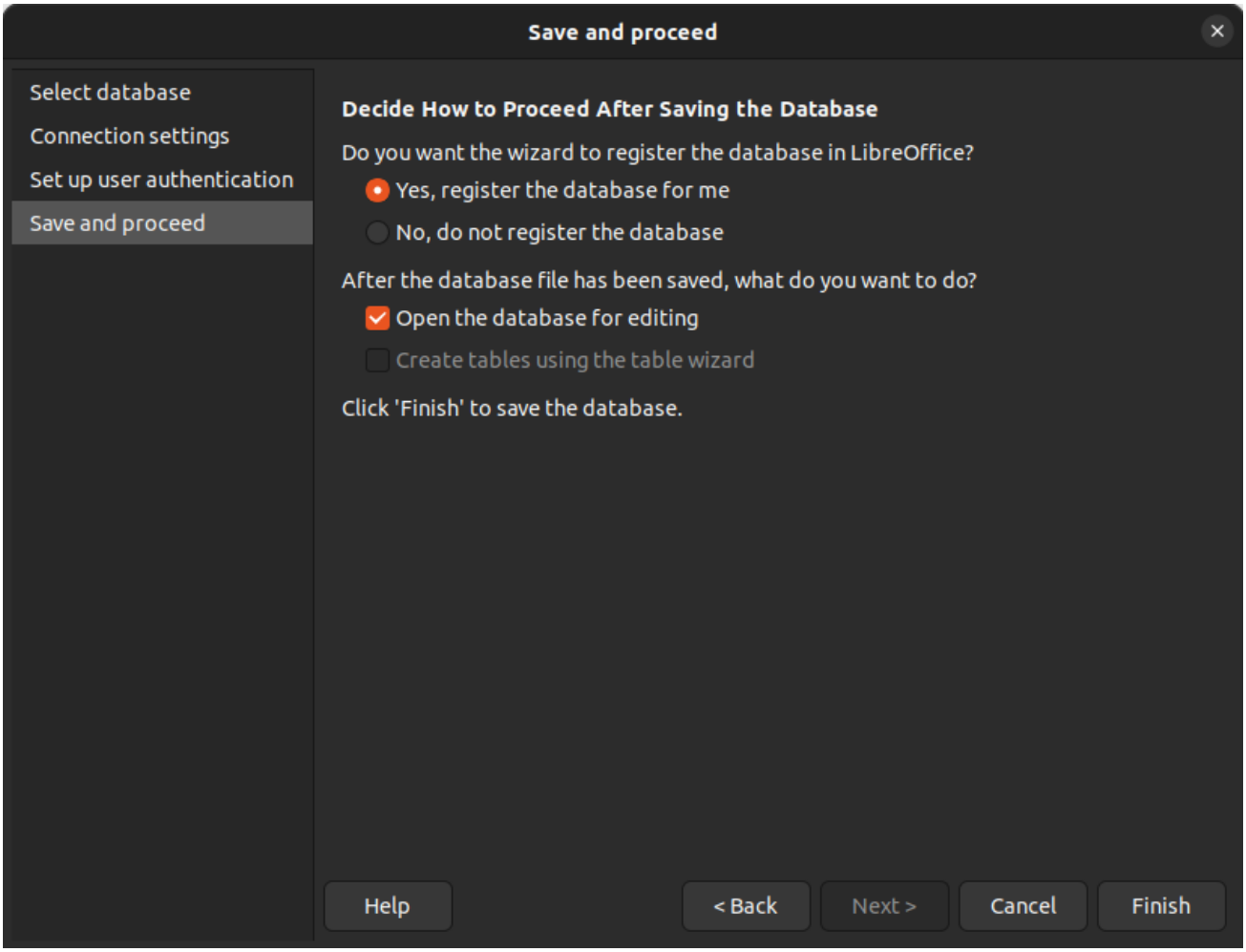
Enter the postgres protocol connection string (without username and password), for example: `postgres://localhost:5432/dbname`



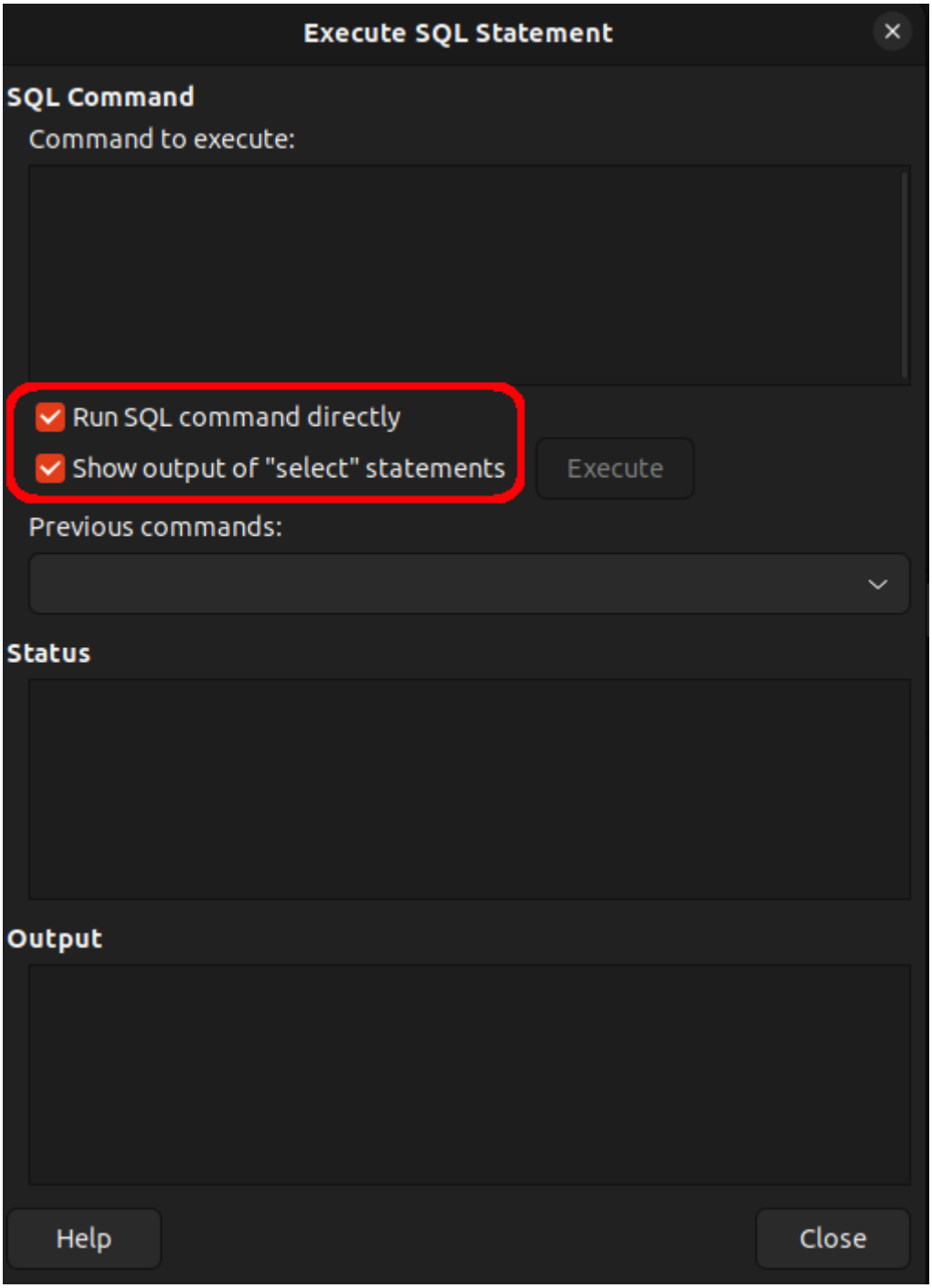
Enter the user name and check that a password is required (try with the "Test Connection" button)



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 ..."

Chapter 8. SQL

Edit this section

Overview Commands

CRUD & Graph	Schema & Buckets	Database & Indexes	Planning & System
SELECT	CREATE TYPE	ALTER DATABASE	EXPLAIN
INSERT	ALTER TYPE	BACKUP DATABASE	PROFILE
UPDATE	DROP TYPE	EXPORT DATABASE	CONSOLE
DELETE	TRUNCATE TYPE	IMPORT DATABASE	
CREATE VERTEX	CREATE BUCKET	ALIGN DATABASE	
CREATE EDGE	ALTER BUCKET (*not implemented*)	CHECK DATABASE	
MOVE VERTEX	DROP BUCKET	CREATE INDEX	
MATCH	TRUNCATE BUCKET	REBUILD INDEX	
TRAVERSE	CREATE PROPERTY	DROP INDEX	
	ALTER PROPERTY		
	DROP PROPERTY		

Overview Functions

Graph	Math	Collections	Geometric	Vector	Misc
out()	count()	set()	point()	vectorNeighbors()	date()
in()	min()	map()	circle()		duration()
both()	max()	list()	rectangle()		sysdate()
outE()	sum()	first()	lineString()		format()
inE()	avg()	last()	polygon()		strcmpci()
bothE()	sqrt()	intersect()	distance()		concat()
outV()	abs()	unionall()			if()
inV()	variance()	distinct()			ifnull()
bothV()	stddev()	difference()			coalesce()
traversedElement()	mode()	symmetricDifference()			uuid()
traversedVertex()	median()				encode()

Graph	Math	Collections	Geometric	Vector	Misc
traversedEdge()	percentile()				decode()
shortestPath()	randomInt()				bool_and()
dijkstra()					bool_or()
astar()					expand()
					version()

Overview Methods

Conversions	String manipulation	Collections	Geometric	Misc
convert()	append()	[]	isWithin()	exclude()
asBoolean()	capitalize()	size()	intersectsWith()	include()
asByte()	charAt()	remove()		type()
asDate()	indexOf()	removeAll()		javaType()
asDateTime()	lastIndexOf()	keys()		toJSON()
asDecimal()	left()	values()		hash()
asDouble()	right()	transform()		ifempty()
asFloat()	prefix()	field()		ifnull()
asInteger()	trim()	sort()		precision()
asList()	replace()			
asLong()	length()			
asMap()	subString()			
asRecord()	toLowerCase()			
asRID()	toUpperCase()			
asSet()	normalize()			
asShort()	split()			
asString()	format()			

Introduction

Edit this section

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
```

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:

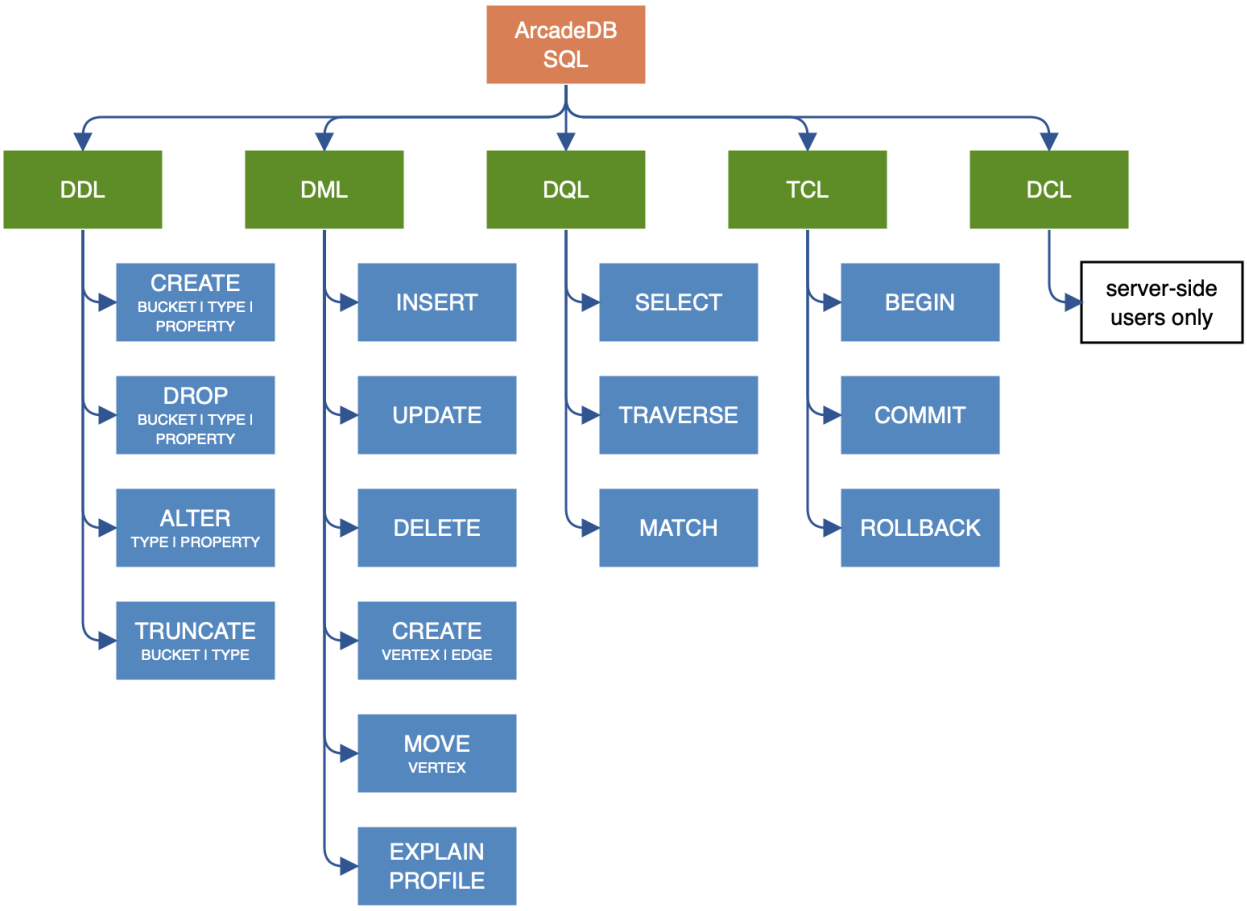
```
SELECT FROM INDEX:myIndex WHERE key = 'Jay'
```

Extra resources

- Syntax
- Projections
- Where clause
- Operators
- Pagination
- SQL Script
- Match for traversing graphs

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.



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`

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'
```

This is much more straight forward and powerful! If you use multiple JOINs, the ArcadeDB SQL equivalent will be an even larger benefit. Example 2: In SQL you might create a join such as:

```
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

Projections

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 `*` is optional:

```
SELECT FROM Customer
```

See Projections.

System Types

To retrieve information about the schema, indexes and the database, there a three special types from which one can "select":

- `schema:types`
- `schema:indexes`
- `schema:database`

these can be treated such as any other types, so projections and filters apply as for any other types.

```
SELECT FROM schema:types
```

DISTINCT

You can use DISTINCT keyword exactly as in a relational database:

```
SELECT DISTINCT name FROM City
```

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 1,000 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
```

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:

```
SELECT expand( $c ) LET $a = ( SELECT FROM E ), $b = ( SELECT FROM V ), $c = unionall(
$a, $b )
```

Projections

Edit this section

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

Query result

By default, a query returns a different result-set based on the projections it has:

- `*` **alone**: The result set is made of records as they arrive from the target, with the original `@rid` and `@type` attributes (if any)
- `*` **plus other projections**: records of the original target, merged with the other projection values, with `@rid` and `@type` of the original record.
- **no projections**: same behavior as `*`
- `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). Nothing in all the other cases.
- **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"}]`

```
SELECT  "hey" AS name, * FROM Foo
```

will return `[{"@type":"Foo", "@rid":"#12:0", "name":"bar"}]`

> 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"}]`

Nested projections

Syntax:

`":{" ( * | (["!"] <identifier> ["*"] (<comma> ["!"] <identifier> ["*"])* ) ) "}"`

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.

Let's clarify this with an 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
   }
}
```

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"
   }
}
```

Pagination

Edit this section

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

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** .

8.1. Filtering

Edit this section

The Where condition is shared among many SQL commands.

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'`

Operators

Conditional Operators

Apply to	Operator	Description	Example
any	`=` or `==`	Equals	`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" and ‘?’ that means "any single character". `LIKE` is case sensitive.	`name LIKE 'Luk%'`
string	`ILIKE`	Similar to `LIKE`, but `ILIKE` is case insensitive.	`name ILIKE 'lUk%'`
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`
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	`name IN ['European','Asiatic']`
collection	`NOT IN`	contains all of the elements not listed	`name NOT IN ['European','Asiatic']`

Apply to	Operator	Description	Example
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	`children CONTAINS (name = 'Luke')`; `map.values() CONTAINS (name = 'Luke')`; `['a','b'] CONTAINS (@this ILIKE 'A')`
collection	`CONTAINSALL`	true if all the elements of the collection satisfy the next condition	`children CONTAINSALL (name = 'Luke')`
collection	`CONTAINSANY`	true if any the elements of the collection satisfy the next condition	`children CONTAINSANY (name = 'Luke')`
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'`
string	`MATCHES`	Matches the string using a Regular Expression	`text MATCHES` `\b[A-Z0-9.%+-@[A-Z0-9.-]\.[A-Z]{2,4}\b]`

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')`

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 * 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 current depth of nesting	TRAVERSE
`$path`	The string representation of the current path. Example: `#6:0.in.#5:0#.out`. You can also display it with `SELECT $path FROM (TRAVERSE * FROM V)`	TRAVERSE

Name	Description	Command(s)
`$stack`	The List of operation in the stack. Use it to access to the history of the traversal	TRAVERSE
`$history`	The set of all the records traversed as a `Set<RID>`	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. `\%`, `\?`.

8.2. Commands

SQL - `ALIGN DATABASE`

Edit this section

Executes a distributed alignment of the database. It must be executed on the Leader server. The alignment computes a checksum of each file and sends them to the replica nodes. Each replica node will compute the checksum on its own files. The files that are mismatching are requested by the replica to the leader. In the future single pages could be transferred instead of the entire file.

> Align Database command is available only when the server is running with the HA module active.

Syntax

```
ALIGN DATABASE
```

The command returns which page have been aligned on each server.

Examples

- Align the current database.

```
ArcadeDB> ALIGN DATABASE
```

SQL - ALTER DATABASE

Edit this section

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

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
```

SQL - ALTER PROPERTY

Edit this section

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>]
```

- `<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
  - `mandatory <true|false>` If true, the property must be present. Default is false
  - `notnull <true|false>` If true, the property, if present, 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
  - `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
  - `default <any>` Defines default value if not present. Default is `null`.
- **`<custom-key>`** Name of the custom property to set
- **`<custom-value>`** Value for the custom property. All data types are supported as values.

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'
```

For more information, see:

- `CREATE PROPERTY`
- `DROP PROPERTY`
- `ALTER TYPE`

SQL - ALTER TYPE

Edit this section

Change a type defined in the schema. The change is persistent.

Syntax

```
ALTER TYPE <type> [<attribute-name> <attribute-value>] [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.

Examples

- Add `Person` to the super types:

```
ArcadeDB> ALTER TYPE Employee SUPERTYPE +Person
```

- Remove a super-type:

```
ArcadeDB> ALTER TYPE Employee SUPERTYPE -Person
```

- 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 a bucket from the type `Account` with the ID `34`:

```
ArcadeDB> ALTER TYPE Account BUCKET -34
```

- 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]};
```

For more information, see:

- `CREATE TYPE`
- `DROP TYPE`.

Supported Attributes

Attribute	Type	Support	Description
`NAME`	Identifier		Changes 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 or Integer		`+` 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 reccomended when your have a unique id

SQL - `BACKUP DATABASE`

Edit this section

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. ArcadeDB allows to execute a non-stop backup of a database while it is used without blocking writes or affecting performance.

Syntax

```
BACKUP DATABASE [ <backup-file-url> ]
```

- `<backup-file-url>` Optional, defines the location for the backup archive. If not specified, the backup file will be `backups/<db-name>/<db-name>-backup-<timestamp>.tgz`, where the timestamp is expresses from the year to the millisecond. Example of backup file name `backups/TheMatrix/TheMatrix-backup-20210921-172750767.zip`.

Examples

- Execute the backup of the current database with the default filename.

```
ArcadeDB> BACKUP DATABASE
```

SQL - CHECK DATABASE

Edit this section

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.

Syntax

```
CHECK DATABASE [ TYPE <type-name>[,]* ] [ BUCKET <bucket-name>[,]* ] [ FIX ]
```

- `<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
- `FIX` Optional, if defined auto fix the issue found with the check

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.
`corruptedRecords`	all the records not found in the database. This could happen with broken edges (edges pointing to deleted vertices). 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.
`totalPages`	total number of pages in the database.
`rebuiltIndexes`	list of indexes that have been rebuilt automatically if corrupted records were found.
`operation`	can be ignored, set by the console.
`invalidLinks`	similar to corrupted records; should be zero.

Field	Description
`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.

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
```

SQL - `CONSOLE`

Edit this section

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.

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')
```

SQL - `CREATE BUCKET`

Edit this section

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> [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.
- `<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`

SQL - `CREATE EDGE`

Edit this section

Creates a new edge in the database.

Syntax

```
CREATE EDGE <type> [BUCKET <bucket>] [UPSERT] FROM <rid>|(<query>)|[<rid>]* TO <rid>|
(<query>)|[<rid>]*
                   [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 in another edge already exists with the same direction (same from/to) and same edge type.
- `UPSERT` allows to skip the creation of edges that already exist between two vertices (ie. a unique edge for a couple of vertices). This works only if the edge type has a UNIQUE index on `out, in` fields, otherwise the statement fails.
- `JSON` provides JSON content to set as the record. Use this instead of entering data field by field.
- `RETRY` define 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 are created ArcadeDB throws a `OCommandExecutionException` 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.

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 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
```

For more information, see:

- `CREATE VERTEX`

SQL - `CREATE INDEX`

Edit this section

Creates a new index. Indexes can be:

- **Unique** Where they don't allow duplicates.
- **Not Unique** Where they allow duplicates.
- **Full Text** Where they index any single word of text.
- **HNSW** (Hierarchical Navigable Small World) vector index.

> 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>*) ]
<index-type> [<key-type>]
[ NULL_STRATEGY SKIP|ERROR]
```

- `<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)
- `<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.
- `<index-type>` Defines the index type you want to use:
  - `UNIQUE` does not allow duplicate keys,
  - `NOTUNIQUE` allows duplicate keys,
  - `FULL_TEXT` based on any single word of text.
  - `HNSW` vector index.
- `<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.

Null values are not indexed, so any query that is looking for null values will not use the index with a full scan.

A unique index does not regard or derived types or embedded documents of the indexed type.

Full-text indexes do not support multiple properties.

> 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 automatic indexes for the `thumbs` property in the type `Movie`:

```
ArcadeDB> CREATE INDEX ON Movie (thumbs) UNIQUE
ArcadeDB> CREATE INDEX ON Movie (thumbs BY KEY) UNIQUE
ArcadeDB> CREATE INDEX ON Movie (thumbs BY VALUE) UNIQUE
```

- 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
```

- Also full text index can be set up:

```
ArcadeDB> CREATE INDEX ON Employee (address) FULL_TEXT
```

which can be searched with:

```
ArcadeDB> SELECT FROM `Employee[address]` WHERE address LIKE '%New York%'
```

- Create a manual index to store dates:

```
ArcadeDB> CREATE INDEX `mostRecentRecords` UNIQUE DATE
```

For more information, see:

- `DROP INDEX`

SQL - `CREATE PROPERTY`

Edit this section

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>[,]*]
```

- `<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`).
- **<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, if present, 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
  - `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
  - `default <any>` Defines default value if not present. Default is `null`.
  - `IF NOT EXISTS` If specified, create the property only if not exists. If a property with the same name already exists in the type, then no error is returned

When you create a property, ArcadeDB checks the data for property and type. In the event that persistent data contains incompatible values for the specified type, the property creation fails. It applies no other constraints on the persistent data.

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.

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 `date` 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'))
```

For more information, see:

- `ALTER PROPERTY`
- `DROP PROPERTY`

Supported Types

ArcadeDB supports the following data types for standard properties:

BOOLEAN	BYTE	SHORT	INTEGER	LONG
STRING	LINK	BINARY	DATE	DATETIME
FLOAT	DOUBLE	DECIMAL		

It supports the following data types for container properties:

LIST	MAP	EMBEDDED

SQL - CREATE TYPE

Edit this section

Creates a new type in the schema.

Syntax

```
CREATE <DOCUMENT|VERTEX|EDGE> TYPE <type>
[ IF NOT EXISTS ]
[EXTENDS <super-type>] [BUCKET <bucket-id>[,]*] [BUCKETS <total-bucket-number>]
```

- 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.
- **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.
- `<bucket-id>` Defines in a comma-separated list the ID's of the buckets you want this type to use.
- `<total-bucket-number>` Defines the total number of buckets you want to create for this type. The default value is `1`.

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.

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
```

Bucket Selection Strategies

When you create a type, it inherits the bucket selection strategy defined at the database-level. By default, this is set to `round-robin`. You can change the database default using the `ALTER DATABASE` command and the selection strategy for the type using the `ALTER TYPE` command.

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.

For more information, see:

- `ALTER TYPE`
- `DROP TYPE`
- `CREATE BUCKET`

SQL - `CREATE VERTEX`

Edit this section

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 vertexes 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> <code type="userinput lang-sql">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"}]
```

For more information, see:

- `CREATE EDGE`

SQL - DELETE

Edit this section

Removes one or more records from the database. You can refine the set of records that it removes using the `WHERE` clause.

Syntax:

```
DELETE FROM <Type>|BUCKET:<bucket>|INDEX:<index> [RETURN <returning>]
  [WHERE <Condition>*] [LIMIT <MaxRecords>] [TIMEOUT <MilliSeconds>] [UNSAFE]
```

- `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.
- `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.

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`
```

SQL - DROP BUCKET

Edit this section

Removes the bucket and all of its content. This operation is permanent and cannot be rolled back.

Syntax

```
DROP BUCKET <bucket-name>|<bucket-id>
```

- `<bucket-name>` Defines the name of the bucket you want to remove.
- `<bucket-id>` Defines the ID of the bucket you want to remove.

Examples

- Remove the bucket `Account`:

```
ArcadeDB> DROP BUCKET Account
```

For more information, see:

- `CREATE BUCKET`
- `DROP TYPE`

SQL - `DROP INDEX`

Edit this section

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.

Examples

- Remove the index on the `Id` property of the `Users` type:

```
ArcadeDB> DROP INDEX `Users[Id]`
```

For more information, see:

- `CREATE INDEX`
- Indexes

SQL - DROP PROPERTY

Edit this section

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.

Syntax

```
DROP PROPERTY <type>.<property> [FORCE]
```

- `<type>` Defines the type where the property exists.
- `<property>` Defines the property you want to remove.
- **FORCE** In case one or more indexes are defined on the property, the command will throw an exception. Use FORCE to drop indexes together with the property

Examples

- Remove the `name` property from the type `User`:

```
ArcadeDB> DROP PROPERTY User.name
```

For more information, see:

- `ALTER PROPERTY`
- `CREATE PROPERTY`

SQL - DROP TYPE

Edit this section

Removes a type from the schema.

Syntax

```
DROP TYPE <type> [UNSAFE] [IF EXISTS]
```

- `<type>` Defines the type you want to remove.
- `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.
- `IF EXISTS` Prevent errors if the type does not exits when attempting to drop it.

Bear in mind, that the schema must remain coherent. For instance, avoid

> removing types that are super-types to others. This operation won't delete the associated bucket.

Examples

- Remove the type `Account`:

```
ArcadeDB> DROP TYPE Account
```

For more information, see:

- `CREATE TYPE`
- `ALTER TYPE`

SQL - `EXPLAIN`

Edit this section

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

SQL - EXPORT DATABASE

Edit this section

Exports a database in the `exports` directory under the root directory where ArcadeDB is running.

Syntax

```
EXPORT DATABASE [<url>] [FORMAT JSONL|GRAPHML|GRAPHSON] [OVERWRITE TRUE|FALSE]
```

- `<url>` 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`.
- `<FORMAT>` The format of the export as a quoted string
  - **JSONL** exports in JSONL format - a newline-delimited JSON variant (this is the default format)
  - **GraphML** exports in the popular GraphML format. GraphML is supported by all the major Graph DBMS. This format does not support complex types, like collection of elements. Using GraphSON instead of GraphML is recommended
  - **GraphSON** database export. GraphSON is supported by all the major Graph DBMS
- `<OVERWRITE>` Overwrite the export file if exists. Default is false.

Examples

- Export the current database under the `exports/` directory:

```
ArcadeDB> EXPORT DATABASE file://database.jsonl.tgz
```

- Export the current database in GraphSON format, overwriting any existent file if present:

```
ArcadeDB> EXPORT DATABASE file://Movies.graphson.tgz FORMAT GraphSON OVERWRITE true
```

SQL - IMPORT DATABASE

Edit this section

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. This format does not support complex types, like collection of elements. Using GraphSON instead of GraphML is recommended
- **GraphSON** database export
- **JSON** 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.
  - `https://` and `http://` as prefix for remote files.

Examples

- 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"
```

See also Importer

SQL - INSERT

Edit this section

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>}[,]*]]
  [RETURN <expression>]
  [FROM <query>]
```

- `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.
- `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.

To insert only documents, vertices, or edges with a certain **unique** property, use a

unique index.

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.

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 a new record, adding a relationship.

In SQL-93 syntax:

```
ArcadeDB> INSERT INTO Employee (name, boss) VALUES ('jack', #11:09)
```

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`.

- 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.

SQL - MATCH

Edit this section

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`:

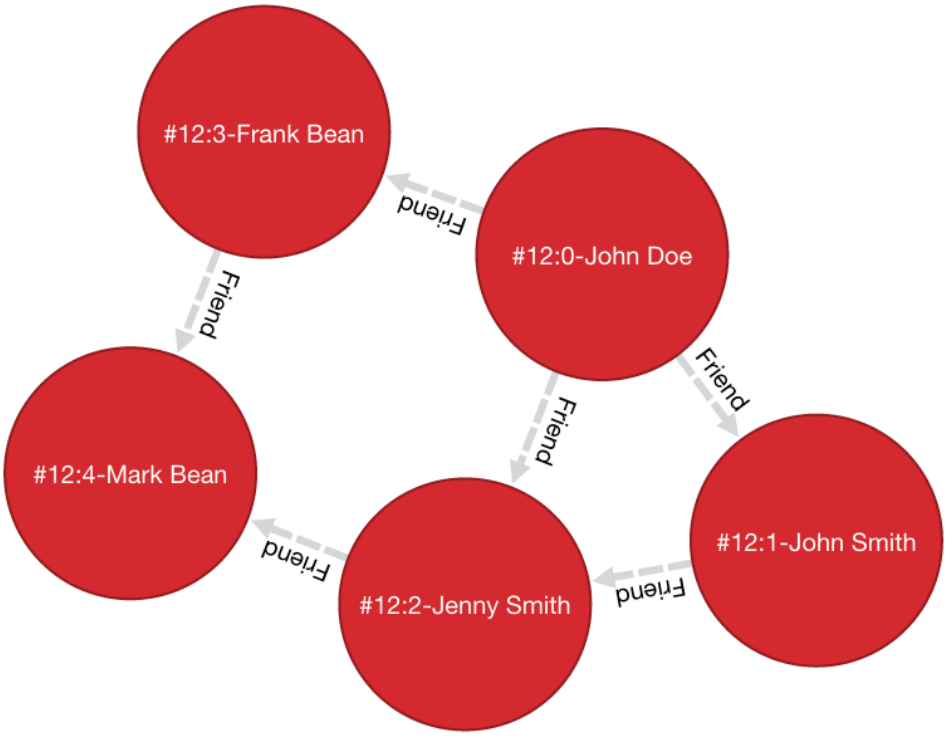
COMMAND

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
--------+----------------
```

- 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
```

Simplified Syntax

```
MATCH
  {
```

```
    [type: <type>],
    [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.
- `as: <alias>` Defines an alias for a node in the pattern.
- `<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.
- `<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-{…}`
- **`<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 nodes are allowed only on right terminal nodes, eg. `{} --> {optional:true}` is allowed, `{optional:true} <-- {}` is not.
- **`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.

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
--------
```

Context Variables

When running these queries, you can use any of the following context variables:

Variable	Description
`$matched`	Gives the current matched record. You must explicitly define the attributes for this record in order to access them. You can use this in the `where:` and `while:` conditions to refer to current partial matches or as part of the `RETURN` value.
`$currentMatch`	Gives the current complete node during the match.

Variable	Description
`$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 `e12s 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.

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')}.both("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.

SQL - MOVE VERTEX

Edit this section

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.

This is command is similar to the UPDATE command.

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
```

SQL - PROFILE

Edit this section

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.

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

SQL - `REBUILD INDEX`

Edit this section

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.

> During normal operations an index rebuild is not necessary. Rebuild an index only if it breaks.

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`
- `DROP INDEX`
- Indexes

SQL - SELECT

Edit this section

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 [ <Projections> ] [ FROM <Target> [ 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)
- `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.
  - 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:
    - `select from schema:types` to retrieve the defined types
    - `select from schema:indexes` to retrieve the defined indexes
    - `select from schema:database` to retrieve information about database settings
  - 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.
- `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 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. Note that ORDER BY works only on projection fields (fields that are returned to the result set) not on LET variables.
- `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.

Examples

- Return all records of the type `Person`, where the name starts with `Luk`:

```
ArcadeDB> SELECT FROM Person WHERE 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 * FROM Movie WHERE $depth <= 5 )
```

`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.

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' )
```

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'
--------+-------------
```

SQL SELECT Statements Execution

Edit this section

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
```

SQL - TRAVERSE

Edit this section

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 [<type.]field>|*
         [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, or a sub-query.
- `MAXDEPTH` Defines the maximum depth of the traversal. `0` indicates that you only want to traverse the root node. Negative values are invalid.
- `WHILE` Defines the condition for continuing the traversal while it is true.
- `LIMIT` Defines the maximum number of results the command can return.
- `STRATEGY` Defines strategy for traversing the graph.

> The use of the `WHERE` clause has been deprecated for this command.

> 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.

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 all fields in the root record:

```
ArcadeDB> TRAVERSE * 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
```

> You can also define the maximum depth in the `SELECT` command, but it's much more efficient to set it at the inner `TRAVERSE` statement because the returning record sets are already filtered by depth.

- 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.
- `<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`.
- `$depth` Gives the current depth of nesting.
- `$path` Gives a string representation of the current path. For instance, `#5:0.out`. You can also display it through `SELECT`:

```
ArcadeDB> SELECT $path FROM (TRAVERSE * FROM V)
```

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 on Twitter accounts, getting the second level of friendship:

```
ArcadeDB> SELECT FROM (TRAVERSE out('follow') FROM TwitterAccounts MAXDEPTH 2 )
          WHERE $depth = 2
```

But, you could also express this same query using `SELECT` operation, in a way that is also shorter and faster:

```
ArcadeDB> SELECT out('follow').out('follow') FROM TwitterAccounts
```

`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') MAXDEPTH 2 AND (@type = 'Message' or
          (@type = 'SentMessage' AND sentOn = '01/01/2012') )) WHERE
          @type = 'Message'
```

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.

SQL - TRUNCATE BUCKET

Edit this section

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.

Examples

- Remove all records in the bucket `profile`:

```
ArcadeDB> TRUNCATE BUCKET profile
```

For more information, see:

- `DELETE`
- `TRUNCATE TYPE`

SQL - TRUNCATE TYPE

Edit this section

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 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 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.

Examples

- Remove all records of the type `Profile`:

```
ArcadeDB> TRUNCATE TYPE Profile
```

For more information, see:

- `DELETE`
- `TRUNCATE BUCKET`
- `CREATE TYPE`

SQL - UPDATE

Edit this section

Update one or more records in the current database. Remember: ArcadeDB can work in schemaless 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]
  [RETURN <returning> [<returning-expression>]]
  [WHERE <conditions>]
  [LIMIT <max-records>] [TIMEOUT <MilliSeconds>]
```

- `SET` Defines the fields to update.
- `REMOVE` Removes an item in collection and map fields or a property.
- `CONTENT` Replaces the record content with a JSON document.
- `MERGE` Merges the record content with a JSON document.
- `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`.

- **`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.
- `TIMEOUT` Defines the time you want to allow the update run before it times out.

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']
```

- 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 elelment).

- 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 a record or insert if it doesn't already exist:

```
ArcadeDB> UPDATE Profile SET nick = 'Luca' UPSERT WHERE nick = 'Luca'
```

- 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")
```

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.

```
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.

8.3. Functions

Edit this section

SQL Functions are all the functions bundled with ArcadeDB SQL engine. Look also to SQL Methods.

> 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:

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.

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.

Function Reference

abs()

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()

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.

If property is not defined in edge or is null, distance between vertexes are 0.

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)
```

```
}
```

Examples

```
SELECT astar($current, #8:10, 'weight') FROM Vehicle
```

---

avg()

Returns the average value.

Syntax: `avg(<field>)`

Examples

```
SELECT avg(salary) FROM Account
```

---

bool_and()

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()

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()

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()

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()

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
```

---

circle()

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()

Returns the first field/value argument not being null parameter. If no field/value is not null, null is returns.

Syntax:

```
coalesce(<field|value> [, <field-n|value-n>]*)
```

Examples

```
SELECT coalesce(amount, amount2, amount3) FROM Account
```

---

concat()

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()

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(<field>)`

Examples

```
SELECT count(*) FROM Account
```

---

date()

Returns a date formatting a string. <date-as-string> is the date in string format, and <format> is the date format 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>] )`

Examples

```
SELECT FROM Account WHERE created <= date('2012-07-02', 'yyyy-MM-dd')
```

---

decode()

Decode 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()

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.

Examples

```
SELECT difference(tags) FROM book
```

```
SELECT difference(inEdges, outEdges) FROM OGraphVertex
```

---

dijkstra()

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.

Syntax: `dijkstra(<sourceVertex>, <destinationVertex>, <weightEdgeFieldName> [, <direction>])`

Examples

```
SELECT dijkstra($current, #8:10, 'weight') FROM Vehicle
```

---

distance()

Syntax: `distance( <x-field>, <y-field>, <x-value>, <y-value> )`

Returns the distance between two points in the globe using the Haversine algorithm. Coordinates must be in degrees.

Examples

```
SELECT FROM POI WHERE distance(x, y, 52.20472, 0.14056 ) <= 30
```

---

distinct()

Syntax: `distinct(<field>)`

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.

> The `distinct()` function has to be the sole projection component if used.

Examples

```
SELECT distinct(name) FROM City
```

---

duration()

Syntax: `duration(<field|integer>,'<string>')`

Returns a Java duration object, which can be useful to compare periods of time.

> The admissible second argument values are given here.

Examples

```
SELECT duration(start,'year') FROM Employees
```

---

encode()

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()

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.

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

on collections:

```
SELECT expand(addresses) FROM Account
```

on RIDs

```
SELECT expand(addresses) FROM Account
```

---

first()

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() [Function]

Formats a value using the String.format() conventions. Look here for more information.

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()

Syntax: `if(<expression>, <result-if-true>, <result-if-false>)`

Evaluates a condition (first parameters) and returns the second parameter if the condition is true, and the third parameter otherwise.

Examples

```
SELECT if( (name = 'John'), "My name is John", "My name is not John") FROM Person
```

---

ifnull() [Function]

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()

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
```

---

inE()

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()

Syntax: `intersect(<field> [,<field-n>]*)`

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.

Examples

```
SELECT intersect(friends) FROM profile WHERE jobTitle = 'programmer'
```

```
SELECT intersect(inEdges, outEdges) FROM OGraphVertex
```

---

inV()

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()

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()

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 OUser
```

---

lineString()

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()

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 OUser
```

---

max()

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.

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()

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()

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.

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()

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()

Get the adjacent outgoing vertices starting from the current record as vertex.

Syntax: `out([<label-1>][,<label-n>]*)`

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()

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()

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()

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.

Syntax: `percentile(<field> [, <quantile-n>]*)`

The quantiles have to be in the range 0—1

Examples

```
SELECT percentile(salary, 0.95) FROM Account
```

```
SELECT percentile(salary, 0.25, 0.75) AS IQR FROM Account
```

---

point()

Creates a 2D point from two numbers specifying X- and Y-coordinate.

Syntax: `point(<x>,<y>)`

Examples

```
SELECT point(10,20) AS point
```

---

polygon()

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
```

---

randomInt()

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>)`

Examples

```
SELECT randomInt(10) AS rand
```

You can use it in SQL Scripts to wait a random amount of milliseconds.

```
SLEEP randomInt(500);
```

---

rectangle()

Creates a 2D rectangle from four numbers specifying the left boundary X-, top boundary Y-, right boundary X- and botton boundary Y-values.

Syntax: `rectangle(<left-x>,<top-y>,<right-x>,<bottom-y>)`

Examples

```
SELECT rectangle(10,10,20,20) AS rectangle
```

---

set()

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 OUser
```

---

shortestPath()

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 parametes (Map<String, Object> in Java, JSON from SQL). Currently allowed parameters are - 'maxDepth': integer, maximum depth for paths (ignore path longer that 'maxDepth')

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})
```

---

sqrt()

Returns the absolute value. It works with Integer, Long, Short, Double, Float, BigInteger, BigDecimal, and null.

> Integer arguments are rounded down and negative arguments result in null.

Syntax: `sqrt(<field>)`

Examples

```
SELECT sqrt(score) FROM Account
```

```
SELECT sqrt(2.0)
```

```
SELECT sqrt(63)
```

---

stddev()

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()

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()

Syntax: `sum(<field>)`

Returns the sum of all the values returned.

Examples

```
SELECT sum(salary) FROM Account
```

---

symmetricDifference()

Syntax: `symmetricDifference(<field> [,<field-n>]*)`

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.

Examples

```
SELECT symmetricDifference(tags) FROM book
```

```
SELECT symmetricDifference(inEdges, outEdges) FROM OGraphVertex
```

---

sysdate()

Returns the current date time. If executed with no parameters, it returns a Date object, otherwise a string with the requested format/timezone. To know more about it, look at Managing Dates.

Syntax: `sysdate( [<format>] [,<timezone>] )`

Examples

```
SELECT sysdate('dd-MM-yyyy') FROM Account
```

---

traversedEdge()

Returns the traversed edge(s) in Traverse commands.

Syntax: `traversedEdge(<index> [,<items>])`

Where: - `<index>` is the starting edge to retrieve. Value ≥ 0 means absolute position in the traversed stack. 0 means the first record. Negative values are counted from the end: -1 means last one, -2 means the edge before last one, etc. - `<items>`, optional, by default is 1. If >1 a collection of edges is returned

Examples

Returns last traversed edge(s) of `TRAVERSE` command:

```
SELECT traversedEdge(-1) FROM ( TRAVERSE outE(), inV() FROM #34:3232 WHILE $depth <=
10 )
```

Returns last 3 traversed edge(s) of `TRAVERSE` command:

```
SELECT traversedEdge(-1, 3) FROM ( TRAVERSE outE(), inV() FROM #34:3232 WHILE $depth
<= 10 )
```

---

traversedElement()

Returns the traversed element(s) in Traverse commands.

Syntax: `traversedElement(<index> [,<items>])`

Where: - `<index>` is the starting item to retrieve. Value ≥ 0 means absolute position in the traversed stack. 0 means the first record. Negative values are counted from the end: -1 means last one, -2 means the record before last one, etc. - `<items>`, optional, by default is 1. If >1 a collection of items is returned

Examples

Returns last traversed item of TRAVERSE command:

```
SELECT traversedElement(-1) FROM ( TRAVERSE out() FROM #34:3232 WHILE $depth <= 10 )
```

Returns last 3 traversed items of TRAVERSE command:

```
SELECT traversedElement(-1, 3) FROM ( TRAVERSE out() FROM #34:3232 WHILE $depth <= 10
)
```

---

traversedVertex()

Returns the traversed vertex(es) in Traverse commands.

Syntax: `traversedVertex(<index> [,<items>])`

Where: - `<index>` is the starting vertex to retrieve. Value >= 0 means absolute position in the traversed stack. 0 means the first vertex. Negative values are counted from the end: -1 means last one, -2 means the vertex before last one, etc. - `<items>`, optional, by default is 1. If >1 a collection of vertices is returned

Examples

Returns last traversed vertex of TRAVERSE command:

```
SELECT traversedVertex(-1) FROM ( TRAVERSE out() FROM #34:3232 WHILE $depth <= 10 )
```

Returns last 3 traversed vertices of TRAVERSE command:

```
SELECT traversedVertex(-1, 3) FROM ( TRAVERSE out() FROM #34:3232 WHILE $depth <= 10 )
```

---

unionall()

Syntax: `unionall(<field> [,<field-n>]*)`

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.

Examples

```
SELECT unionall(friends) FROM profile
```

```
SELECT unionall(inEdges, outEdges) FROM OGraphVertex WHERE label = 'test'
```

---

uuid()

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()

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
```

---

vectorNeighbors()

Returns an array with the `num` most similar vectors from `index` (as string) to the `key`. The items in the returned array hold objects with their distance and keys.

> This function requires a vector index, see CREATE INDEX.

Syntax: `vectorNeighbors(<index>,<key>,<num>)`

Examples

```
SELECT vectorNeighbors('Word[name,vector]','Life',10)
```

---

version()

Returns the ArcadeDB version number and build as string.

Syntax: `version()`

Examples

```
SELECT version()
```

8.4. Methods

Edit this section

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.toJSON() 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.

> methods are case-insensitive.

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.

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.

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()`

Applies to the following types:

- short,
- int,
- long,
- float,
- double

---

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

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()`

Applies to the following types:

- any

Examples

```
SELECT salary.asDecimal() FROM Employee
```

---

asDouble()

Transforms the field into a double type.

Syntax: `<value>.asDouble()`

Applies to the following types:

- any

Examples

```
SELECT ray.asDouble() > 3.14
```

---

asFloat()

Transforms the field into a float type.

Syntax: `<value>.asFloat()`

Applies to the following types:

- any

Examples

```
SELECT ray.asFloat() > 3.14
```

---

asInteger()

Transforms the field into an integer type.

Syntax: `<value>.asInteger()`

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
```

---

asList()

Transforms the value in a List. 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()`

Applies to the following types:

- any

Examples

```
SELECT date.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
```

---

asShort()

Transforms the field into a short type.

Syntax: `<value>.asShort()`

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.

Syntax: `<value>.asString()`

Applies to the following types:

- any

Examples

Get all the salaries with decimals:

```
SELECT salary.asString().indexof('.') > -1
```

---

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>)`

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()

Excludes some properties in the resulting document.

Syntax: `<value>.exclude(<field-name>[,]*)`

Applies to the following types:

- document record

Examples

```
SELECT expand( @this.exclude( 'password' ) ) FROM OUser
```

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
```

---

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
```

---

format() [Method]

Returns the value formatted using the common "printf" syntax. For the complete reference goto Java Formatter JavaDoc.

Syntax: `<value>.format(<format>)`

Applies to the following types:

- any

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()

Include only some properties in the resulting document.

Syntax: `<value>.include(<value>[,]*)`

Applies to the following types:

- document record

Examples

```
SELECT expand( @this.include('name') ) FROM OUser
```

You can specify a wildcard as ending character to inclide all the fields that start with a certain string. Example to include all the fields that starts with `amonut`:

```
SELECT expand( @this.include('amount*') ) FROM V
```

---

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>])`

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.

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 FROM date.javaType() FROM Events
```

---

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 map.keys()
```

---

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>])`

Applies to the following types:

- string

```
SELECT 'Hello Elon'.lastIndexOf('l')
```

---

left()

Returns a substring of the original cutting from the begin and getting 'len' characters.

Syntax: `<value>.left(<length>)`

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`, 0 will be returned.

Syntax: `<value>.length()`

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
```

---

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.

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 out().in().remove( @this ) FROM V
```

---

removeAll()

Removes all the occurrences of the passed items.

Syntax: `<value>.removeAll(<item>*)`

Applies to the following types:

- collection

Examples

```
SELECT out().in().removeAll( @this ) FROM V
```

---

replace()

Replace a string with another one.

Syntax: `<value>.replace(<to-find>, <to-replace>)`

Applies to the following types:

- string

Examples

```
SELECT name.replace('Mr.', 'Ms.') FROM User
```

---

right()

Returns a substring of the original cutting from the end of the string 'length' characters.

Syntax: `<value>.right(<length>)`

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.

Syntax: `<value>.size()`

Applies to the following types:

- collection

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. 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:

- collection

Examples

Sort inline lists:

```
SELECT [4,1,9,23].sort()
SELECT ["a","Y","b","9"].sort(false)
```

---

split()

Returns a list from a string separated by the provided delimiter.

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(',')
```

---

subString()

Returns a substring of the original cutting from 'begin' index up to 'end' index (not included).

Syntax: `<value>.subString(<begin> [,<end>] )`

Applies to the following types:

- string

Examples

Get all the items where the name begins with an "L":

```
SELECT name.substring( 0, 1 ) = 'L' FROM StockItems
```

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
```

---

toJSON()

Returns the record in JSON format.

Syntax: `<value>.toJSON([<format>])`

Where:

- **format** optional, allows custom formatting rules (separate multiple options by comma).

Rules are the following:

- **rid** to include records's RIDs as attribute "@rid"
- **type** to include the type name in the attribute "@type"
- **attribSameRow** put all the attributes in the same row
- **indent** is the indent level as integer. By Default no ident is used
- **fetchPlan** is the fetching strategy to use while fetching linked records
- **alwaysFetchEmbedded** to always fetch embedded records (without considering the fetch plan)
- **dateAsLong** to return dates (Date and Datetime types) as long numbers
- **prettyPrint** indent the returning JSON in readeable (pretty) way.

Applies to the following types:

- record

Examples

```
CREATE VERTEX TYPE Test
INSERT INTO Test CONTENT {"attr1": "value 1", "attr2": "value 2"}

SELECT @this.toJson('rid,version,fetchPlan:in_*:-2 out_*:-2') FROM Test
```

---

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 date.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

Examples

```
SELECT FROM Clients WHERE map.values() CONTAINSALL (name IS NOT NULL)
```

8.5. SQL Script

Edit this section

ArcadeDB allows execution of arbitrary scripts written in Javascript or any scripting language installed in the JVM. ArcadeDB supports a minimal SQL engine to allow a batch of commands.

Batch of commands are very useful when you have to execute multiple things at the server side avoiding the network roundtrip for each command.

SQL Batch supports all the ArcadeDB SQL Commands, plus the following:

- `BEGIN [isolation <isolation-level>]`, where `<isolation-level>` can be `READ_COMMITTED`, `REPEATABLE_READ`. By default is `READ_COMMITTED`
- `COMMIT [retry <retry>]`, where:
  - `<retry>` is the number of retries in case of concurrent modification exception
- `ROLLBACK`
- `LET <variable> = <SQL>`, to assign the result of a SQL command to a variable. To reuse the variable prefix it with the dollar sign `$`.

> A `LET` can be used as an ephemeral view.

- `IF(<condition>){ <statememt>; [<statement>;]* }`. Look at conditional execution.
- `WHILE(<condition>){ <statememt>; [<statement>;]* }`. Look at loop.
- `FOREACH(<variable> IN <expression>){ <statememt>; [<statement>;]* }`. Look at loops.
- `SLEEP <ms>`, put the batch in wait for `<ms>` milliseconds.
- `RETURN <value>`, where `value` can be:
  - any value. Example: `return 3`
  - any variable with `$` as prefix. Example: `return $a`
  - arrays (HTTP protocol only, see below). Example: `return [ $a, $b ]`
  - maps (HTTP protocol only, see below). Example: `return { 'first' : $a, 'second' : $b }`
  - a query. Example: `return (SELECT FROM Foo)`

> to return arrays and maps (eg. Java or Node.js driver) it's strongly recommended using a RETURN SELECT, eg.

```
RETURN (SELECT $a as first, $b as second)
```

This will work on any protocol and driver.

Optimistic transaction

Example to create a new vertex in a Transactions and attach it to an existent vertex by creating a new edge between them. If a concurrent modification occurs, repeat the transaction up to 100 times:

```
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;
```

Note the usage of `$account` and `$city` in further SQL commands.

Conditional execution

SQL Batch provides IF constructor to allow conditional execution. The syntax is

```
IF( <sql-predicate> ){
  <statement>;
  <statement>;
  ...
}
```

There has to be a linebreak after `IF( <sql-predicate> ){` and `}` has to follow a linebreak.

`<sql-predicate>` is any valid SQL predicate (any condition that can be used in a `WHERE` clause):

```
IF( $a.size() > 0 ) {
  ROLLBACK;
}
```

Loops

SQL Batch provides two different loop blocks: `FOREACH` and `WHILE`.

FOREACH

Loops on all the items of a collection and, for each of them, executes a set of SQL statements. The expression argument should return an array, and can be a literal array, variable, or result of a sub-

query.

The syntax is:

```
FOREACH( <variable> IN <expression> ){
   <statement>;
   <statement>;
   ...
}
```

Example

```
FOREACH( $i IN [1, 2, 3] ){
  INSERT INTO Foo SET value = $i;
}
```

WHILE

Loops while a condition is true.

The syntax is:

```
WHILE( <condition> ){
   <statement>;
   <statement>;
   ...
}
```

Example

```
LET $i = 0;
WHILE ($i < 10){
  INSERT INTO Foo SET value = $i;
  LET $i = $i + 1;
}
```

8.6. Custom Functions

The SQL engine can be extended with custom functions written with a scripting language or via Java.

Database's function

Look at the Database Interface page.

Custom Functions in SQL

Custom functions can be defined via SQL using the following command:

```
DEFINE FUNCTION <library>.<name> "<body>" [PARAMETERS [<parameter>,*]] LANGUAGE
<language>
```

- `<library>` A name space to group functions.
- `<name>` The function's name.
- `<body>` The function's body encapsuled in a string in the chosen language's syntax.
- `[<parameter>,*]` A list of parameter identifiers used in the function body. For functions without parameters omit the `PARAMETERS []` block instead of supplying an empty array.
- `<language>` Either `js` (Javascript) or `sql`.

To invoke a custom function, the function identifier (`library.function`) must be enclosed with back ticks.

The return value in a custom SQL function is determined by the projection named `'result'`.

Examples

```
DEFINE FUNCTION my.fma 'return a + b * c' PARAMETERS [a,b,c] LANGUAGE js;
SELECT `my.fma`(1,2,3);
```

```
DEFINE FUNCTION the.answer 'SELECT "fourty-two" AS result' LANGUAGE sql;
SELECT `the.answer`();
```

A custom function can be deleted via SQL using the following command:

```
DELETE FUNCTION <library>.<name>
```

- `<library>` A name space to group functions.
- `<name>` The function's name.

Example

```
DELETE FUNCTION extra.tsum
```

Custom Functions in Java

Before to use them in your queries you need to register:

```
// REGISTER 'BIGGER' FUNCTION WITH FIXED 2 PARAMETERS (MIN/MAX=2)
SQLEngine.getInstance().registerFunction("bigger",
                                          new SQLFunctionAbstract("bigger", 2, 2) {
  public String getSyntax() {
    return "bigger(<first>, <second>)";
  }

  public Object execute(Object[] iParameters) {
    if (iParameters[0] == null || iParameters[1] == null)
      // CHECK BOTH EXPECTED PARAMETERS
      return null;

    if (!(iParameters[0] instanceof Number) || !(iParameters[1] instanceof Number))
      // EXCLUDE IT FROM THE RESULT SET
      return null;

    // USE DOUBLE TO AVOID LOSS OF PRECISION
    final double v1 = ((Number) iParameters[0]).doubleValue();
    final double v2 = ((Number) iParameters[1]).doubleValue();

    return Math.max(v1, v2);
  }

  public boolean aggregateResults() {
    return false;
  }
});
```

Now you can execute it:

```
Resultset result = database.command("sql", "SELECT FROM Account WHERE bigger( salary,
10 ) > 10");
```

Chapter 9. Security

ArcadeDB manages the security at server level only. This means if you work in embedded mode, there is no security available by default unless you install the server security or your own implementation. Without any kind of security active, any user can read and write in the database. For this reason it's important your application is managing security and profiling. You can work in embedded mode and still run a `ArcadeDBServer` instance to use the security for the incoming connections.

9.1. 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!");
  }
});
```

9.2. 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:

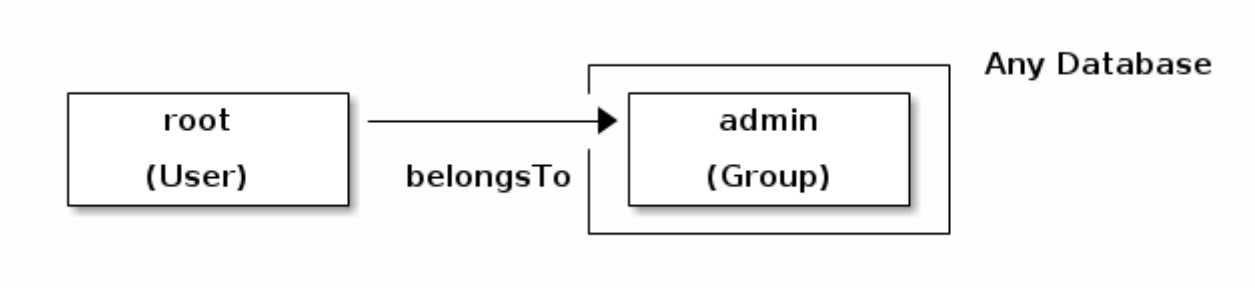
```
+-------------------------------------------------------------------+
|                WARNING: FIRST RUN CONFIGURATION                   |
+-------------------------------------------------------------------+
| This is the first time the server is running. Please type a       |
| password of your 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:

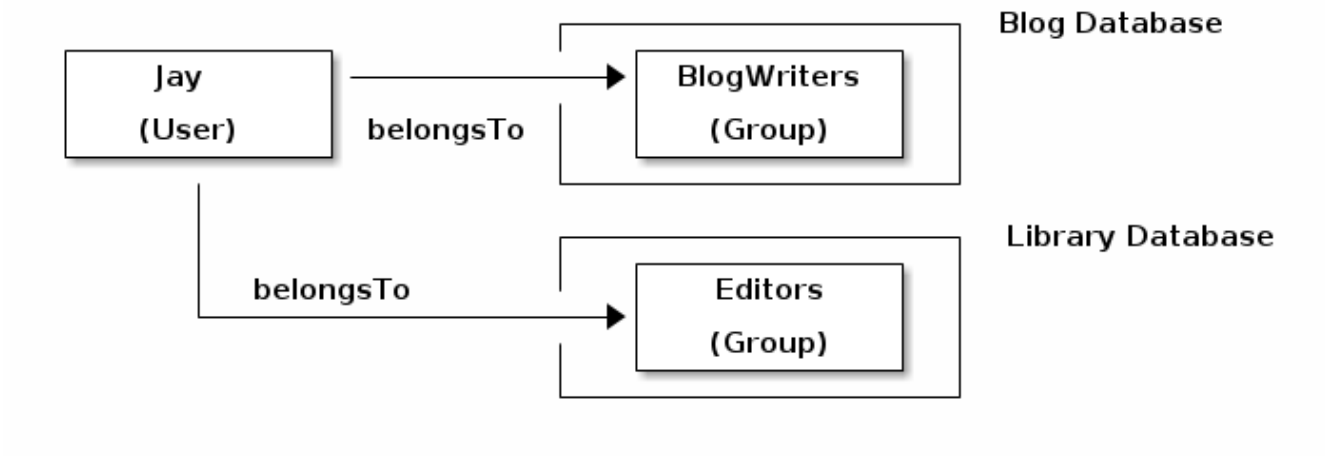
- Name, mandatory
- Password. It is always saved hashed by using the algorithm `PBKDF2` with a configurable salt (default = 32). The password is mandatory for all the users, but root. In the case root has no password, then ArcadeDB server asks to insert a password at startup (see above).
- Databases, 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.



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.

Via Console

Alternatively, users can be created via the console, see `Create User` Console Command and Server Command.

9.3. 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.)
  - `updateSchema`: to update the database schema (create, modify and drop buckets, types and indexes)

- `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.

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.

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"
  ],
  "resultSetLimit": -1,
  "readTimeout": -1,
  "types": {
    "*": {
      "access": [
        "createRecord",
        "readRecord",
        "updateRecord",
        "deleteRecord"
      ]
    }
  }
}
```

Which allows to execute any operation against the security, the schema and records.

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.

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.

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.

Chapter 10. Comparison with other DBMSs

This chapter contains the comparison between ArcadeDB and other DBMS. If you're familiar with one of those, understanding ArcadeDB takes a few minutes.

10.1. OrientDB

Edit this section

ArcadeDB was born initially as a fork of OrientDB. Today more than 80% of ArcadeDB code has been rewritten from scratch from the same original authors of the OrientDB project. This allowed to get rid of many legacy parts that makes OrientDB slow, heavy and hard to maintain. Also, since OrientDB was the first Multi-Model project out there, a lot of work of the initial R&D and experiments are still in the OrientDB code base. You can consider ArcadeDB as the natural evolution of the legacy OrientDB project.

If you're coming from OrientDB, please use the OrientDB Importer tool to import an OrientDB export into an ArcadeDB database.

Main similarities and differences

- Both can run on any platform
- Both can run SQL; ArcadeDB and OrientDB both share the same SQL Engine.
- ArcadeDB's SQL dialect is closely related to OrientDB's OSQL, but more streamlined.
- ArcadeDB "types" are the "classes" in OrientDB
- ArcadeDB "buckets" are similar to the "clusters" in OrientDB, but without the limitation of having only 32,768 clusters. The maximum number of buckets in ArcadeDB are 2,147,483,648
- Both ArcadeDB and OrientDB support multiple inheritance
- ArcadeDB shares the same database instance across threads, this makes it much easier developing with ArcadeDB than with OrientDB with multi-threads applications. With OrientDB you have to use a pool of databases and be careful on acquiring and releasing instanced. With ArcadeDB create a Database instance at the beginning, share it with all your threads and close when your application shuts down.
- ArcadeDB uses thread locals only to manage transactions, while OrientDB makes a strong usage of thread local structures internally, making hard to pass the database instance across threads and a pool if needed
- There is no base `V` and `E` classes in ArcadeDB, but rather vertex and edge are first type citizens types of records. Use `CREATE VERTEX TYPE Product` vs OrientDB `CREATE CLASS Product EXTENDS V`. Same for edges, use `CREATE EDGE TYPE Sold` vs OrientDB `CREATE CLASS Sold EXTENDS E`.
- ArcadeDB saves every type and property name in the dictionary to compress record size by storing only the names ids (as varint)
- ArcadeDB keeps the MVCC counter on the page rather than on the record. This means the transaction must be repeated if there are consurrent modification on the same page, not only

on the same record (like with OrientDB)

- ArcadeDB manages everything as files and pages, for transactions and replication. OrientDB has a mixed pages/record approach. Using the page-only approach keeps everything much faster and easier to maintain
- ArcadeDB allows custom page size per bucket/index
- ArcadeDB supports light-weight edges (edges without properties), but they must be used with a different syntax. This avoids automatic upgrade of edges and unexpected behavior experienced in OrientDB
- ArcadeDB supports replication by using a Leader/Replica model with Raft election without sharding for now. Instead, OrientDB is based on a Multi-Master model (the sharding was experimental, never production ready) with a multi-paxos style protocol not efficient on large volume of transactions and still not rock-solid after years because of its complexity
- ArcadeDB replicates the pages across servers, so all the databases are identical at binary level, while with OrientDB there is a mix of logical and physical replication leaving room for not managed edge cases
- ArcadeDB Server supports HTTP/JSON, Postgres, MongoDB and Redis protocols, while OrientDB supports only HTTP/JSON and a proprietary binary protocol
- ArcadeDB supports OrientDB SQL, the latest Gremlin version, Open Cypher and MongoDB query language, while OrientDB supports only its SQL and an old version of Gremlin

What ArcadeDB does not support

- ArcadeDB supports only UNIQUE constraints on data (by creating an index), while OrientDB supports multiple constraints and validation at class level
- ArcadeDB does not provide a dirty manager, so it's up to the developer to mark the object to save by calling `.save()` method on it. This makes the code of ArcadeDB smaller without handling edge cases, but if you have a tree of objects it is the developer responsibility to mark the modified objects without auto-tracking
- ArcadeDB does not allow a document to have no class. If you want to store an embedded document without a class, use a `Map<String,Object>` instead

What ArcadeDB has more than OrientDB

- ArcadeDB is much Faster than OrientDB. On a single server it is common to see 10X-20X improvement in performance, with 3 nodes the gap in performance with OrientDB can reach 50X-200X faster. With 10 servers it is over 500X faster than OrientDB!
- The maximum number of buckets are `2,147,483,648`, while with OrientDB the maximum number of clusters is `32,768`
- ArcadeDB uses much less RAM. With the right tuning over the settings, it's able to work with only 4MB of JVM heap, while OrientDB requires at least 8GB to run
- ArcadeDB codebase is much smaller and easier to maintain and improve
- ArcadeDB is lightweight, the engine is about 1MB
- ArcadeDB mandate all the operations to be inside a transaction, even operations against the

schema. With OrientDB, they are no transactional and in case of error they can break the database

- ArcadeDB saves every type and property names in the dictionary to compress the record by storing only the names ids
- ArcadeDB is much more efficient on data structure. That means ArcadeDB takes less space on disk than OrientDB and uses less RAM for caching
- ArcadeDB natively supports asynchronous operations (by using `.async()`). Asynchronous calls are automatically balanced on the available cores with a nice API

Chapter 11. Appendix

11.1. Data Types

Edit this section

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
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
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

Type	SQL type	Description	Java type	Minimum — Maximum	Auto-conversion from/to
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
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

Type	SQL type	Description	Java type	Minimum — Maximum	Auto-conversion from/to
Decimal	DECIMAL	Decimal numbers without rounding	`java.math.BigDecimal`		Any Number, String

11.2. Settings

Edit this section

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.

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 `JAVA_OPTS`:

```
JAVA_OPTS="-Darcadedb.server.rootPassword=playwithdata" java ...
```

SQL (Database Level)

All the changes executed via SQL Alter Database command are relative to the current database only and are persistent. Example to increase the default page size for buckets to 1 MB:

```
ALTER DATABASE `arcadedb.bucketDefaultPageSize` 1048576
```

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);
```

11.2.1. 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
`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
`profile`	Specify the preferred profile among: default, high-performance, low-ram, low-cpu	String	default

Name	Description	Type	Default Value
`test`	Tells if it is running in test mode. This enables the calling of callbacks for testing purpose	Boolean	false

SERVER

Name	Description	Type	Default Value
`ha.clusterName`	Cluster name. By default is 'arcadedb'. Useful in case of multiple clusters in the same network	String	arcadedb
`ha.enabled`	True if HA is enabled for the current server	Boolean	false
`ha.k8s`	The server is running inside Kubernetes	Boolean	false
`ha.k8sSuffix`	When running inside Kubernetes use this suffix to reach the other servers. Example: arcadedb.default.svc.cluster.local	String	
`ha.quorum`	Default quorum between 'none', 1, 2, 3, 'majority' and 'all' servers. Default is majority	String	majority
`ha.quorumTimeout`	Timeout waiting for the quorum	Long	10000
`ha.replicationChunkMaxSize`	Maximum channel chunk size for replicating messages between servers. Default is 16777216	Integer	16777216
`ha.replicationFileMaxSize`	Maximum file size for replicating messages between servers. Default is 1GB	Long	1073741824
`ha.replicationIncomingHost`	TCP/IP host name used for incoming replication connections. By default is 0.0.0.0 (listens to all the configured network interfaces)	String	0.0.0.0
`ha.replicationIncomingPorts`	TCP/IP port number used for incoming replication connections	String	2424-2433
`ha.replicationQueueSize`	Queue size for replicating messages between servers	Integer	512
`ha.serverList`	Servers in the cluster as a list of <hostname/ip-address:port> items separated by comma. Example: 192.168.0.1:2424,192.168.0.2:2424	String	
`ha.serverRole`	Enforces a role in a cluster, either "any" or "replica"	String	"any"
`network.socketTimeout`	TCP/IP Socket timeout (in ms)	Integer	30000
`ssl.keyStore`	Path where the SSL certificates are stored	String	null
`ssl.keyStorePass`	Password to open the SSL key store	String	null
`ssl.trustStore`	Path to the SSL trust store	String	null
`ssl.trustStorePass`	Password to open the SSL trust store	String	null

Name	Description	Type	Default Value
`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
`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	Boolean	true
`server.databaseDirectory`	Directory containing the database	String	${arcadedb.server.rootPath}/databases
`server.backupDirectory`	Directory containing the backups	String	${arcadedb.server.rootPath}/backups
`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`. Example: `Universe[elon:musk:admin];Amiga[Jay:Miner,Jack:Tramiel]{import:/tmp/movies.tgz}`	String	
`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

Name	Description	Type	Default Value
`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.httpsIoThreads`	Number of threads to use in the HTTP server	Integer	2 per core
`server.httpTxExpireTimeout`	Timeout in seconds for a HTTP transaction to expire. This timeout is computed from the latest command against the transaction	Long	30
`serverMetrics`	True to enable metrics	Boolean	true
`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	
`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.securitySaltCacheSize`	Cache size of hashed salt passwords. The cache works as LRU. Use 0 to disable the cache	Integer	64
`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

DATABASE

Name	Description	Type	Default Value
`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
`asyncTxBatchSize`	Maximum number of operations to commit in batch by async thread	Integer	10240
`asyncWorkerThreads`	Number of asynchronous worker threads. 0 (default) = available cores minus 1	Integer	15
`bucketDefaultPageSize`	Default page size in bytes for buckets. Default is 65536	Integer	65536
`command.timeout`	Default timeout for commands (in ms)	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	Long	5000
`cypher.statementCache`	Max number of entries in the cypher statement cache. Use 0 to disable. Caching statements speeds up execution of the same cypher queries	Integer	1000
`dateFormat`	Default date format using Java SimpleDateFormat syntax	String	yyyy-MM-dd
`dateImplementation`	Default date implementation to use on deserialization. By default java.util.Date is used, but the following are supported: java.util.Calendar, java.time.LocalDate	Class	class java.util.Date
`dateTimeFormat`	Default date time format using Java SimpleDateFormat syntax	String	yyyy-MM-dd HH:mm:ss
`dateTimeImplementation`	Default datetime implementation to use on deserialization. By default java.util.Date is used, but the following are supported: java.util.Calendar, java.time.LocalDateTime, java.time.ZonedDateTime	Class	class java.util.Date
`flushOnlyAtClose`	Never flushes pages on disk until the database closing	Boolean	false
`freePageRAM`	Percentage (0-100) of memory to free when Page RAM is full	Integer	50
`gremlin.timeout`	Default timeout for gremlin commands (in ms)	Long	8000

Name	Description	Type	Default Value
`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
`maxPageRAM`	Maximum amount of pages (in MB) to keep in RAM	Long	4096
`pageFlushQueue`	Size of the asynchronous page flush queue	Integer	512
`polyglotCommand.timeout`	Default timeout for polyglot commands (in ms)	Long	10000
`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.This setting is intended as a safety measure against excessive resource consumption from a single query (eg. prevent OutOfMemory)	Long	500000
`sqlStatementCache`	Maximum number of parsed statements to keep in cache	Integer	300
`txRetries`	Number of retries in case of MVCC exception	Integer	3
`txWAL`	Uses the WAL	Boolean	true
`txWalFlush`	Flushes the WAL on disk at commit time. It can be 0 = no flush, 1 = flush without metadata and 2 = full flush (fsync)	Integer	0
`typeDefaultBuckets`	Default number of buckets to create per type	Integer	8

Available Plugins

Name	`server.plugins`-String
Gremlin	`GremlinServer:com.arcadedb.server.gremlin.GremlinServerPlugin`
MongoDB	`MongoDB:com.arcadedb.mongo.MongoDBProtocolPlugin`
Postgres	`Postgres:com.arcadedb.postgres.PostgresProtocolPlugin`
Redis	`Redis:com.arcadedb.redis.RedisProtocolPlugin`

11.3. 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).

Starting from v21.1.1, ArcadeDB sully support 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.

If you're working with the Java API, the class used for both `DATE` and `DATETIME` is `java.util.Date`. With `DATETIME` values, if you're using higher precision than millisecond you need to use the new Java Time API:

- `java.time.LocalDateTime`
- `java.time.ZonedDateTime`
- `java.time.Instant`

If you're starting a new project with ArcadeDB, using the new Java Time API is strongly advised. For example, if you want to use `LocalDateTime` as default class that represent `DATETIME`, set it in the binary serializer at startup and ArcadeDB will always return a LocalDateTime for DATETIME types:

```
alter database `arcadedb.dateTimeImplementation` `java.time.LocalDateTime`
```

You can do the same for `DATE` types using the new Java Time API for dates `java.time.LocalDate`:

```
alter database `arcadedb.dateImplementation` `java.time.LocalDate`
```

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`

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.

```
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.

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'`

11.4. Binary Types (BLOB)

Edit this section

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");
});
```

11.5. SQL Syntax

Edit this section

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
- `@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

- 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
- 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:

```
(single precision)
1.5
12345678.65432
-45.0

(double precision)
0.23D
.23D
```

- **absolute precision, decimal numbers**: like BigDecimal in Java

Use the `bigDecimal(<number>)` function 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.

- **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 (withot 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 result is `NULL`;

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)

Float numbers are represented in SQL as `[-][<number>].<number>`, eg. valid Float values are `1.5`, `-1567.0`, `.556767`. If the number represented exceeds the Float maximum size (see Java java.lang.Float `MAX_VALUE` and `MIN_VALUE`), then it's automatically converted to a Double.

Double numbers are represented in SQL as `[-][<number>].<number>D` (D can be uppercase or lowercase), eg. valid Float values are `1.5d`, `-1567.0D`, `.556767D`. If the number represented exceeds the Double maximum size (see Java java.lang.Double `MAX_VALUE` and `MIN_VALUE`), then the result is `NULL`

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 `bigDecimal(<number> | <string>)` funciton, eg. `bigDecimal(124.4)` or `bigDecimal("124.4")`

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
```

the overflow follows Java rules.

The conversion of a number to BigDecimal can be done explicitly, using the `bigDecimal()` funciton, eg. `bigDecimal(124.4)` or `bigDecimal("124.4")`

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
- `<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 expresson (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.

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'`)
- `==` **(equals)**: same as `=`
- `!=` **(not equals)**: inequality operator.
- `<>` **(not equals)**: same as `!=`
- `>` **(greater than)**
- `>=` **(greater or equal)**
- `<` **(less than)**
- `<=` **(less or equal)**
- `<=>` **(null-safe equals)**

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'`. It can also be used as a unary operator (no effect).

- `-` **(minus)**: subtraction between numbers. Non-number operands are evaluated to zero. Null values are treated as a zero, eg `1 + null = 1`. Minus can also be used as a unary operator, to invert the sign of a number.
- `*` **(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. The result of a division by zero is NaN.
- `%` **(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.
- `^` **(bitwise XOR)** executes a bitwise XOR operation. Both operands have to be Integer or Long values, otherwise the result will be null.
- `||`: 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`.

- `+=` **(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`: logical NOT
- `CONTAINS`: checks if the left collection contains the right element. The left argument has to be a colleciton, otherwise it returns FALSE. It's NOT the check of colleciton intersections, so `['a', 'b', 'c'] CONTAINS ['a', 'b']` will return FALSE, while `['a', 'b', 'c'] CONTAINS 'a'` will return TRUE.
- `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
- `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 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'`

11.6. Graph Database

ArcadeDB is a native graph database; and in this section we explain what this means and how this relates to applications.

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:

```
v_1 --e_(1,2)--> v_2
```

with:

- Vertices: V = {v_1, v_2}
- Edges: E = {e_(1,2)}

Graph Database Types

There are two common types of graph databases, which are instances of directed graphs, but differ by the underlying objects of vertices and edges:

1. **Knowledge Graph**: the underlying vertex and edge objects are (just) labels.
2. **Property Graph**: the underlying vertex and edge objects are (labeled) documents.

ArcadeDB is a property graph, which means a vertex or edge consists of an identifier (RID), a label (type), and properties (document), in addition to the ordered pairs of endpoints. The latter are here vertex properties of incoming and outgoing edges, instead of edge properties of ordered vertex pairs, for technical reasons.

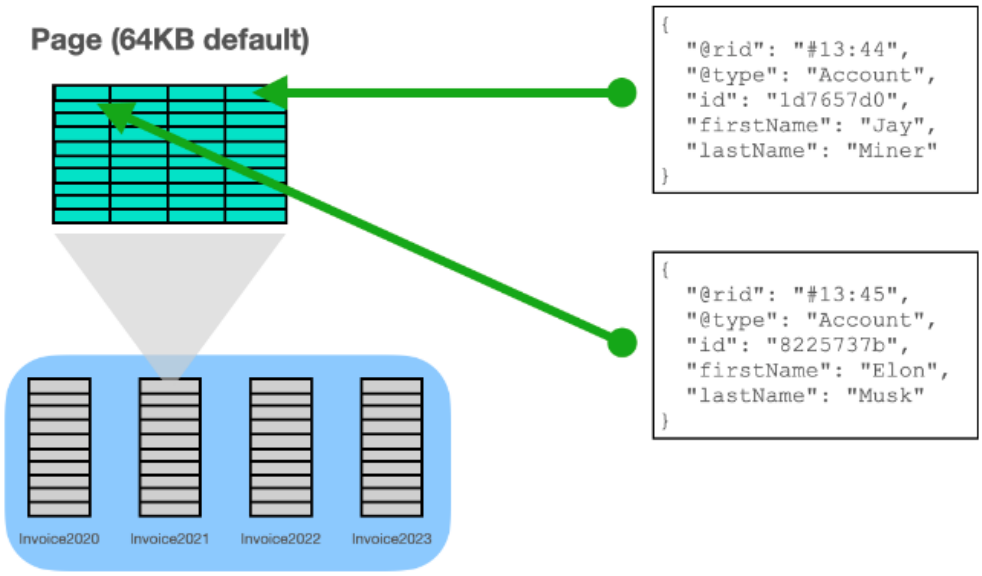
Why (Property) Graph Databases?

- 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 knowledge graphs.

11.7. Storage Internals

Edit this section

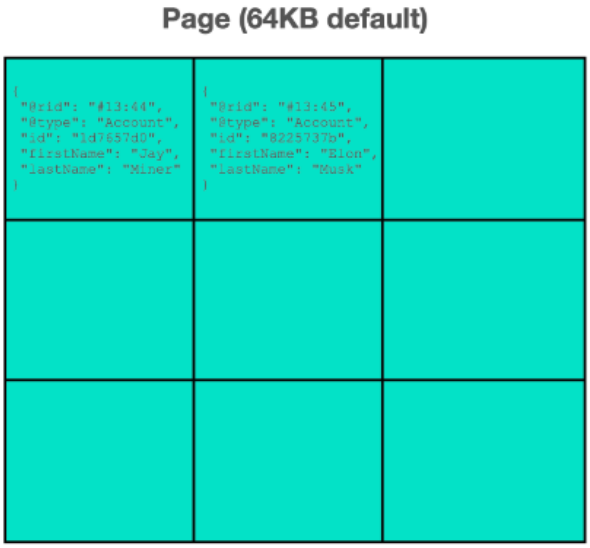
11.7.1. Pages and Record IDs



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.

11.7.2. Zero Copy Objects



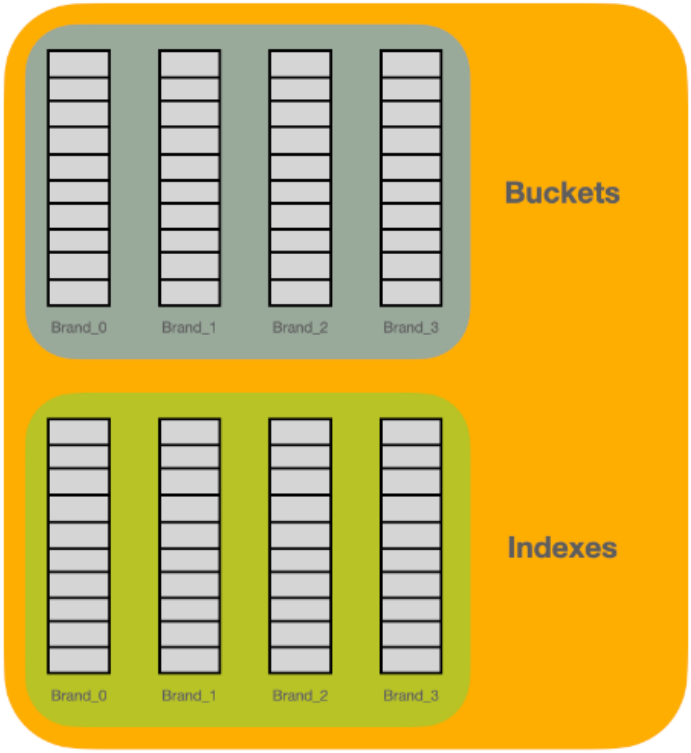
Pages are loaded in RAM as `ByteBuffer` objects (backed by byte arrays - `byte[]`). When a record is read, an in **immutable** vertex object is created which points to the portion of the `ByteBuffer` relative at the record.

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[]`.

11.7.3. How Buckets Work

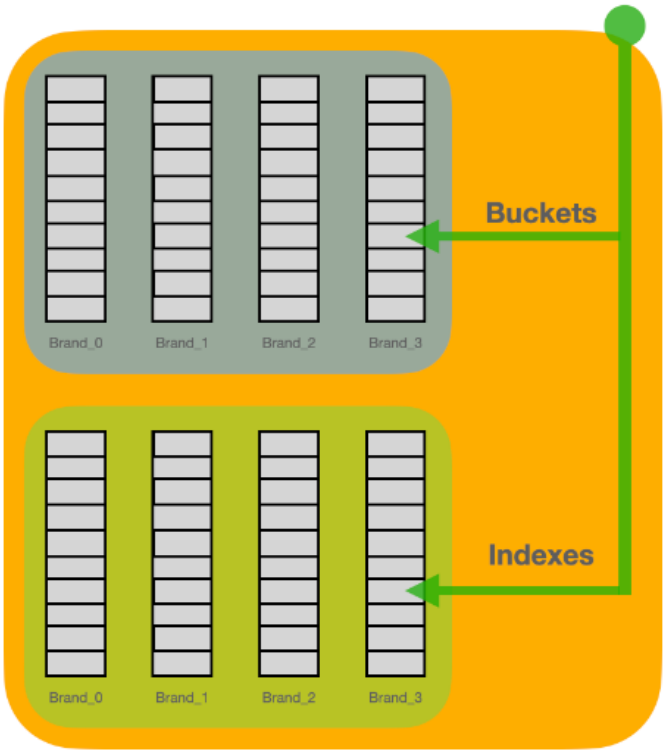
```
sql> create vertex type Brand
```



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.

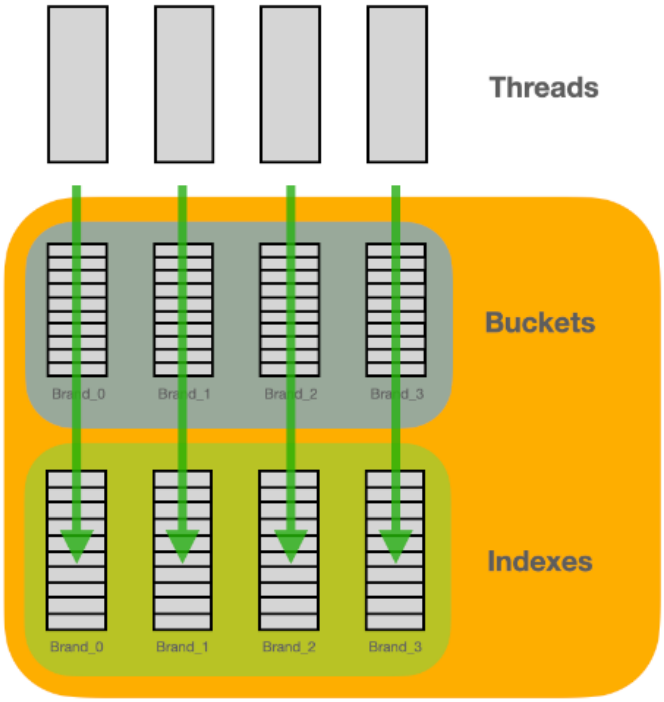
```
gremlin> g.addV('Brand').property('name', 'Maserati')
```



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')
```

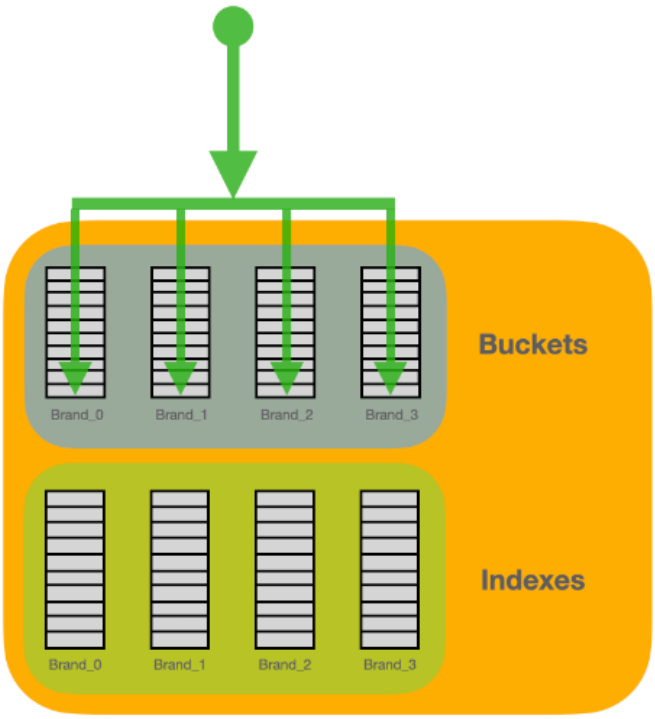


Thread strategy is preferable with multi-thread insertion where each thread insert on the same bucket and index files with zero contention with other threads.

```
bucket = threadId % available buckets
```

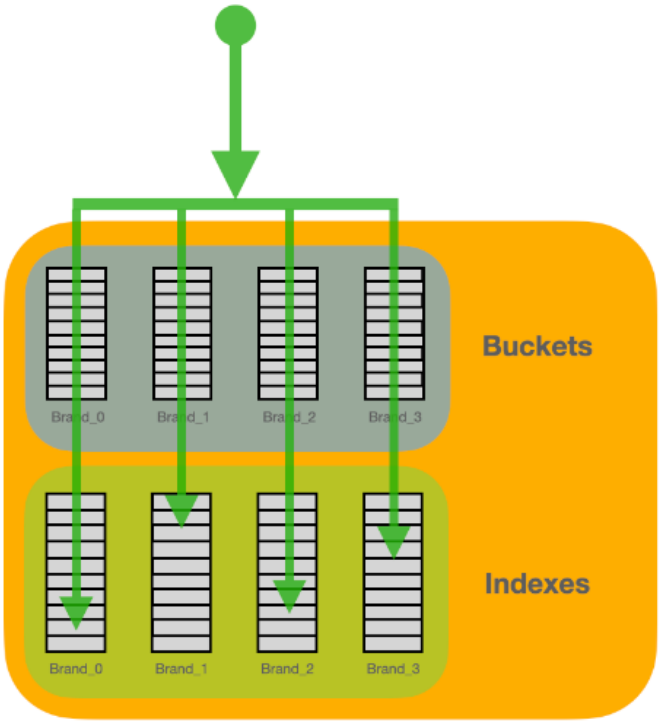
On a 2019 MacBook Pro ArcadeDB can insert **3.5 Million vertices/s** without index and **1.6 Million vertices/s** with one index

```
gremlin> g.V().hasLabel('Brand')
```



When no indexes are involved, the entire type is scanned. ArcadeDB browses all the buckets associated with the type.

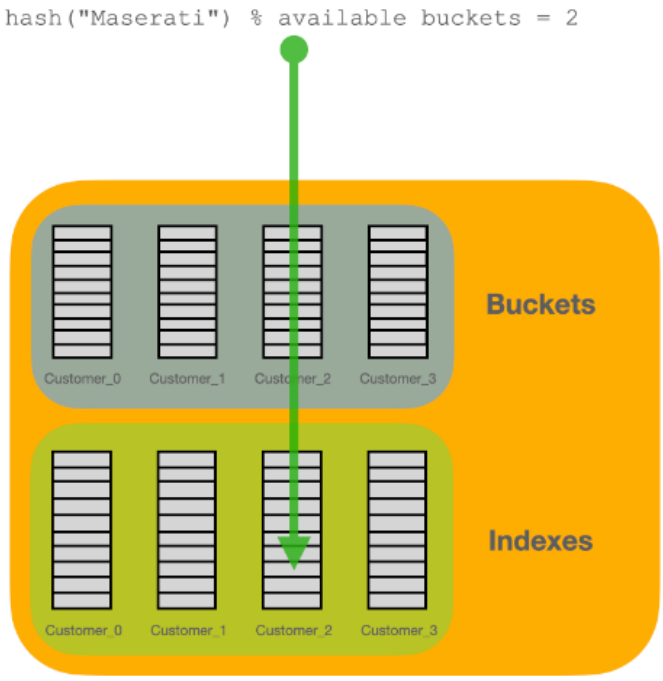
```
gremlin> g.V().hasLabel('Brand').has('name', 'Maserati')
```



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.

11.7.4. 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:

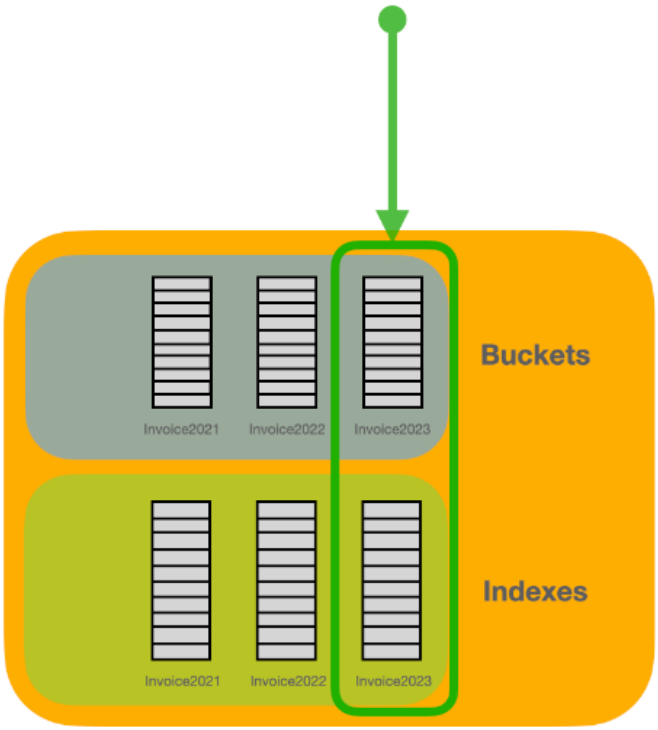
   ```
   create index on Brand (name) unique
   ```

2. Use the bucket selection strategy "partitioned(name)":

   ```
   alter type Brand BucketSelectionStrategy
   `partitioned(name)`
   ```

11.7.5. Add and Remove Buckets from Types

```
sql> ALTER TYPE Invoice BUCKET +Invoice2023
```



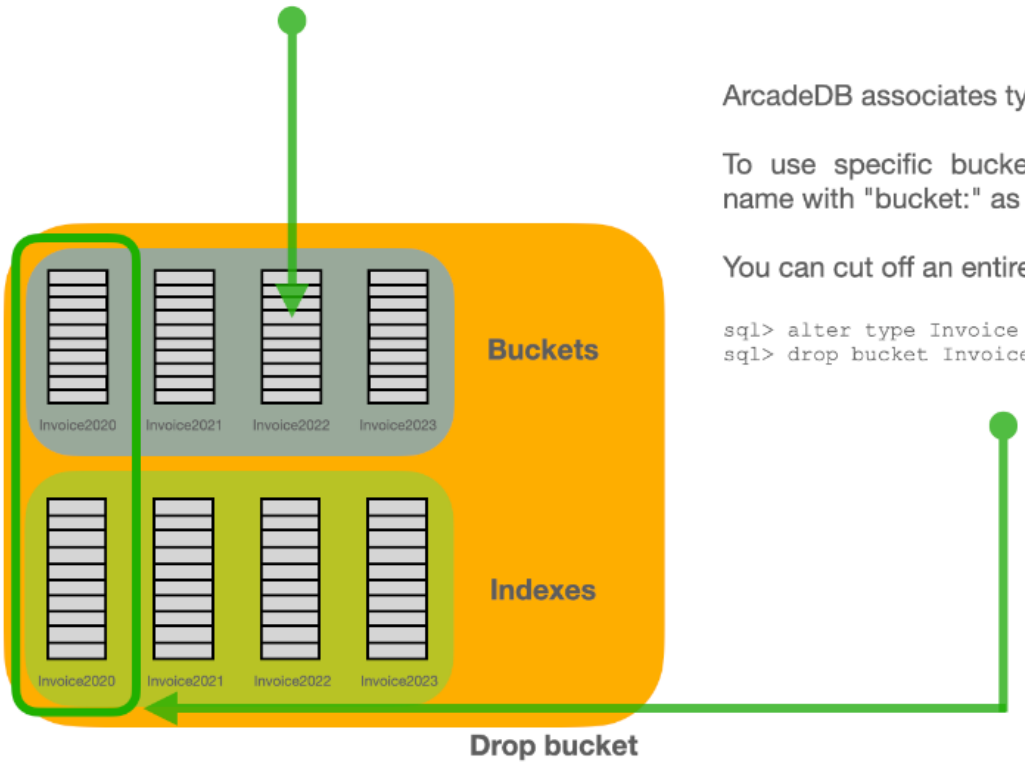
When a new bucket is added to a type, ArcadeDB creates automatically the underlying index.

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.

```
gremlin> g.V().addV('bucket:Invoice2022').property('id', '123')
gremlin> g.V().hasLabel('bucket:Invoice2022').has('id', '123')
```



ArcadeDB associates types to Gremlin labels.

To use specific buckets, prefix the bucket name with "bucket:" as label.

You can cut off an entire year of invoices with:

```
sql> alter type Invoice bucket -Invoice2020
sql> drop bucket Invoice2020
```

11.7.6. 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.

11.8. LSM-Tree Algorithm

Edit this section

ArcadeDB indexes are built by using the LSM Tree algorithm.

11.8.1. 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.

11.9. Requirements

Edit this section

Memory

ArcadeDB's memory (RAM) requirements depend on the configured profile:

- `default` profile: 800MB
- `low-ram` profile: 16MB

see also RAM Configuration.

Java

ArcadeDB runs on the Java Runtime Environment (JRE) in versions **11** to **17**; tested JREs are:

- OpenJDK 11 is compatible
- OpenJDK 17 is compatible and the suggested version to use
- OpenJDK 21 is not compatible

> That **headless** variants of JRE packages are compatible to ArcadeDB.

11.10. 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
- 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/

Frequently Asked Questions

- How do I build or contribute to ArcadeDB?
  - See the contributing guide in the ArcadeDB github repository: CONTRIBUTING.md
- 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.

11.11. Report an Issue

Edit this section

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.

```
* Java and JVM are registered trademarks of Oracle Corporation
* * All the trademarks are property of their owner. ArcadeDB does not own such
trademarks.
```

```
ArcadeDB is a trademark registered by Arcade Data Ltd. Copyright (c) Arcade Data LTD.

Every effort has been made to ensure the accuracy of this manual. However, Arcade
```

```
Data, LTD. makes no warranties with respect to this documentation and disclaims any
implied warranties of merchantability and fitness for a particular purpose. The
information in this document is subject to change without notice.

If you would like to report an issue in the documentation or you would like to be part
of our community on improving the documentation for ArcadeDB Open Source project,
please send your changes through our GitHub project and send a Pull Request for
approval.
```