<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6. Downgrade to older version of ArcadeDB</td>
<td>49</td>
</tr>
<tr>
<td>7. API</td>
<td>50</td>
</tr>
<tr>
<td>7.1. Java API (Embedded)</td>
<td>50</td>
</tr>
<tr>
<td>7.1.1. 10-Minute Tutorial</td>
<td>51</td>
</tr>
<tr>
<td>7.1.2. Schema</td>
<td>59</td>
</tr>
<tr>
<td>7.1.3. Working with buckets</td>
<td>60</td>
</tr>
<tr>
<td>7.1.4. Working with indexes</td>
<td>60</td>
</tr>
<tr>
<td>7.1.5. Database Configuration</td>
<td>61</td>
</tr>
<tr>
<td>7.1.6. Embedded Documents</td>
<td>62</td>
</tr>
<tr>
<td>7.1.7. Events</td>
<td>64</td>
</tr>
<tr>
<td>7.1.8. DatabaseFactory Class</td>
<td>65</td>
</tr>
<tr>
<td>7.1.9. Database Interface</td>
<td>67</td>
</tr>
<tr>
<td>7.2. HTTP/JSON Protocol</td>
<td>87</td>
</tr>
<tr>
<td>7.2.1. Reference</td>
<td>90</td>
</tr>
<tr>
<td>7.3. Postgres Protocol Plugin</td>
<td>96</td>
</tr>
<tr>
<td>7.4. Execute multiple query languages</td>
<td>97</td>
</tr>
<tr>
<td>7.5. Current limitations</td>
<td>97</td>
</tr>
<tr>
<td>7.5.1. Transactions</td>
<td>97</td>
</tr>
<tr>
<td>7.6. Postgres Tools Known to Work</td>
<td>98</td>
</tr>
<tr>
<td>7.6.1. JetBrains DataGrip/Database Plugin</td>
<td>98</td>
</tr>
<tr>
<td>7.6.2. psql</td>
<td>99</td>
</tr>
<tr>
<td>7.7. Connect with JDBC Driver</td>
<td>99</td>
</tr>
<tr>
<td>7.8. Open Cypher</td>
<td>100</td>
</tr>
<tr>
<td>7.9. Gremlin API</td>
<td>101</td>
</tr>
<tr>
<td>7.10. GraphQL</td>
<td>103</td>
</tr>
<tr>
<td>7.11. MongoDB API</td>
<td>107</td>
</tr>
<tr>
<td>7.11.1. MongoDB Query Language</td>
<td>108</td>
</tr>
<tr>
<td>7.11.2. MongoDB Protocol Plugin</td>
<td>109</td>
</tr>
<tr>
<td>7.12. Redis API</td>
<td>109</td>
</tr>
<tr>
<td>7.12.1. Installation</td>
<td>110</td>
</tr>
<tr>
<td>7.12.2. How it works</td>
<td>110</td>
</tr>
<tr>
<td>8. SQL</td>
<td>115</td>
</tr>
<tr>
<td>8.1. SQL - DELETE</td>
<td>180</td>
</tr>
<tr>
<td>8.2. SQL - INSERT</td>
<td>186</td>
</tr>
<tr>
<td>8.3. SQL - MATCH</td>
<td>189</td>
</tr>
<tr>
<td>8.4. SQL - SELECT</td>
<td>206</td>
</tr>
<tr>
<td>8.5. SQL - UPDATE</td>
<td>221</td>
</tr>
<tr>
<td>9. Security</td>
<td>226</td>
</tr>
<tr>
<td>10. Comparison with other DBMSs</td>
<td>230</td>
</tr>
<tr>
<td>11. Appendixes</td>
<td>233</td>
</tr>
<tr>
<td>11.1. Data Types</td>
<td>233</td>
</tr>
</tbody>
</table>
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
- Checkout the Open Source project on GitHub
- What is Multi-Model?
- ArcadeDB supports the following models:
  - Graph Model (compatible with Gremlin and OrientDB SQL)
  - Document Database (compatible with the MongoDB driver and MongoDB queries)
  - Key/Value (compatible with the Redis driver)
  - Time Series
- 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
- 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)
- Tutorials: Java Tutorial
- Tools: Working with the Console Tutorial
- Misc: Docker, Kubernetes
- Migrating from OrientDB
Chapter 2. Introduction

2.1. What is ArcadeDB?

ArcadeDB is the new generation of DBMS that runs on pretty much every hardware/software configuration. ArcadeDB is Multi-Model, that means it can work with graphs, documents and other forms 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 high-level API. The result is that ArcadeDB does not allocate many objects at run-time on the Heap, so the Garbage Collection doesn’t do much. At the same time, it’s still able to run on pretty much every sw/hw configuration and leverage of 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 the records
- Traverses parts of or entire trees and graphs of records in milliseconds
- Traversing speed is not affected by 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) and Microsoft Azure consoles (coming soon).

Is ArcadeDB FREE?

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

Which is more likely to have better quality? A DBMS created and tested by a handful of developers or one tested by over 100,000 developers globally? When code is public, everyone can scrutinize, test, report and resolve issues. All things Open Source move faster compared to the proprietary world.

2.2. Multi Model

Edit this section
The ArcadeDB engine supports **Graph**, **Document**, **Key/Value**, and **Time-Series** 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 abilities, which combine all the features of the four models into the core. These abilities are not just interfaces to the database engine, but rather the engine itself was built to support all four 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. In the table below, you can find a comparison between the graph model, the relational...
data model, and the ArcadeDB graph model:

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Graph Model</th>
<th>ArcadeDB Graph Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Vertex and Edge Types</td>
<td>Type</td>
</tr>
<tr>
<td>Row</td>
<td>Vertex</td>
<td>Vertex</td>
</tr>
<tr>
<td>Column</td>
<td>Vertex and Edge property</td>
<td>Vertex and Edge property</td>
</tr>
<tr>
<td>Relationship</td>
<td>Edge</td>
<td>Edge</td>
</tr>
</tbody>
</table>

### 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](Concepts.md#type)" and "Bucket" 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 "Link" as 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:

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Document Model</th>
<th>ArcadeDB Document Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Collection</td>
<td>Type or Bucket</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
<td>Document</td>
</tr>
<tr>
<td>Column</td>
<td>Key/value pair</td>
<td>Document property</td>
</tr>
<tr>
<td>Relationship</td>
<td>not available</td>
<td>Relationships</td>
</tr>
</tbody>
</table>

### 2.2.3. Key/Value Model

This is the simplest model of the three. 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 typeic Key/Value model. The typeic 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:

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Key/Value Model</th>
<th>ArcadeDB Key/Value Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Bucket</td>
<td>Bucket</td>
</tr>
<tr>
<td>Row</td>
<td>Key/Value pair</td>
<td>Document</td>
</tr>
<tr>
<td>Column</td>
<td>not available</td>
<td>Document field or Vertex/Edge property</td>
</tr>
<tr>
<td>Relationship</td>
<td>not available</td>
<td>Relationships</td>
</tr>
</tbody>
</table>

2.2.4. Time-Series Model

Coming Soon.

2.3. Run ArcadeDB

You can run ArcadeDB in the following ways:

• On the cloud (coming soon), by using ArcadeDB instance on Amazon AWS, Microsoft Azure and Google Cloud Engine marketplaces

• On-premise, on your servers, any OS is good. You can run with Docker, Kubernetes or by just run the server script.

• 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 configuration with at least 3 servers, with 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 of 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 crashes, but don’t worry, ArcadeDB uses a WAL to recover partially committed transactions. Your data is safe.

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. We have created the RemoteDatabase class in Java that hide the HTTP calls. Feel free to use it if your application is running on a JVM.

**High Availability (HA)**

You can spin up as many ArcadeDB servers 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.
Chapter 3. Main Concepts

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,

```json
{
  "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.

**Types**

The concept of the 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.

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 Operative System has any limitation with the number of files supported and opened at the same time (ulimit for Unix-like systems).

**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 cannot share the same bucket.

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 cases, 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 helps 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:
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.

```python
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:

```python
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:

```python
CREATE BUCKET 'Customer_Europe_Italy'
CREATE BUCKET 'Customer_Europe_Spain'
```

Now get all the customers that live in Italy.

```python
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.

```python
arcadeDB> SELECT FROM BUCKET:Customer_Europe_Italy, Customer_Europe_Spain
```

### Relationships

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

**Referenced Relationships**

In Relational databases, tables are linked through `JOIN` commands, which can prove costly on computing resources. ArcadeDB manages relationships natively without computing JOIN's by storing direct links to the target objects of the relationship. This boosts the load speed for the entire graph of connected objects, such as in Graph and Object database systems.

Example

<table>
<thead>
<tr>
<th>Customer Record A</th>
<th>Record B Invoice</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID  #5:23</td>
<td>RID  #10:2</td>
</tr>
</tbody>
</table>

**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,

<table>
<thead>
<tr>
<th>Record A</th>
<th>Record B</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE=Account</td>
<td>TYPE=Address</td>
</tr>
<tr>
<td>RID  #5:23</td>
<td>NO RID</td>
</tr>
</tbody>
</table>

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.

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.

Transactions

A transaction comprises a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions. Transactions in a database environment have two main purposes:

- to provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status
- to provide isolation between programs accessing a database concurrently. If this isolation is not provided, the program’s outcome are possibly erroneous.

A database transaction, by definition, must be atomic, [consistent](#consistency), [isolated](#isolation) and [durable](#durability). 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.

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:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Client/Thread 1</th>
<th>Client/Thread 2</th>
<th>Version of page containing record X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Begin of Transaction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>read(x)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Begin of Transaction</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>read(x)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>write(x)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>commit</td>
<td></td>
<td>10 → 11</td>
</tr>
<tr>
<td>7</td>
<td>write(x)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>commit</td>
<td></td>
<td>10 → 11 = Error, in database x already is at 11</td>
</tr>
</tbody>
</table>

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
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:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Client/Thread 1</th>
<th>Client/Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Begin of Transaction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>read(x)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Begin of Transaction</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>read(x)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>write(x)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>7</td>
<td>read(x)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

At operation 7 the client 1 continues to read the same version of x read in operation 2.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Client/Thread 1</th>
<th>Client/Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Begin of Transaction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>read(x)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Begin of Transaction</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>read(y)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>write(y)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>7</td>
<td>read(y)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

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 Orient 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 OConcurrentModificationException 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.

**Inheritance**

Unlike many Object-relational mapping tools, ArcadeDB does not split documents between different types. Each document resides in one or a number of buckets associated with its specific type. When you execute a query against a type that has subtypes, OrientDB 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. This is an abstract concept that applies to both [POJO's](../java/Object-DB-Interface.md#inheritance) and [Documents](../java/Document-Database.md).
For example,

```java
DocumentType account = database.getSchema().createDocumentType("Account");
DocumentType company = database.getSchema().createDocumentType("Company").addParent(account);
```

**Using Polymorphic Queries**

By default, OrientDB treats all queries as polymorphic. Using the example above, you can run the following query from the console:

```sql
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, OrientDB limits the result-sets to only the records of the buckets contained in the `bucketIds` property. For example,

```sql
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.
Chapter 4. Server

4.1. Server

To start ArcadeDB as a server run the script `server.sh` under the `bin` directory of ArcadeDB distribution. If you're using MS Windows OS, replace `server.sh` with `server.bat`.

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

PLAY WITH DATA arcadedb.com
```

INFO [ArcadeDBServer] ArcadeDB Server v21.9.1 (build 258eb/163044331/main) is starting up...
INFO [ArcadeDBServer] Starting ArcadeDB Server with plugins [] ...
INFO [ArcadeDBServer] - JMX Metrics 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

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)...
INFO [undertow] starting server: Undertow - 2.2.10.Final
INFO [xnio] XNIO version 3.8.4.Final
INFO [nio] XNIO NIO Implementation Version 3.8.4.Final
INFO [threads] JBoss Threads version 3.1.0.Final
INFO [HttpServer] <ArcadeDB_0> - HTTP Server started (host=0.0.0.0 port=2480)
INFO [ArcadeDBServer] <ArcadeDB_0> ArcadeDB Server started (CPUs=16 MAXRAM=2.00GB)
```

By default, the following components start with the server:

- JMX Metrics, to monitor server performance and statistics
- HTTP Server, that listens on port 2480 by default. if 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:

```
~/arcadedb $ 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. 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`.

```
~/arcadedb $ 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".

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

- **HTTP/JSON**
- **Postgres Driver**
4.2. Docker

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

```
```

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 5bb2ef9d4704ae9b55b4dfa08a00568596ab8a89fd8a2135f2a15736c891d248 bin/console.sh
```

4.2.1. Quick start with the demo 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 that download and install in a few seconds the OrientDB's OpenBeer dataset:

```
```

Now point your browser on [https://localhost:2480](https://localhost:2480) and you'll see ArcadeDB Studio. Now enter "root" as a user and "playwithdata" as a password.
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.
Tuning

In general, the RAM allocated for the JVM should be $\approx 80\%$ 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 ...
```
4.3. High Availability

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=localhost,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

4.3.1. Architecture

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

![Figure 1. Cluster of Servers](image)

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](image)

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](image)

### 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

Servers do not see each other in the same LAN

Check UDP Broadcast protocol is enabled in your LAN on both firewalls and routers.

4.3.5. HA Settings

The following settings are used by the High Availability module:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>arcadedb.ha.clusterName</td>
<td>Cluster name. Useful in case of multiple clusters in the same network</td>
<td>arcadedb</td>
</tr>
<tr>
<td>arcadedb.ha.serverList</td>
<td>Servers in the cluster as a list of &lt;hostname/ip-address:port&gt; items separated by comma. Example: localhost:2424,192.168.0.1:2424. If not specified, auto-discovery is enabled</td>
<td>NOT DEFINED (auto discovery is enabled by default)</td>
</tr>
<tr>
<td>arcadedb.ha.quorum</td>
<td>Default quorum between 'none', 1, 2, 3, 'majority' and 'all' servers</td>
<td>MAJORITY</td>
</tr>
<tr>
<td>arcadedb.ha.quorumTimeout</td>
<td>Timeout waiting for the quorum</td>
<td>10000</td>
</tr>
<tr>
<td>arcadedb.ha.k8s</td>
<td>The server is running inside Kubernetes</td>
<td>false</td>
</tr>
<tr>
<td>arcadedb.ha.k8sSuffix</td>
<td>When running inside Kubernetes use this suffix to reach the other servers. Example: arcadedb.default.svc.cluster.local</td>
<td></td>
</tr>
<tr>
<td>arcadedb.ha.replicationQueueSize</td>
<td>Queue size for replicating messages between servers</td>
<td>512</td>
</tr>
<tr>
<td>arcadedb.ha.replicationFileSize</td>
<td>Maximum file size for replicating messages between servers&quot;</td>
<td>1GB</td>
</tr>
<tr>
<td>arcadedb.ha.replicationChunkMaxSize</td>
<td>Maximum channel chunk size for replicating messages between servers</td>
<td>16777216</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>arcadedb.ha.replicationIncomingHost</td>
<td>TCP/IP host name used for incoming replication connections</td>
<td>localhost</td>
</tr>
<tr>
<td>arcadedb.ha.replicationIncomingPorts</td>
<td>TCP/IP port number used for incoming replication connections</td>
<td>2424-2433</td>
</tr>
</tbody>
</table>

### 4.4. Kubernetes (k8s)

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

```bash
~/arcadedb $ kubectl create secret generic server-root-password --from-literal
=arcadedb.server.rootPassword='<password>'
```

This will set the password in the environment variable `arcadedb.server.rootPassword`. 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:

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

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

```bash
~/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:

```bash
~/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.

More coming soon.

### 4.5. Settings
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 Appendix A for all the available settings.

### 4.5.1. 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.

### 4.6. Embed a Server

Embedding the server in your JVM allows to have all the benefits of working in embedded mode with ArcadeDB (zero cost for network transport and marshalling) and still having the database accessible from the outside, such as Studio, remote API, Postgres, REDIS and MongoDB drivers.

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-engine</artifactId>
</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.

### 4.6.1. 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:

```java
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`:

```java
config.setValue(GlobalConfiguration.HA_SERVER_LIST, "localhost,192.168.10.1,192.168.10.2");
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.

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

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

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

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)
- **Apache Tinkerpop Gremlin** (only for graphs)
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 \textit{Hide Properties} button.

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

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

\textbf{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 file 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.
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. Use 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.
  
  ◦ **Import** to import a database from a path on the server filesystem or a remote path by using http:// and https:// prefix for the URL where the file is located. Look at Importer for more information.

• **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 the first 30 records** of the selected type
  
  ◦ **Display the 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/api/v1/command/{database}</td>
<td>Executes a non-idempotent command.</td>
</tr>
<tr>
<td>POST</td>
<td>/api/v1/create/{database}</td>
<td>Creates a database</td>
</tr>
<tr>
<td>GET</td>
<td>/api/v1/databases</td>
<td>Returns the list of databases the current authenticated user can access to.</td>
</tr>
<tr>
<td>POST</td>
<td>/api/v1/document/{database}</td>
<td>Creates a new document</td>
</tr>
<tr>
<td>GET</td>
<td>/api/v1/document/{database}/{rid}</td>
<td>Returns a document by record id (rid)</td>
</tr>
<tr>
<td>POST</td>
<td>/api/v1/drop/{database}</td>
<td>Drops a database</td>
</tr>
<tr>
<td>GET</td>
<td>/api/v1/query/{database}</td>
<td>Executes an idempotent commands, like SELECT ad MATCH.</td>
</tr>
<tr>
<td>GET</td>
<td>/api/v1/server</td>
<td>Returns the current HA configuration.</td>
</tr>
</tbody>
</table>
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](http://example.com)
  - Remotely by using a [Postgres driver](http://example.com) (ArcadeDB implements Postgres Wire protocol)
  - Remotely by using a [MongoDB driver](http://example.com) (only a subset of the operations are implemented)
  - Remotely by using a [Redis driver](http://example.com) (only a subset of the operations are implemented)
- Misc: Docker, Kubernetes
Chapter 6. Tools

6.1. Console

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

```bash
~/arcadebin $ bin/console.sh
ArcadeDB Console v.0.1-SNAPSHOT - Copyrights (c) 2020 Arcade Data
(https://arcadedata.com)
>
```

The console supports the following commands (you can always retrieve this help by typing `HELP` or just `?`:

- `begin` -> begins a new transaction
- `close` -> closes the database
- `create database <path>|remote:<url>` -> creates a new database
- `commit` -> commits current transaction
- `connect <path>|remote:<url>` -> connects to a database stored on `<path>`
- `info types` -> print available types
- `info transaction` -> print current transaction
- `rollback` -> rollbacks current transaction
- `quit or exit` -> exits from the console

6.1.1. Tutorial

Let's create our first database "mydb" under the "/temp" directory:

```bash
> create database /temp/mydb
{mydb}>
```

If you already have a database, you can simply connect to it:

```bash
> connect /temp/mydb
{mydb}>
```

Now let's create a "Profile" type:
Create a new type `Profile`:

```plaintext
{mydb}> create document type Profile
```

```
+-----------------+--------+
|operation |typeName|
+-----------------+--------+
|create document type|Profile |
+-----------------+--------+
Command executed in 176ms
```

Check your new type is there:

```plaintext
{mydb}> info types
```

```
AVAILABLE TYPES
+-------+--------+------------+-----------+----------+-------------+
|NAME   |TYPE    |PARENT TYPES|BUCKETS    |PROPERTIES|SYNC STRATEGY|
+-------+--------+------------+-----------+----------+-------------+
|Profile|Document|[]          |[Profile_0]|[]        |round-robin  |
+-------+--------+------------+-----------+----------+-------------+
```

Finally, create a document of type "Profile":

```plaintext
{mydb}> insert into Profile set name = 'Jay', lastName = 'Miner'
```

```
+----+-------+----+--------+
|@RID|@TYPE  |name|lastName|
+----+-------+----+--------+
|#1:0|Profile|Jay |Miner   |
+----+-------+----+--------+
Command executed in 29ms
```

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:

```plaintext
{mydb}> select from Profile
```

```
+----+-------+----+--------+
|@RID|@TYPE  |name|lastName|
+----+-------+----+--------+
|#1:0|Profile|Jay |Miner   |
+----+-------+----+--------+
Command executed in 33ms
```

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.

6.2. Backup of a database

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

Look at Backup Database SQL command for more information.

6.3. Restore a database

ArcadeDB allows to restore a database previously backed up.

<table>
<thead>
<tr>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a server is running, it ust be restarted in order to access to the restored database.</td>
</tr>
</tbody>
</table>

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

ArcadeDB is able to import automatically any dataset in the following formats:

- OrientDB database export
- Neo4j database export
- Generic XML files
• Generic JSON files
• Generic CSV files
• Generic RDF files

From file of types:

• Plain text
• Compressed with ZIP
• 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

To start importing it's super easy as providing the URL where the source file to import is located. URLs can be local paths or from the Internet by using `http` and `https`.

Example of loading the Freebase RDF dataset:

```
~/arcadedb $ create database FreeBase

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":

```
```

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:
See also:

SQL Import Database command Neo4j Importer OrientDB Importer

6.4.1. OrientDB Importer

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
```

6.4.2. Neo4j Importer

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": {
  }
}
{
  "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
~/arcadedb

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
```
6.5. Upgrade ArcadeDB

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 a database (POST)`).
3. Copy the databases directory from the old server to the new one
4. Start the new server

6.6. Downgrade to older version of ArcadeDB

In the case you need to downgrade to an older version of ArcadeDB, check the binary compatibility between the versions. ArcadeDB uses the [https://semver.org/,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 powerful 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.

<table>
<thead>
<tr>
<th>Feature</th>
<th>JVM Embedded API</th>
<th>SQL</th>
<th>Apache Gremlin</th>
<th>Cypher</th>
<th>GraphQL</th>
<th>MongoDB Query</th>
<th>Redis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>⭐⭐⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐⭐</td>
</tr>
<tr>
<td>Flexibility</td>
<td>⭐⭐⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
<td>⭐</td>
</tr>
<tr>
<td>Support for Documents</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Support for Graph</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Embedded mode</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

7.1. Java API (Embedded)

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

```xml
<dependency>
  <groupId>com.arcadedb</groupId>
  <artifactId>arcadedb-engine</artifactId>
  <version>21.10.1</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.
<table>
<thead>
<tr>
<th>Synchronous API</th>
<th>Asynchronous API</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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-threaded to use all the available cores.</td>
</tr>
</tbody>
</table>

### 7.1.1. 10-Minute Tutorial

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.

```java
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:**

```java
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:

```java
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 denies 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
}
```
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 rolled back 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 to create a document, a vertex or an edge.

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

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

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

```java
try (Database db = databaseFactory.open(); ){
    db.beginTransaction() -> {
        // CREATE A CUSTOMER INSTANCE
        MutableDocument customer = db.newDocument("Customer");
        customer.set("name", "Jay");
        customer.set("surname", "Miner");
    };
}
```
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:

```java
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:

```java
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:

```java
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:

![Diagram showing a social network with two vertices and a friendship edge]

```java
try( Database db = databaseFactory.open(); ){
    db.transaction() -> {
        MutableVertex elon = db.newVertex("User", "name", "Elon", "lastName", "Musk");
        MutableVertex steve = db.newVertex("User", "name", "Steve", "lastName", "Jobs");
        elon.newEdge("IsFriendOf", steve, true, "since", 2010);
    }
}
```

In the code snippet above, we have just created our first graph, made of 2 vertices and one edge that connects them. 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:
<table>
<thead>
<tr>
<th></th>
<th>JVM Embedded API</th>
<th>SQL</th>
<th>Apache Gremlin</th>
<th>Cypher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>***</td>
<td>**</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Flexibility</td>
<td>***</td>
<td>*</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Embedded mode</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Remote mode</td>
<td>No</td>
<td>Yes</td>
<td>Yes (through the Gremlin Server plugin)</td>
<td>Yes (through the Gremlin Server plugin)</td>
</tr>
</tbody>
</table>

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:

```java
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 \( O(\log N) \). Example:

```java
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:

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

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

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:

```java
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:

```java
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:

```java
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:

```java
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 3.4.x, you can run this query against the Apache Gremlin Server configured with ArcadeDB:

```java
g.V().has('name','Steve').has('lastName','Jobs').out('IsFriendOf');
```

For more information about Apache Gremlin:

- ArcadeDB Gremlin API support
- Introduction to Gremlin
- Getting Started with Gremlin
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:

- ArcadeDB Cypher support
- Open Cypher
- The Neo4j Cypher Manual

### 7.1.2. Schema

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()`:

```java
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:

```java
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:

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

And the same for edge types:

```java
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:

```java
Collection<DocumentType> getTypes();
DocumentType getType(String typeName);
void dropType(String typeName);
String getTypeNameByBucketId(int bucketId);
DocumentType getTypeByBucketId(int bucketId);
boolean existsType(String typeName);
```

### 7.1.3. 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:

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

### 7.1.4. 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.

Null values are not indexed, so any query that is looking for null values will not
use the index with 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 specific API to manage indexes are:

```java
Index[] createClassIndexes(SchemaImpl.INDEX_TYPE indexType, boolean unique, String typeName, String[] propertyNames);
Index[] createClassIndexes(SchemaImpl.INDEX_TYPE indexType, boolean unique, String typeName, String[] propertyNames, int pageSize);
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
- **pageSize** is the page size. If not specified, the default of 2MB is used

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

```java
Index createManualIndex(SchemaImpl.INDEX_TYPE indexType, boolean unique, String indexName, byte[] 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.

### 7.1.5. 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:
7.1.6. Embedded Documents

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

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

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

```
db.transaction( (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("companyName", "Commodore", "since", 2010);

  commodore.save();  
});
```

Modeling with a graph it would be something like this:
And this would be the code to create the types and the graph.

```javascript
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.

```javascript
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:

```java
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"));
});
```

### 7.1.7. Events

ArcadeDB allows hooking listener to the following events on records (vertices, edges, documents):

- **before is created**, by registering the interface `BeforeRecordCreateListener`
- **after is created**, by registering the interface `AfterRecordCreateListener`
- **before is 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:

```java
database.getEvents().registerListener()
```

And at specific type level by using:

```java
database.getSchema().getType(<type-name>).registerListener()
```

All the interface listeners that work **before** a record is created, 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, 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):

```java
database.getEvents().registerListener((BeforeRecordCreateListener) record -> record instanceof Vertex && record.asVertex().getBoolean("validated"));
```

The same by only for vertex type "Client":

```java
database.getSchema().getType("Client").getEvents().registerListener((BeforeRecordCreateListener) record -> record.asVertex().getBoolean("validated"));
```

## Java Reference

### 7.1.8. DatabaseFactory Class

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

### Methods

Example:

```java
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 the databases.
Syntax:

```java
void close()
```

events()

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

Syntax:

```java
boolean exists()
```

**Database create()**

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

Example:

```java
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:

```java
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:

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

```
7.1.9. **Database Interface**

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

**Methods (Alphabetic order)**

<table>
<thead>
<tr>
<th>async()</th>
<th>begin()</th>
<th>close()</th>
<th>commit()</th>
<th>deleteRecord()</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop()</td>
<td>getSchema()</td>
<td>isOpen()</td>
<td>iterateBucket()</td>
<td>iterateType()</td>
</tr>
<tr>
<td>query() positional parameters</td>
<td>query() (parameter map)</td>
<td>command() positional parameters</td>
<td>command() (parameter map)</td>
<td>lookupByKey()</td>
</tr>
<tr>
<td>lookupByRID()</td>
<td>newDocument()</td>
<td>newEdgeByKeys()</td>
<td>newVertex()</td>
<td>rollback()</td>
</tr>
<tr>
<td>scanBucket()</td>
<td>scanType()</td>
<td>transaction() default</td>
<td>transaction() with retries</td>
<td></td>
</tr>
</tbody>
</table>

**Methods (By category)**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Lifecycle</th>
<th>Query</th>
<th>Records</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction() default</td>
<td>close()</td>
<td>query() positional parameters</td>
<td>newDocument()</td>
<td>async()</td>
</tr>
<tr>
<td>transaction() with retries</td>
<td>drop()</td>
<td>query() (parameter map)</td>
<td>newVertex()</td>
<td>command() positional parameters</td>
</tr>
<tr>
<td>begin()</td>
<td>isOpen()</td>
<td>lookupByKey()</td>
<td>newEdgeByKeys()</td>
<td>command() (parameter map)</td>
</tr>
<tr>
<td>commit()</td>
<td></td>
<td>lookupByRID()</td>
<td>deleteRecord()</td>
<td>getSchema()</td>
</tr>
<tr>
<td>rollback()</td>
<td></td>
<td>iterateType()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>iterateBucket()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scanBucket()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scanType()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**async()**

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

**Syntax:**

```javascript
db.close();
```
Example:

Execute an asynchronous query:

```java
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:

```java
begin()
```

Example:

```java
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:

```java
void close()
```

Example:

```java
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:

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

drop()

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

Syntax:

```java
void drop()
```

Example:

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

getSchema()

Returns the Schema instance for the database.

Syntax:

```java
Schema getSchema()
```

Example:
db.getSchema().createVertexType("Song");

isOpen()

Returns true if the database is open, otherwise false.

Syntax:

boolean isOpen()

Example:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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-
• **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:

```java
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 rolled back 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:

```java
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 rather this method returns an Iterator<Record> that fetches the records only when .next() is called.

Syntax:

Iterator<Record> iterateBucket( String bucketName )

Example:

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

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

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

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

Example:

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

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

    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 rather this method returns an Iterator<Record> that fetches the records only when .next() is called.

Syntax:

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

Example:

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

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

Iterator<Record> result = db.iterateType("V", true);
while( result.hasNext() ){
    Record record = result.next();
    String age = (String) record.get("age");
    AtomicInteger counter = aggregate.get(age);
    if (counter == null) {
        counter = new AtomicInteger(1);
        aggregate.put(age, counter);
    } else
        counter.incrementAndGet();
}
```

lookupByKey( type, properties, keys )

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

Syntax:

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

Where:

- **type** type name
- **properties** array of property names to match
- **keys** array of keys
It returns a `Cursor<RID>` (like an iterator).

Examples:

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

```java
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:

```java
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:

```java
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:

```java
MutableDocument newDocument( typeName )
```

Where:
• **typeName** type name

It returns a **MutableDocument** instance.

**Examples:**

Create a new document of type "Customer":

```java
MutableDocument doc = db.newDocument("Customer");
doc.set("name", "Jay");
doc.set("surname", "Miner");
doc.save();
```

**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:**

```java
MutableVertex newVertex( typeName )
```

Where:

• **typeName** type name

It returns a **MutableVertex** instance.

**Examples:**

Create a new document of type "Customer":

```java
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:**

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

```java
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:

```java
rollback()
```

Example:

```java
db.begin();
try{
```java
// 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 bucket. 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:**

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

**Example:**

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

```java
    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:**

```java
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":

```java
    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 method 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

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

<table>
<thead>
<tr>
<th>command(…)</th>
<th>command(Map&lt;K,V&gt;)</th>
<th>query(…) positional parameters</th>
<th>query(Map&lt;K,V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>positional parameters</td>
<td>parameter map</td>
<td>parameter parameters</td>
<td>parameter map</td>
</tr>
<tr>
<td>transaction()</td>
<td>transaction(retries)</td>
<td>transaction(retries, callbacks)</td>
<td>createRecord()</td>
</tr>
<tr>
<td>updateRecord()</td>
<td>deleteRecord()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```java
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:

```java
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:

```java
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");
```

**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:**

```java
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:

```java
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:

```java
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:

```java
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.toJSONString());
        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:

```java
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.toJSONString());
        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:

```java
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>`.
- **callback** is 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:

```java
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.toJSONString());
        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:**

```java
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.

```java
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:**

```java
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.

```java
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:

```java
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.

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

### 7.2. HTTP/JSON Protocol

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 command 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 "root".

Server-Side Transactions

ArcadeDB implements server-side transaction over HTTP stateless protocol by using sessions. A session is created with the `/begin` command and returns a session id in the response header (example `arcadedb-session-id: A5-ee056170-dc9b-4956-8d71-d7cfa81900d4`). 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 rolls back and purges expired transactions.

**Streaming Change Events**

This feature presently only supports single server deployments. 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>REQUIRED</td>
<td>subscribe or unsubscribe.</td>
</tr>
<tr>
<td>database</td>
<td>REQUIRED</td>
<td>The database name.</td>
</tr>
<tr>
<td>type</td>
<td>Optional</td>
<td>The entity type to filter by.</td>
</tr>
<tr>
<td>changeTypes</td>
<td>Optional</td>
<td>Array of change types you’d like to receive. Must be create, update, or delete.</td>
</tr>
</tbody>
</table>

Example: to subscribe to all changes (create, update, delete) for the type *Movie* in the database *movies*, you’d send this:

```json
{"action": "subscribe", "database": "movies", "type": "Movie"}
```

If instead, you only wanted updates, send this:

```json
{"action": "subscribe", "database": "movies", "type": "Movie", "changeTypes": ["update"]}
```

If you wanted every change on the database (use with caution!):

```json
{"action": "subscribe", "database": "movies"}
```
Once subscribed, you will get JSON messages for any matching changes with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>The source database.</td>
</tr>
<tr>
<td>changeType</td>
<td>create, update or delete.</td>
</tr>
<tr>
<td>record</td>
<td>The full record that generated the change event.</td>
</tr>
</tbody>
</table>

## Tutorial

Let's first create an empty database "school" on the server:

```bash
curl -X POST http://localhost:2480/api/v1/create/school
   --user root:root
```

Now let’s create the type "Class":

```bash
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:root
```

We could insert our first Class by using SQL:

```bash
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:root
```

Or better, using parameters with SQL:

```bash
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:root
```

Or by using the `api/v1/document` API:
7.2.1. Reference

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

Example:

```
curl -X POST http://localhost:2480/api/v1/begin/school
--user root:root
```

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.

Execute a 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"
```
The payload, as a JSON, accepts the following parameters:

- **language** is the query language used, between "sql", "cypher", "gremlin", "neo4j" and any other language supported by ArcadeDB and available at runtime.
- **command** the command to execute in encoded format
- **limit** (optional) is the maximum number of results to return
- **params** (optional), is the map of parameters to pass to the query engine
- **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

Example of insertion of a new Client by using parameters:

```bash
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:root
```

**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 a 500 Internal server error is returned.

Example:

```bash
curl -X POST http://localhost:2480/api/v1/commit/school
   -H "arcadedb-session-id: AS-ee056170-dc9b-4956-8d71-d7cfa01900d4"
--user root:root
```
Create a database (POST)

URL Syntax: /api/v1/create/{database}

Where:

- **database** is the database name

Example to create a new database:

```
curl -X POST http://localhost:2480/api/v1/create/school
    --user root:root
```

List of databases (GET)

Returns the list of databases the current user can access to.

URL Syntax: /api/v1/databases

Example:

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

The response contains the name of available database in the "result" array. Example:

```
{"result":["Movies","Universe"],"user":"root","version":"22.3.1-SNAPSHOT (build 0454b5b18e33a79ccaeff780e75adf5af4c1d25/1641970368529/main)"}
```

Create a document (POST)

URL Syntax: /api/v1/document/{database}

Where:

- **database** is the database name

The Payload is the JSON document to insert.

Example of inserting a new document of type "Person":

```
  -d '
    {"@type": "Person", "name": "Jay", "surname": "Miner", "age": 69}"
  -H "Content-Type: application/json"
  --user root:root
```
Load a document (GET)

URL Syntax: /api/v1/document/{database}/{rid}

Where:

- **database** is the database name

Example of retrieving a document by RID:

```
--user root:root
```

The output will be:

```
{
  "@rid": "#3:4",
  "@type": "Person",
  "name": "Jay",
  "surname": "Miner",
  "age": 69
}
```

Open a database (POST)

Opens a database on the server. By default, all the databases under the databases/ directory on the server are loaded at startup. You can manually load the databases by setting arcadedb.server.databaseLoadAtStartup=false and invoking the open command on the databases you are going to use. Also, you can open a database previously closed because of a restore database command.

URL Syntax: /api/v1/open/{database}

Where:

- **database** is the database name

Example of opening the database "school":

```
curl -X POST http://localhost:2480/api/v1/open/school
--user root:root
```

Close a database (POST)

Closes a database on the server. Use this command to free resources in case there are many databases managed by the server. Also, close the database before a restore of the database.

URL Syntax: /api/v1/close/{database}

Where:

- **database** is the database name

Example of closing the database "school":

```
curl -X POST http://localhost:2480/api/v1/close/school
--user root:root
```
curl -X POST http://localhost:2480/api/v1/close/school
   --user root:root

Drop a database (POST)

URL Syntax: /api/v1/drop/{database}

Where:

- **database** is the database name

Example of deleting the database "school":

curl -X POST http://localhost:2480/api/v1/drop/school
   --user root:root

Execute a query (GET)

This command allows executing idempotent commands, like SELECT and MATCH:

URL Syntax 1: /api/v1/query/{database}

Where:

- **database** is the database name

The payload, as a JSON, accepts the following parameters:

- **language** is the query language used, between "sql", "cypher", "gremlin", "neo4j" 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

Example of retrieving the class with name "English" by using parameters:

curl -X POST http://localhost:2480/api/v1/command/company
   -d '{ "language": "sql", "command": "select from Class where name = :name",
        params: { "name": "English" } }'
   -H "Content-Type: application/json"
   --user root:root

Example of retrieving the class with name "English" by executing a SQL query:

curl -X POST http://localhost:2480/api/v1/query/school
   -d '{ "language": "sql", "command": "select from Class where name = "English\""
   }'
   -H "Content-Type: application/json"
There is also this alternative syntax that takes the language and command in the URL:

URL Syntax 2: `/api/v1/query/{database}/{language}/{command}`

Where:

- **database** is the database name
- **language** is the query language used. Only "sql" is available with latest release
- **command** the command to execute in encoded format

**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 a 500 Internal server error is returned.

Example:

```
curl -X POST http://localhost:2480/api/v1/rollback/school
   -H "arcadedb-session-id: AS-ee056170-dc9b-4956-8d71-d7cfa01900d4"
   --user root:root
```

**Get server information (GET)**

Returns the current HA configuration.

URL Syntax: `/api/v1/server`

Example:

```
curl -X GET http://localhost:2480/api/v1/server
   --user root:root
```

Return:

```json
{ "leaderServer": "europe0", "replicaServers" : ["usa0", "usa1"]}
```
7.3. Postgres Protocol Plugin

ArcadeDB Server supports a subset of the Postgres wire protocol, such as connection and queries.

If you're using ArcadeDB as embedded, please add the dependency to the `arcadedb-postgresw` library. If you're using Maven include this dependency in your `pom.xml` file.

```xml
<dependency>
    <groupId>com.arcadedb</groupId>
    <artifactId>arcadedb-postgresw</artifactId>
    <version>21.10.1</version>
</dependency>
```

To start the Postgres plugin, enlist it in the `server.plugins` settings. To specify multiple plugins, use the comma `,` as separator. Example:

```bash
~/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:

```bash
docker run --rm -p 2480:2480 -p 2424:2424 -p 5432:5432 \
    -e arcadedb.server.rootPassword=playwithdata \
    -e arcadedb.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
7.4. Execute multiple query languages

By default the Postgres driver interpret 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:

```csharp
{graphql}{ bookById(id: "book-1") { id name authors { firstName, lastName } } }
```

Example to use Cypher:

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

Example of using Gremlin:

```csharp
{gremlin}g.V()
```

7.5. 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.

7.5.1. Transactions

Enabling auto commit to false is not 100% supported. With JDBC, leave the default settings or set:
7.6. 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. If the tool that you use to work with Postgres is not compatible with ArcadeDB, please open an issue.

7.6.1. JetBrains DataGrip/Database Plugin

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

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 squiggles!

7.6.2. psql

Postgres’s `psql` tool works out of the box, just like with a “real” Postgres server.

Connect like this: `psql -h localhost -U root movies`

After authenticating, you can run queries as normal.

7.7. Connect with JDBC Driver

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

```java
Class.forName("org.postgresql.Driver");

Properties props = new Properties();
props.setProperty("user", "user");
props.setProperty("password", "password");
props.setProperty("ssl", "false");
```
```java
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.executeUpdate();
        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));
            }
        }
    }
}
```

### 7.8. Open Cypher

ArcadeDB supports Open Cypher as query engine, but it doesn’t support the 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 or by using the Postgres driver.

**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:

```java
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:

```
```

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

```
```

For more information about Cypher:

- Open Cypher
- The Neo4j Cypher Manual

7.9. Gremlin API

ArcadeDB supports Gremlin v3.5.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>21.12.1</version>
</dependency>
```
Gremlin from Java API

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

```java
ResultSet result = database.query("gremlin", "g.V().has('name','Steve').has('lastName','Jobs').out('IsFriendOf')");
```

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','Jobs').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 `server.sh` with `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
  -e arcadedb.server.rootPassword=playwithdata
  -e "arcadedb.server.defaultDatabases=graph[root]{import:https://github.com/ArcadeData/arcadedb-datasets/raw/main/orientdb/OpenBeer.gz}"
  arcadedata/arcadedb:latest
```

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

For more information about Gremlin:

- Introduction to Gremlin
- Getting Started with Gremlin
- The Gremlin Console
- Gremlin Recipes
- PRACTICAL GREMLIN: An Apache TinkerPop Tutorial

### 7.10. GraphQL

ArcadeDB Server supports a subset of the GraphQL specification. Please open an issue or a discussion on GitHub to increase the support for GraphQL.

If you’re using ArcadeDB as embedded, please add the dependency to the `arcadedb-graphql` library.
If you're using Maven include this dependency in your `pom.xml` file.

```xml
<dependency>
  <groupId>com.arcadedb</groupId>
  <artifactId>arcadedb-graphql</artifactId>
  <version>21.12.1</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:
  ```java
  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:
  ```java
  {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:

```java
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"):

```java
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):

```java
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:

```java
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:

```graphql
bookByName(bookNameParameter: String): Book @sql(statement: "select from Book where name = :bookNameParameter")
```

Invoke the query defined above passing the book name as parameter:

```java
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:

```graphql
bookByName(bookNameParameter: String): Book @gremlin(statement: "g.V().has('name', bookNameParameter")
```

Invoke the query defined above passing the book name as parameter:

```java
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:

```graphql
bookByName(bookNameParameter: String): Book @cypher(statement: "MATCH (b:Book {name: ")
```
Invoke the query defined above passing the book name as parameter:

```java
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:

```java
{ bookById(id: "book-1")
  {
    rid @rid
    id
    name
    authors {
      firstName
      lastName
    }
  }
}
```

### 7.11. 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.

```xml
<dependency>
  <groupId>com.arcadedb</groupId>
  <artifactId>arcadedb-mongodbw</artifactId>
  <version>21.10.1</version>
</dependency>
```
7.11.1. 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:

```java
// 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' } } ], $orderBy: { id: 1 } } }")
) { resultset.hasNext(); ++i) {
    Result doc = resultset.next();
    ...
}
```

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:

```
```
You can also execute the same query in HTTP POST, passing the language and query in payload:

```bash
```

### 7.11.2. 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:

```bash
~/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):

```bash
```

The Server output will contain this line:

```
```

### 7.12. Redis API

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.

```xml
<dependency>
  <groupId>com.arcadedb</groupId>
  <artifactId>arcadedb-redisw</artifactId>
  <version>21.10.1</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.

### 7.12.1. 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:

```
```

The Server output will contain this line:

```
```

### 7.12.2. 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

![Diagram of Redis commands](image)

**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)
- **GET**, Returns the value associated with a key
- **GETDEL**, Remove and returns 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**, Sets 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’,’firstName’:’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’,’firstName’:’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’,’firstName’"
```
To retrieve the document inserted above by its 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","firstName":"Jay","lastName":"Miner"}"
```

You can also get multiple records 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","firstName":"Jay","lastName":"Miner"}"
"{\@rid":"#1:1","\@type":"Account","id":232,"email":"jay.miner@commodore.com","firstName":"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","firstName":"Jay","lastName":"Miner"}"
"{\@rid":"#1:1","\@type":"Account","id":232,"email":"jay.miner@commodore.com","firstName":"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**, to delete one or more records by a key, a composite key or record's id
- **HGET**, to retrieve a record by a key, a composite key or record's id
- **HMGET**, to retrieve multiple records by a key, a composite key or record's id
- **HSET**, to create and update one or more records by a key, a composite key or record's id
**Settings**

To change the host where the Redis protocol is listening, set the setting `arcadedb.redis.host`. By default, it is `0.0.0.0` which means listen to all the configured network interfaces. To change the default port (6379) set `arcadedb.redis.port`. 
# Chapter 8. SQL

## Commands

<table>
<thead>
<tr>
<th>CRUD</th>
<th>Graph</th>
<th>Schema &amp; Indexes</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>CREATE VERTEX</td>
<td>CREATE TYPE</td>
<td>CREATE BUCKET</td>
</tr>
<tr>
<td>INSERT</td>
<td>CREATE EDGE</td>
<td>ALTER TYPE</td>
<td>ALTER BUCKET</td>
</tr>
<tr>
<td>UPDATE</td>
<td>MATCH</td>
<td>DROP TYPE</td>
<td>DROP BUCKET</td>
</tr>
<tr>
<td>DELETE</td>
<td></td>
<td>CREATE PROPERTY</td>
<td>ALTER DATABASE</td>
</tr>
<tr>
<td>TRAVERSE</td>
<td></td>
<td>ALTER PROPERTY</td>
<td>CREATE DATABASE (console only)</td>
</tr>
<tr>
<td>TRUNCATE TYPE</td>
<td></td>
<td>DROP PROPERTY</td>
<td>DROP DATABASE (console only)</td>
</tr>
<tr>
<td>TRUNCATE BUCKET</td>
<td></td>
<td>CREATE INDEX</td>
<td>BACKUP DATABASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REBUILD INDEX</td>
<td>IMPORT DATABASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DROP INDEX</td>
<td>EXPORT DATABASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CHECK DATABASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ALIGN DATABASE</td>
</tr>
</tbody>
</table>

## Functions

<table>
<thead>
<tr>
<th>Graph</th>
<th>Math</th>
<th>Collections</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>out()</td>
<td>eval()</td>
<td>set()</td>
<td>date()</td>
</tr>
<tr>
<td>in()</td>
<td>min()</td>
<td>map()</td>
<td>sysdate()</td>
</tr>
<tr>
<td>both()</td>
<td>max()</td>
<td>list()</td>
<td>format()</td>
</tr>
<tr>
<td>outE()</td>
<td>sum()</td>
<td>difference()</td>
<td>distance()</td>
</tr>
<tr>
<td>inE()</td>
<td>abs()</td>
<td>first()</td>
<td>ifnull()</td>
</tr>
<tr>
<td>bothE()</td>
<td>abs()</td>
<td>intersect()</td>
<td>coalesce()</td>
</tr>
<tr>
<td>outV()</td>
<td>avg()</td>
<td>distinct()</td>
<td>uuid()</td>
</tr>
<tr>
<td>inV()</td>
<td>count()</td>
<td>expand()</td>
<td>if()</td>
</tr>
<tr>
<td>traversedElement()</td>
<td></td>
<td>unionall()</td>
<td>traversedVertex()</td>
</tr>
<tr>
<td>median()</td>
<td>flatten()</td>
<td>traversedEdge()</td>
<td>percentile()</td>
</tr>
<tr>
<td>last()</td>
<td>shortestPath()</td>
<td>variance()</td>
<td>symmetricDifference()</td>
</tr>
<tr>
<td>dijkstra()</td>
<td>stddev()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Methods

<table>
<thead>
<tr>
<th>Conversions</th>
<th>String manipulation</th>
<th>Collections</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert()</td>
<td>append()</td>
<td>[]</td>
<td>exclude()</td>
</tr>
<tr>
<td>asBoolean()</td>
<td>charAt()</td>
<td>size()</td>
<td>include()</td>
</tr>
<tr>
<td>asDate()</td>
<td>indexOf()</td>
<td>remove()</td>
<td>javaType()</td>
</tr>
<tr>
<td>asDatetme()</td>
<td>left()</td>
<td>removeAll()</td>
<td>toJSON()</td>
</tr>
<tr>
<td>asDecimal()</td>
<td>right()</td>
<td>keys()</td>
<td>type()</td>
</tr>
<tr>
<td>asFloat()</td>
<td>prefix()</td>
<td>values()</td>
<td>asInteger()</td>
</tr>
<tr>
<td>trim()</td>
<td>asList()</td>
<td>replace()</td>
<td>asLong()</td>
</tr>
<tr>
<td>length()</td>
<td>asMap()</td>
<td>subString()</td>
<td>asSet()</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td>asString()</td>
<td>toUpperCase()</td>
<td>normalize()</td>
</tr>
</tbody>
</table>

Introduction

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 the SQL-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.

If you are not yet familiar with SQL, we suggest you to get the course on KhanAcademy.

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
- Conditions
- Where clause
- Operators
- Pagination
- Batch
- Match for traversing graphs

**ArcadeDB SQL dialect**

ArcadeDB supports SQL as a query language with some differences compared with SQL. 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.

If you want to learn SQL, there are many online courses such as:

- Online course Introduction to Databases by Jennifer Widom from Stanford university
- Introduction to SQL at W3 Schools
- Beginner guide to SQL
- SQLCourse.com
- YouTube channel Basic SQL Training by Joey Blue

To know more, look to ArcadeDB SQL Syntax.
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:

```sql
SELECT * FROM Employee A, City B
WHERE A.city = B.id
AND B.name = 'Rome'
```

In ArcadeDB, an equivalent operation would be:

```sql
SELECT * FROM Employee WHERE city.name = 'Rome'
```

This is much more straightforward 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:

```sql
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:

```sql
SELECT * FROM Employee WHERE city.country.name = 'Italy'
```

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:

```sql
SELECT * FROM Customer
```

In ArcadeDB, the * is optional:

```sql
SELECT FROM Customer
```

See SQL-Projections.
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 (typees 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 typees, 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 fetchplan.

**IMPORTANT** - 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}] /* see the space before and after the + sign */

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 LINKLIST or a LINKSET. In some
cases you can be interested in the expanded object intead of the RID.
Let’s clarify this with an example. This is our dataset:
@rid

name

surname

#12:0

foo

fooz

122

parent


Given this query:

```sql
SELECT name, parent FROM TheType WHERE name = 'baz'
```

The result is

```json
{
  "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:

```sql
SELECT name, parent.name FROM TheType WHERE name = 'baz'
```

```json
{
  "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.

```sql
SELECT name, parent:{name} FROM TheType WHERE name = 'baz'
```

```json
{
  "name": "baz",
  "parent": {
    "name": "bar"
  }
}
```

or with multiple attributes
SELECT name, parent:{name, surname} FROM TheType WHERE name = 'baz'

```json
{
  "name": "baz",
  "parent": {
    "name": "bar",
    "surname": "barz"
  }
}
```

or using a wildcard

SELECT name, parent:{*} FROM TheType WHERE name = 'baz'

```json
{
  "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'

```json
{
  "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'

```json
{
  "name": "baz",
  "parent": {
    "name": "bar",
    "parent": #12:0
  }
}
```
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'
```
Functions

SQL Functions are all the functions bundled with OrientDB SQL engine. Look also to [SQL Methods](SQL-Methods.md).

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**
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:
  ```sql
  SELECT out() FROM V
  ```
- Get all the incoming vertices connected with edges with label (class) "Eats" and "Favorited" from all the Restaurant vertices in Rome:
  ```sql
  SELECT out('Eats','Favorited') FROM Restaurant WHERE city = 'Rome'
  ```

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 V vertices:
  ```sql
  SELECT in() FROM V
  ```
- Get all the incoming vertices connected with edges with label (class) "Friend" and "Brother":
  ```sql
  SELECT in('Friend','Brother') FROM V
  ```

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 vertices connected by edges with label (class) "Friend" and "Brother":

```
SELECT both('Friend','Brother') FROM V
```

**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 vertices:

```
SELECT outE() FROM V
```

Get all the outgoing edges of type "Eats" from all the SocialNetworkProfile vertices:

```
SELECT outE('Eats') FROM SocialNetworkProfile
```

**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 vertices:

```
SELECT inE() FROM V
```

Get all the incoming edges of type "Eats" from the Restaurant 'Bella Napoli':

```
SELECT inE('Eats') FROM Restaurant WHERE name = 'Bella Napoli'
```

**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 vertices:

```
SELECT bothE() FROM V
```

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 all the edges:

```
SELECT bothV() FROM E
```

**outV()**

Get outgoing vertices starting from the current record as Edge.

Syntax:
outV()

**Examples**

Get outgoing vertices from all edges

```
SELECT outV() FROM E
```

inV()

**Examples**

Get incoming vertices starting from the current record as Edge.

```
inV()
```

eval()

**Syntax:** `eval('<expression>')`

Evaluates the expression between quotes (or double quotes).

**Examples**

```
SELECT eval('price * 120 / 100 - discount') AS finalPrice FROM Order
```

coalesce()

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

**Syntax:**

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

**Examples**

```
SELECT coalesce(amount, amount2, amount3) 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(eval("name = 'John'")), "My name is John", "My name is not John") FROM Person

ifnull()

Returns the passed field/value (or optional parameter return_value_if_not_null). If field/value is not null, otherwise it returns return_value_if_null.

Syntax:

ifnull(<field/value>, <return_value_if_null>)

Examples

SELECT ifnull(salary, 0) FROM Account

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>)

Since version 2.1 the preferred operator to unwind collections is [UNWIND](SQL-Query.md#unwinding). Expand usage for this use case will probably be deprecated in next releases.

Examples

on collectinos:

SELECT EXPAND( addresses ) FROM Account.
on RIDs

```sql
SELECT EXPAND( addresses ) FROM Account.
```

This replaces the flatten() now deprecated

---

**flatten()**

> Deprecated, use the EXPAND() instead.

Extracts the collection in the field `<field>` and use it as result.

**Syntax:**

```sql
flatten(<field>)
```

**Examples**

```sql
SELECT flatten( 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**

```sql
select first( addresses ) from Account
```

---

**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**

```sql
```

132
SELECT last( addresses ) FROM Account

count()
Counts the records that match the query condition. If \* is not used as a field, then the record will be counted only if the field content is not null.

Syntax: \texttt{count(<field>)}

Examples

\begin{verbatim}
SELECT COUNT(*) FROM Account
\end{verbatim}

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: \texttt{min(<field> [, <field-n>]* )}

Examples

Returns the minimum salary of all the Account records:

\begin{verbatim}
SELECT min(salary) FROM Account
\end{verbatim}

Returns the minimum value between 'salary1', 'salary2' and 'salary3' fields.

\begin{verbatim}
SELECT min(salary1, salary2, salary3) FROM Account
\end{verbatim}

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: \texttt{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

abs()

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

Syntax: abs(<field>)

Examples

SELECT abs(score) FROM Account
SELECT abs(-2332) FROM Account
SELECT abs(999) FROM Account

avg()

Returns the average value.

Syntax: avg(<field>)

Examples

SELECT avg(salary) FROM Account

sum()

Syntax: sum(<field>)

Returns the sum of all the values returned.

Examples

SELECT sum(salary) 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](http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html). If no format is specified, then the default database format is used. To know more about it, look at [Managing Dates](../general/Managing-Dates.md).

Syntax: `date( <date-as-string> [<format>] [,<timezone>] )`

Examples

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

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](../general/Managing-Dates.md).

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

Examples

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

format()

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

Syntax: `format( <format> [,<arg1>] [,<arg-n>] )`

Examples

```
SELECT format("%d - Mr. %s %s (%s)", id, name, surname, address) FROM Account
```

#decimal()

Converts a number or a String in an absolute precision, decimal number.

Syntax: `decimal( <number> | <string> )`

Examples
SELECT decimal('99.999999999999999999') FROM Account

astar()

A*’s algorithm describes how to find the cheapest path from one node to another node in a directed weighted graph with 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:

```json
{
  direction: "OUT", //the edge direction (OUT, IN, BOTH)
  edgeTypeNames:[],
  vertexAxisNames:[],
  parallel: false,
  tieBreaker: true,
  maxDepth: 99999,
  dFactor: 1.0,
  customHeuristicFormula: 'custom_Function_Name_here' // (MANHATAN, MAXAXIS, DIAGONAL, EUCLIDEAN, EUCLIDEANNOSQR, CUSTOM)
}
```

Examples

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

dijkstra()

Returns the cheapest path between two vertices using the [Dijkstra algorithm](http://en.wikipedia.org/wiki/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
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. Since 2.0.9 and 2.1-rc2. This can also be a list of edge class names (eg. ["edgeType1", "edgeType2"] ) - `additionalParams` (since v 2.1.12), optional, here you can pass a map of additional parameters (Map<String, Object> in Java, JSON from SQL). Currently allowed parameters are - 'maxDepth': integer, maximum depth for paths (ignore path longer that 'maxDepth')

Examples on finding the shortest path between vertices #8:32 and #8:10

```sql
SELECT shortestPath(#8:32, #8:10)
```

Examples on finding the shortest path between vertices #8:32 and #8:10 only crossing outgoing edges

```sql
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'

```sql
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'

```sql
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

```sql
SELECT shortestPath(#8:32, #8:10, null, null, {"maxDepth": 5})
```
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 as degrees.

Examples

```sql
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 OrientDB, a function with parenthesis and only one field can be specified.

Examples

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

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

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

```sql
select unionall(inEdges, outEdges) from OGraphVertex where label = 'test'
```

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
```

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
```

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 difference(tags) FROM book
```

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

set()

Adds a value to a set. The first time the set is created. If `<value>` is a collection, then is merged with the set, otherwise `<value>` is added to the set.
Syntax: `set(<field>)`

**Examples**

```sql
SELECT name, set(roles.name) AS roles FROM OUser
```

**list()**

Adds a value to a list. The first time the list is created. If `<value>` is a collection, then is merged with the list, otherwise `<value>` is added to the list.

Syntax: `list(<field>)`

**Examples**

```sql
SELECT name, list(roles.name) AS roles FROM OUser
```

**map()**

Adds a value to a map. The first time the map is created. If `<value>` is a map, then is merged with the map, otherwise the pair `<key>` and `<value>` is added to the map as new entry.

Syntax: `map(<key>, <value>)`

**Examples**

```sql
SELECT map(name, roles.name) FROM OUser
```

**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 )

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 )

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 )
```

**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
```

**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
```

**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

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

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

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()

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"

```sql
SELECT * FROM State WHERE strcmpci("washington", name) = 0
```

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 [Functions](../admin/Functions.md) page.

Custom functions in Java

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

```java
// 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() {
```

144
Methods

**SQL Methods are similar to SQL Functions** but they apply to values. In Object-Oriented paradigm they are called "methods", as functions related to a type. So what's the difference between a function and a method?

This is a **SQL Functions**:

```sql
SELECT FROM sum( salary ) FROM employee
```

This is a SQL method:

```sql
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.**

[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:

```sql
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 a string to another one.

Syntax: `<value>.append(<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" and "false" is checked. If it's a number than 1 means TRUE while 0 means FALSE.

Syntax: `<value>.asBoolean()`

Applies to the following types: - string, - short, - int, - long

**Examples**

```
SELECT FROM Users WHERE online.asBoolean() = true
```

**.asDate()**

Transforms the field into a Date type.

Syntax: `<value>.asDate()`

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:
.asDateTime()

Transforms the field into a Date type but parsing also the time information.

Syntax: `<value>.asDateTime()`

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'
```

.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
```

.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

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
```

.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
```

.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
```

.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

.format()

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

Example

Get the SHA-512 of the field "password" in the type User:

```
SELECT password.hash('SHA-512') FROM User
```

.include()

Include only some properties in the resulting document.

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

Applies to the following types: - document record

Examples

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

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

```
SELECT EXPAND(@this.exclude('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:

```sql
SELECT FROM Contact WHERE phone.indexOf('+44') > -1
```

**.javaType()**

Returns the corresponding Java Type.

Syntax: `<value>.javaType()`

Applies to the following types: - any

**Examples** Prints the Java type used to store dates:

```sql
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**

```sql
SELECT FROM Actor WHERE 'Luke' IN map.keys()
```

**.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**

```sql
SELECT FROM Actors WHERE name.left(4) = 'Luke'
```
.length()

Returns the length of the string. If the string is null 0 will be returned.

Syntax: `<value>.length()`

Applies to the following types: - string

Examples

```
SELECT FROM Providers WHERE name.length() > 0
```

.normalize()

Form can be NDF, NFD, NFKC, NFKD. Default is NDF. pattern-matching if not defined is "\p{InCombiningDiacriticalMarks}+". For more information look at <a href="http://www.unicode.org/reports/tr15/tr15-23.html">Unicode Standard</a>.

Syntax: `<value>.normalize([<form>]<<,<pattern-matching>> )`

Applies to the following types: - string

Examples

```
SELECT FROM V WHERE name.normalize() AND name.normalize('NFD')
```

.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:

```sql
SELECT FROM TreeItem WHERE children.size() > 0
```

.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":

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

Substring of ArcadeDB

```sql
SELECT "ArcadeDB".substring(0,6)
```

returns Orient

.trim()

Returns the original string removing white spaces from the begin and the end.

Syntax: `<value>.trim()`
Applies to the following types: - string

Examples

```sql
SELECT name.trim() == 'Luke' 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 indent is used - `fetchPlan` is the ../java/Fetching-Strategies 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 numerers - `prettyPrint` indent the returning JSON in readable (pretty) way

Applies to the following types: - record

Examples

```sql
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

```sql
SELECT name.toLowerCase() == 'luke' FROM Actors
```
.toUpperCase()

Returns the string in upper case.

Syntax: `<value>.toUpperCase()`

Applies to the following types: - string

Examples

```
SELECT name.toUpperCase() == 'LUKE' FROM Actors
```

.type()

Returns the value's ArcadeDB Type.

Syntax: `<value>.type()`

Applies to the following types: - any

Examples

Prints the type used to store dates:

```
SELECT FROM 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)
```

Filtering

The `WHERE` condition is shared among many SQL commands.
## Syntax

```
[item] <operator> <item>
```

### Items

And `item` can be:

<table>
<thead>
<tr>
<th>What</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>Document field</td>
<td>where <code>price &gt; 1000000</code></td>
</tr>
<tr>
<td>field&lt;indexes&gt;</td>
<td>Document field part. To know more about field part look at the full syntax: <a href="#">Document-API-Property</a></td>
<td>where `tags&lt;&lt;name=&quot;Hi&quot;] or tags&lt;&lt;0-3] IN ('Hello') and employees IS NOT NULL</td>
</tr>
<tr>
<td>record attribute</td>
<td>Record attribute name with @ as prefix</td>
<td>where <code>@type = 'Profile'</code></td>
</tr>
<tr>
<td>column</td>
<td>The number of the column. Useful in Column Database</td>
<td>where <code>column(1) &gt; 300</code></td>
</tr>
<tr>
<td>any()</td>
<td>Represents any field of the Document. The condition is true if ANY of the fields matches the condition</td>
<td>where <code>any() like 'L%'</code></td>
</tr>
<tr>
<td>all()</td>
<td>Represents all the fields of the Document. The condition is true if ALL the fields match the condition</td>
<td>where <code>all() is null</code></td>
</tr>
<tr>
<td>functions</td>
<td>Any SQL-Functions between the defined ones</td>
<td>where <code>distance(x, y, 52.20472, 0.14056 ) &lt;= 30</code></td>
</tr>
<tr>
<td>$variable</td>
<td>Context variable prefixed with $</td>
<td>where <code>$depth &lt;= 3</code></td>
</tr>
</tbody>
</table>

### Record attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>@this</td>
<td>returns the record itself</td>
<td>select <code>@this.toJSON()</code> from Account</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>@rid</td>
<td>returns the RID in the form <a href="">bucket:position</a>. It’s null for embedded records. *NOTE: using @rid in where condition slow down queries. Much better to use the RID as target. Example: change this: select from Profile where @rid = #10:44 with this: select from #10:44 *</td>
<td>@rid = #11:0</td>
</tr>
<tr>
<td>@size</td>
<td>returns the record size in bytes</td>
<td>@size &gt; 1024</td>
</tr>
<tr>
<td>@type</td>
<td>returns the record type between: ‘document’, ‘column’, ‘flat’, ‘bytes’</td>
<td>@type = ‘flat’</td>
</tr>
</tbody>
</table>

**Operators**

**Conditional Operators**

<table>
<thead>
<tr>
<th>Apply to</th>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>=</td>
<td>Equals to</td>
<td>name = ‘Luke’</td>
</tr>
<tr>
<td>string</td>
<td>like</td>
<td>Similar to equals, but allow the wildcard ‘%’ that means ‘any’</td>
<td>name like ‘Luk%’</td>
</tr>
<tr>
<td>any</td>
<td>&lt;</td>
<td>Less than</td>
<td>age &lt; 40</td>
</tr>
<tr>
<td>any</td>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>age &lt;= 40</td>
</tr>
<tr>
<td>any</td>
<td>&gt;</td>
<td>Greater than</td>
<td>age &gt; 40</td>
</tr>
<tr>
<td>any</td>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>age &gt;= 40</td>
</tr>
<tr>
<td>any</td>
<td>&lt;&gt;</td>
<td>Not equals (same of !=)</td>
<td>age &lt;&gt; 40</td>
</tr>
<tr>
<td>any</td>
<td>BETWEEN</td>
<td>The value is between a range. It’s equivalent to &lt;field&gt; &gt;= &lt;from-value&gt; AND &lt;field&gt; &lt;= &lt;to-value&gt;</td>
<td>price BETWEEN 10 and 30</td>
</tr>
<tr>
<td>any</td>
<td>IS</td>
<td>Used to test if a value is NULL</td>
<td>children is null</td>
</tr>
<tr>
<td>record, string (as type name)</td>
<td>INSTANCEOF</td>
<td>Used to check if the record extends a type</td>
<td>@this instanceof 'Customer' or @type instanceof 'Provider'</td>
</tr>
<tr>
<td>Apply to</td>
<td>Operator</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| collection | IN | contains any of the elements listed | name in '
'European', 'Asiatic' |
| collection | CONTAINS | true if the collection contains at least one element that satisfy the next condition. 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') |
| 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**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>true if both the conditions are true</td>
<td>name = 'Luke' <strong>and</strong> surname like 'Sky%'</td>
</tr>
<tr>
<td>OR</td>
<td>true if at least one of the condition is true</td>
<td>name = 'Luke' <strong>or</strong> surname like 'Sky%'</td>
</tr>
<tr>
<td>NOT</td>
<td>true if the condition is false. NOT needs parenthesis on the right with the condition to negate</td>
<td><strong>not</strong>( name = 'Luke')</td>
</tr>
</tbody>
</table>

**Mathematics Operators**

<table>
<thead>
<tr>
<th>Apply to</th>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>+</td>
<td>Plus</td>
<td>age + 34</td>
</tr>
<tr>
<td>Numbers</td>
<td>-</td>
<td>Minus</td>
<td>salary - 34</td>
</tr>
<tr>
<td>Numbers</td>
<td>*</td>
<td>Multiply</td>
<td>factor * 1.3</td>
</tr>
<tr>
<td>Numbers</td>
<td>/</td>
<td>Divide</td>
<td>total / 12</td>
</tr>
<tr>
<td>Numbers</td>
<td>%</td>
<td>Mod</td>
<td>total % 3</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Command(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$parent</td>
<td>Get the parent context from a sub-query. Example: select from V let $type = ( traverse * from $parent.$current.children )</td>
<td>SQL-Query and SQL-Traverse</td>
</tr>
<tr>
<td>$current</td>
<td>Current record to use in sub-queries to refer from the parent's variable</td>
<td>SQL-Query and SQL-Traverse</td>
</tr>
<tr>
<td>$depth</td>
<td>The current depth of nesting</td>
<td>SQL-Traverse</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Command(s)</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>$path</td>
<td>The string representation of the current path. Example: 6:0.in.#5:0.out. You can also display it with → select $path from (traverse * from V)</td>
<td>SQL-Traverse</td>
</tr>
<tr>
<td>$stack</td>
<td>The List of operation in the stack. Use it to access to the history of the traversal</td>
<td>SQL-Traverse</td>
</tr>
<tr>
<td>1.1.0</td>
<td></td>
<td>$history</td>
</tr>
</tbody>
</table>

To set custom variable use the <<LET,SQL-Select-Let) keyword.

**SQL - ALIGN DATABASE**

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.

**Syntax**

```sql
ALIGN DATABASE
```

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

**Examples**

- Align the current database.

```
ArcadeDB> ALIGN DATABASE
```

**SQL - ALTER DATABASE**

Change a database setting. You can find the available settings in Settings appendix. The update is persistent.

**Syntax**
**ALTER DATABASE** `<setting-name> <setting-value>`

- `<setting-name>` Check the available settings in `Settings` appendix. Since the setting name contains . characters, surround the setting name with `\`.
- `<setting-value>` The new value to set

**Examples**

- 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**

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

**Syntax**

```
ALTER PROPERTY <type-name>..<property-name> <attribute-name> = <attribute-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
  - `nonnull = <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)
  - `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)
  - `regexp = <string>` Defines the mask to validate the input as a Regular Expression

**Examples**
• Set the custom value with key 'description':

ArcadeDB> ALTER PROPERTY User.subscribedOn CUSTOM description = 'timestamp when the user subscribed'

• Remove the custom value set above

ArcadeDB> ALTER PROPERTY User.subscribedOn CUSTOM description = null

For more information, see:

• CREATE PROPERTY
• DROP PROPERTY.

**SQL - ALTER TYPE**

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>` Defines the custom property you want to define.
• `<custom-value>` Defines the custom value for the property you want to set. Supported types are strings and numbers.

**Examples**

• Define a super-type:

ArcadeDB> ALTER TYPE Employee SUPERTYPE Person

• Add `Person` to the super types:

ArcadeDB> ALTER TYPE Employee SUPERTYPE +Person
- Remove a super-type:

ArcadeDB> ALTER TYPE Employee SUPERTYPE -Person

- Define multiple inheritances:

ArcadeDB> ALTER TYPE Employee SUPERTYPES Person, `Resource`

- 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

- Set the custom value with key 'description':

ArcadeDB> ALTER TYPE Account CUSTOM description = 'All users'

For more information, see:

- **CREATE TYPE**
- **DROP TYPE.**

**Supported Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Identifier</td>
<td></td>
<td>Changes the type name.</td>
</tr>
<tr>
<td>SUPERTYPE</td>
<td>Identifier</td>
<td></td>
<td>Defines a super-type for the type. Use NULL to remove a super-type assignment. Beginning with version 2.1, it supports multiple inheritances. To add a new type, you can use the syntax +&lt;type&gt;, to remove it use -&lt;type&gt;.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Support</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BUCKET</td>
<td>Identifier or Integer</td>
<td>+ 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.</td>
<td></td>
</tr>
</tbody>
</table>

**SQL Batch**

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
- `let <variable> = <SQL>`, to assign the result of a SQL command to a variable. To reuse the variable prefix it with the dollar sign $.
- `IF(<condition>){ <statement>; [<statement>;]* }`. Look at <<Conditional execution,SQL-batch.md#conditional-execution>>.
- `WHILE(<condition>){ <statement>; [<statement>;]* }`. Look at <<Conditional execution,SQL-batch.md#loops>>.
- `FOREACH(<variable> IN <expression>){ <statement>; [<statement>;]* }`. Look at <<Conditional execution,SQL-batch.md#loops>>.
- `SLEEP <ms>`, puts the batch in wait for `<ms>` milliseconds.
- `console.log <text>`, logs a message in the console. Context variables can be used with `${<variable>}`.
- `console.error <text>`, writes a message in the console’s standard output. Context variables can be used with `${<variable>}`.
- `console.output <text>`, writes a message in the console’s standard error. Context variables can be used with `${<variable>}`.
- `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)`

```
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 `../internals/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.

**Pessimistic transaction**

This script above used an Optimistic approach: in case of conflict it retries up top 100 times by re-executing the entire transaction (commit retry 100). To follow a Pessimistic approach by locking the records, try this:

```
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;
return $e;
```

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

```
if(<sql-predicate>){
  <statement>;
  <statement>;
  ...
}
```
<sql-predicate> is any valid SQL predicate (any condition that can be used in a WHERE clause). In current release it's mandatory to have If(), <statement> and } on separate lines, eg. the following is not a valid script

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

The right syntax is following:

```sql
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 syntax is

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

Example

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

**WHILE**

Loops while a condition is true

The syntax is

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

**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 expressed 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

```bash
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.

Examples

• Execute the integrity check of the entire database without fixing any issue found.

  ```bash
  ArcadeDB> CHECK DATABASE
  ```

• Execute the integrity check of the types 'Account' and 'Bill' without fixing any issue found.

  ```bash
  ArcadeDB> CHECK DATABASE TYPE Account, Bill
  ```

• Execute the integrity check only on the bucket 'Account_Europe' without fixing any issue found.

  ```bash
  ArcadeDB> CHECK DATABASE BUCKET Account_Europe
  ```

• Execute the integrity check of the entire database and auto fix any issues found.

  ```bash
  ArcadeDB> CHECK DATABASE FIX
  ```

SQL - CREATE BUCKET

Creates a new bucket in the database. Once created, you can use the bucket to save records by specifying its name during saves. If you want to add the new bucket to a type, follow its creation with the `ALTER TYPE` command, using the `ADDBUCKET` option.
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**

Creates a new edge in the database.

Syntax

```
CREATE EDGE <type> [BUCKET <bucket>] [UPSERT] [IF NOT EXISTS] [SET <field> = <expression>[,]*] [CONTENT {<JSON>}] [RETRY <retry>] [WAIT <pauseBetweenRetriesInMs>] [BATCH <batch-size>]
```

- `<type>` Defines the type name for the edge. Use the default edge type `E` in the event that you don't want to use sub-types.
- `<bucket>` Defines the bucket in which you want to store the edge.
- **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. The base type for an edge is `E`.

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 an edge of the type `E` between two vertices:

ArcadeDB> CREATE EDGE FROM #10:3 TO #11:4

• 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 <code type='lang-json userinput'>{ "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.

> There are several index algorithms available to determine how ArcadeDB indexes your database. For more information on these, see ../indexing/Indexes.

**Syntax**

**CREATE INDEX** [<name>]
[ IF NOT EXISTS ]
[ ON <type> (<property>*) ]
<index-type> [<key-type>]
[ NULL_STRATEGY SKIP | ERROR]

- **<name>** Defines the logical name for the index. It’s required only for manual indexes, otherwise 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. For a complete list, see ../indexing/Indexes.

- **<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.

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 a manual index to store dates:

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

- 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 typees 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

For more information, see:

• DROP INDEX

**SQL - CREATE PROPERTY**

Creates a new property in the schema. It requires that the type for the property already exist on the database.
Syntax

```
CREATE PROPERTY
<type>.<property> <data-type>
( <property-constraint> [, <property-constraint>] )* 
[IF NOT EXISTS]
```

- `<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.
- `<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)
  - 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)
  - regexp = `<string>` Defines the mask to validate the input as a Regular Expression
  - 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.

Examples

- Create the property `name` of the string type in the type `User`:

  ArcadeDB> CREATE PROPERTY User.name STRING

- Create a property formed from a list of strings called `tags` in the type `Profile`:

  ArcadeDB> CREATE PROPERTY Profile.tags LIST

- Create the property `friends`, as an embedded map:
ArcadeDB> CREATE PROPERTY Profile.friends MAP

- 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

For more information, see:

- ALTER PROPERTY
- DROP PROPERTY

Supported Types

ArcadeDB supports the following data types for standard properties:

<table>
<thead>
<tr>
<th>BOOLEAN</th>
<th>SHORT</th>
<th>DATE</th>
<th>DATETIME</th>
<th>BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>LONG</td>
<td>STRING</td>
<td>LINK</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT</td>
<td>BINARY</td>
<td>EMBEDDED</td>
<td></td>
</tr>
</tbody>
</table>

It supports the following data types for container properties.

| LIST | MAP |

For these data types, you can optionally define the contained type and type. The supported link types are the same as the standard property data types above.

**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`,SQL-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`:

```sql
ArcadeDB> CREATE DOCUMENT TYPE Account
```

• Create the vertex type `Car` to extend `Vehicle`:

```sql
ArcadeDB> CREATE VERTEX TYPE Car EXTENDS Vehicle
```

• Create the vertex type `Car`, using the bucket with name 'Car_classic' and 'Car_modern':

```sql
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:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>round-robin</code></td>
<td>Selects the next bucket in a circular order, restarting once complete.</td>
</tr>
<tr>
<td>Strategy</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>thread</td>
<td>Selects the next bucket by using the partition ( \text{mod} ) from the current thread id.</td>
</tr>
<tr>
<td>partitioned</td>
<td>Selects the smallest bucket. Allows the type to have all underlying buckets balanced on size. When adding a new bucket to an existing type, it fills the new bucket first. When using a distributed database, this keeps the servers balanced with the same amount of data. It calculates the bucket size every five seconds or more to avoid slow-downs on insertion.</td>
</tr>
</tbody>
</table>

For more information, see:

- \texttt{ALTER TYPE}
- \texttt{DROP TYPE}
- \texttt{CREATE BUCKET}

**SQL - CREATE VERTEX**

**Syntax**

\[
\text{CREATE VERTEX} \ [<\text{type}>] \ [\text{BUCKET} \ <\text{bucket}>] \ [\text{SET} \ <\text{field}> = <\text{expression}>[,]*]
\]

- \texttt{<type>} Defines the type to which the vertex belongs.
- \texttt{<bucket>} Defines the bucket in which it stores the vertex.
- \texttt{<field>} Defines the field you want to set.
- \texttt{<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 on the base type \( V \):
ArcadeDB> CREATE VERTEX

• Create a new vertex type, then create a vertex in that type:
  ArcadeDB> CREATE VERTEX TYPE V1
  ArcadeDB> CREATE VERTEX V1

• Create a new vertex within a particular bucket:
  ArcadeDB> CREATE VERTEX V1 BUCKET recent

• Create a new vertex, defining its properties:
  ArcadeDB> CREATE VERTEX SET brand = 'fiat'

• Create a new vertex of the type V1, defining its properties:
  ArcadeDB> CREATE VERTEX V1 SET brand = 'fiat', name = 'wow'

• Create a vertex using JSON content:
  ArcadeDB> CREATE VERTEX Employee CONTENT { "name" : "Jay", "surname" : "Miner" }

For more information, see:

• CREATE EDGE

8.1. SQL - DELETE

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

Syntax:

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

• 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** Allows for the processing of a DELETE on a Vertex or an Edge, without an exception error. It is not recommended to use this! If you must delete an Edge or a Vertex, use the corresponding commands DELETE EDGE or DELETE VERTEX.

**Examples:**

• Delete all records with the surname *unknown*, ignoring case:

```plaintext
ArcadeDB> DELETE FROM Profile WHERE surname.toLowerCase() = 'unknown'
```

**SQL - DROP BUCKET**

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

**Syntax**

```plaintext
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*:

```plaintext
ArcadeDB> DROP BUCKET Account
```

For more information, see:

• **CREATE BUCKET**
• **ALTER 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**

**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:
**SQL - DROP TYPE**

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 classes 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`
- `ALTER BUCKET`

**SQL - EXPLAIN**

EXPLAIN SQL command returns information about query execution planning of a specific statement, without executing the statement itself.

**Syntax**
EXPLAIN <command>

- `<command>` Defines the command that you want to profile, eg. a SELECT statement

Examples

- Profile a query that executes on a type filtering based on an attribute:

   ArcadeDB {db=foo}> explain select from v where name = 'a'

   Profiled command '[[

   executionPlan:{...},

   executionPlanAsString:

   + FETCH FROM TYPE v
     + FETCH FROM BUCKET 9 ASC
     + FETCH FROM BUCKET 10 ASC
     + FETCH FROM BUCKET 11 ASC
     + FETCH FROM BUCKET 12 ASC
     + FETCH FROM BUCKET 13 ASC
     + FETCH FROM BUCKET 14 ASC
     + FETCH FROM BUCKET 15 ASC
     + FETCH FROM BUCKET 16 ASC
     + FETCH NEW RECORDS FROM CURRENT TRANSACTION SCOPE (if any)
     + FILTER ITEMS WHERE
       name = 'a'

   ]]' in 0,022000 sec(s):

For more information, see:

- SQL Commands
- PROFILE

SQL - EXPORT DATABASE

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

Syntax

```
EXPORT DATABASE <url> [FORMAT "JSONL"|"GRAPHML"] [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

• `<FORMAT>` The format of the export as a quoted string
  - `jsonl` exports in JSONL format (one json per line)
  - `<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:

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

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

```bash
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

**Syntax**

```sql
IMPORT DATABASE [ <url> ]
```

• `<url>` Defines the location of the file to import. Use:
Examples

- Import the public OpenBeer database available as demo database for OrientDB and exported in TGZ file

ArcadeDB> IMPORT DATABASE https://github.com/ArcadeData/arcadedb-datasets/raw/main/orientdb/OpenBeer.gz

- Import the Movie database used in Neo4j's examples:


8.2. SQL - INSERT

The **INSERT** command creates a new record in the database. Records can be schema-less or follow rules specified in your model.

**Syntax:**

```
INSERT INTO [TYPE:]<type>|BUCKET:<bucket>|INDEX:<index> [(<field>[,]* VALUES (<expression>[,]*[,]*)[*]]
[SET <field> = <expression>|<sub-command>[,]*]
[CONTENT {<JSON>}]  
[RETURN <expression>]
[FROM <query>]
```

- **CONTENT** Defines JSON data as an option to set field values.
- **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.

**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')
```

• 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`.

188
• Insert a new record, adding it as an embedded document:

ArcadeDB> **INSERT INTO** Profile (name, address) **VALUES** ('Luca', { "@type": "d", "street": "Melrose Avenue"})

• 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`.

**8.3. SQL - MATCH**

Queries the database in a declarative manner, using pattern matching.

**Simplified Syntax**

MATCH
{
    [type: <type>],
    [as: <alias>],
    [where: (<whereCondition>)]
}
.<functionName>()
    [type: <typeName>],
    [as: <alias>],
    [where: (<whereCondition>)],
    [while: (<whileCondition>)],
    [maxDepth: <number>],
    [depthAlias: <identifier>],
• <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,#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,#context-variables). For more information, see <<Deep Traversal While Condition,#deep-traversal), 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.

Examples

The following examples are based on this sample data-set from the type `People`:

![Table](image-url)
• 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: friendOffFriend}
    RETURN person, friendOffFriend

person | friendOffFriend
--------+----------------
#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: friendOffFriend, where: ($matched.person != $currentMatch)}
    RETURN person, friendOffFriend

person | friendOffFriend
- 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
```

<table>
<thead>
<tr>
<th>person</th>
<th>friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>#12:0</td>
</tr>
<tr>
<td>#12:0</td>
<td>#12:1</td>
</tr>
<tr>
<td>#12:0</td>
<td>#12:2</td>
</tr>
<tr>
<td>#12:0</td>
<td>#12:3</td>
</tr>
<tr>
<td>#12:0</td>
<td>#12:4</td>
</tr>
</tbody>
</table>

- Finding friends of friends to six degrees of separation, since a particular date:

```
ArcadeDB> MATCH {type: Person, as: person, where: (name = 'John')}.bothE('Friend').bothV()+().both('Friend').both('Friend'){$as: friend, where: (date < ?)}.$as: person}.both('Friend').both('Friend'){$as: friend, while: ($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
```

<table>
<thead>
<tr>
<th>person</th>
<th>friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>#12:1</td>
</tr>
</tbody>
</table>
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')}.both('Friend') {as: friend} RETURN friend
```

```
#12:0
```

```
#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
```

**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
```

```
#12:0
```

```
#12:2
```
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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$matched</td>
<td>Gives the current matched record. You must explicitly define the attributes for this record in order to access them. You can use this in the <code>where:</code> and <code>while:</code> conditions to refer to current partial matches or as part of the <code>RETURN</code> value.</td>
</tr>
<tr>
<td>$currentMatch</td>
<td>Gives the current complete node during the match.</td>
</tr>
<tr>
<td>$depth</td>
<td>Gives the traversal depth, following a single path item where a <code>while:</code> condition is defined.</td>
</tr>
</tbody>
</table>

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}
```
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 firendName, friend.surname as friendSurname

Incomplete Hierarchy

Consider building a database for a company that shows a hierarchy of departments within the company. For instance,
This loosely shows that, - Department 0 is the company itself, manager 0 (m0) is the CEO - e10 works at department 7, his manager is m3 - e12 works at department 9, this department has no direct manager, so e12's manager is 'm3 (the upper manager)

In this case, you would use the following query to find out who's the manager to a particular employee:

ArcadeDB> SELECT EXPAND(manager) FROM (MATCH {type:Employee, where: (name = ?)}.out('WorksAt').out('ParentDepartment') {while: (out('Manager').size() == 0), where: (out('Manager').size() > 0)}.out('Manager') {as: manager} RETURN 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")
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, \((a, \text{in this case})\), as the result of zero traversal.

To exclude the starting point, you need to add a `where:` condition, such as:

```graphql
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, \( \text{level} \mod 2 == 0 \).

To get this, you might use the following query:

```graphql
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.

**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')}.(both("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 particularly 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:

<table>
<thead>
<tr>
<th>person</th>
<th>friendship</th>
<th>friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>#13:0</td>
<td>#12:2</td>
</tr>
<tr>
<td>#12:0</td>
<td>#13:1</td>
<td>#12:3</td>
</tr>
<tr>
<td>#12:1</td>
<td>#13:2</td>
<td>#12:3</td>
</tr>
</tbody>
</table>

```
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:

<table>
<thead>
<tr>
<th>name</th>
<th>since</th>
<th>friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2015</td>
<td>Frank</td>
</tr>
<tr>
<td>John</td>
<td>2015</td>
<td>Jenny</td>
</tr>
<tr>
<td>Joe</td>
<td>2016</td>
<td>Jenny</td>
</tr>
</tbody>
</table>

```
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:

<table>
<thead>
<tr>
<th>person</th>
<th>friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>#12:2</td>
</tr>
<tr>
<td>#12:0</td>
<td>#12:3</td>
</tr>
<tr>
<td>#12:1</td>
<td>#12:3</td>
</tr>
</tbody>
</table>
```

#### $paths$, 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:

<table>
<thead>
<tr>
<th>person</th>
<th>friend</th>
<th>$ORIENT_DEFAULT_ALIAS_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>#12:2</td>
<td>#13:0</td>
</tr>
<tr>
<td>#12:0</td>
<td>#12:3</td>
<td>#13:1</td>
</tr>
<tr>
<td>#12:1</td>
<td>#12:3</td>
<td>#13:2</td>
</tr>
</tbody>
</table>
```

#### $elements$ 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:
<table>
<thead>
<tr>
<th>@rid</th>
<th>@type</th>
<th>name</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>Person</td>
<td>John</td>
<td>......</td>
</tr>
<tr>
<td>#12:1</td>
<td>Person</td>
<td>Joe</td>
<td>......</td>
</tr>
<tr>
<td>#12:2</td>
<td>Person</td>
<td>Frank</td>
<td>......</td>
</tr>
<tr>
<td>#12:3</td>
<td>Person</td>
<td>Jenny</td>
<td>......</td>
</tr>
</tbody>
</table>

$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:
<table>
<thead>
<tr>
<th>@rid</th>
<th>@type</th>
<th>name</th>
<th>since</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>#12:0</td>
<td>Person</td>
<td>John</td>
<td></td>
<td>......</td>
</tr>
<tr>
<td>#12:1</td>
<td>Person</td>
<td>Joe</td>
<td></td>
<td>......</td>
</tr>
<tr>
<td>#12:2</td>
<td>Person</td>
<td>Frank</td>
<td></td>
<td>......</td>
</tr>
<tr>
<td>#12:3</td>
<td>Person</td>
<td>Jenny</td>
<td></td>
<td>......</td>
</tr>
<tr>
<td>#13:0</td>
<td>Friend</td>
<td></td>
<td>2015</td>
<td>......</td>
</tr>
<tr>
<td>#13:1</td>
<td>Friend</td>
<td></td>
<td>2015</td>
<td>......</td>
</tr>
<tr>
<td>#13:2</td>
<td>Friend</td>
<td></td>
<td>2016</td>
<td>......</td>
</tr>
</tbody>
</table>

IMPORTANT: When using 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

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
```

**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.

**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

**8.4. SQL - SELECT**

ArcadeDB supports the SQL language to execute queries against the database engine. For more information, see << operators,SQL-Where.md#operators) and SQL-Where.md#functions). For more information on the differences between this implementation and the SQL-92 standard, please refer to <<this,SQL-Introduction section.

**Syntax:**

```
SELECT [ <Projections> ] [ FROM <Target> [ LET <Assignment>* ] ]
[ WHERE <Condition>* ]
[ GROUP BY <Field>* ]
```
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.

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,SQL-Query.md#projections), 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,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 ANY() LIKE '%danger%'

- Return any record where up to the third level of connections has some field that contains the word `danger`, ignoring case:

  ArcadeDB> SELECT FROM Profile WHERE ANY() TRAVERSE(0, 3) ( ANY().toUpperCase().indexOf('danger') > -1 )

- 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
• Traverse records from a root node:

ArcadeDB> SELECT FROM #11:4 WHERE ANY() TRAVERSE(0,10) (address.city = 'Rome')

• 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 '^[a-zA-Z]+$' AND
```

210
Using the **LET** makes the query shorter and faster, because it traverses the relationships only once:

```plaintext
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.

```plaintext
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,#projections)`. For instance, the query below displays the city name from the previous example:

```plaintext
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:

```plaintext
ArcadeDB> SELECT name, OUT("Friend").name AS friendName FROM Person

<table>
<thead>
<tr>
<th>name</th>
<th>friendName</th>
</tr>
</thead>
<tbody>
<tr>
<td>'John'</td>
<td>&lt;&lt;'Mark', 'Steve' ]</td>
</tr>
</tbody>
</table>
```

In the event if you want one record for each element in `friendName`, you can rewrite the query using **UNWIND**:

```plaintext
ArcadeDB> SELECT name, OUT("Friend").name AS friendName FROM Person UNWIND friendName
```
### Execution planning

For details about query execution planning, please refer to [SQL SELECT Execution](#).

### SQL SELECT Statements Execution

The execution flow of a SELECT statement is made of many steps. Understanding these steps will help you to write better and more optimized queries.

The SELECT query execution, at a very high level, is made of three steps: - Query optimization - Creation of execution plans - Choice of the optimal execution plan - Actual execution

#### Query optimization

The first step for the query executor is to run a query optimizer. This operation can change the internal structure of the SQL statement to make it more efficient, preserving the same semantics of the original query.

Typical optimization steps are:

- Early calculation of expressions

eg. consider the following statement

```
SELECT FROM Person WHERE fullName = "John" + " " + "Smith"
```

The result of the string concatenation "John" + " " + "Smith" does not depend on the query context (eg. the content of a record in the result set), so it can be calculated only once in the execution phase. The result of the optimization of this query will be the equivalent of

```
SELECT FROM Person WHERE fullName = "John Smith"
```

- Early calculation of sub-queries

eg. consider the following statement

```
SELECT FROM Person WHERE father in (SELECT FROM Person WHERE name = 'John')
```

The result of the subquery does not depend on the parent query context, so it can be executed only...
once, and then use the result as an argument for the parent query:

```
LET $a = (SELECT FROM Person WHERE name = 'John');
SELECT FROM Person WHERE father in $a
```

It is possible only if the subquery does not depend on the context of the parent query, so for example the following cannot be split:

```
SELECT FROM Person WHERE father in (SELECT FROM Person WHERE name = 'John' and surname = $parent.$current.surname)
```

- Refactoring of the WHERE conditions

eg. consider the following:

```
SELECT FROM Person
WHERE (name = 'John' AND surname = 'Smith')
OR (name = 'John' AND surname = 'Doe')
OR (name = 'John' AND surname = 'Travolta')
OR (name = 'John' AND surname = 'Lennon')
OR (name = 'John' AND surname = 'Nash')
```

If the WHERE condition is evaluated as is, the condition name = 'John' has to be evaluated five times for each record that does not have a 'John' as a name. This query can be rewritten as:

```
SELECT FROM Person
WHERE name = 'John' AND ( surname = 'Smith'
  OR surname = 'Doe'
  OR surname = 'Travolta'
  OR surname = 'Lennon'
  OR surname = 'Nash'
)
```

Sometimes, like in case of full type scan, this is convenient. In other cases it’s not. Eg. if Person type has an index on <name, surname>, the original query can be executed as the union of five index lookups. The query optimizer will create multiple versions of optimized conditions, for different execution plans (see below).

**Creation of execution plans**

An execution plan is a sequence of operations that the query engine has to execute to calculate the query result.
Each step in the execution plan typically does a single operation, eg. fetch data from a type, filter results, calculate projections and so on.

For the same query, ArcadeDB can calculate multiple execution plans, based on involvement of indexes, optimized sorting and so on.

An execution plan has an execution cost that depends on the number of processed records, the number of operations performed and the elaboration time. The query executor uses the execution cost as the main criterion to choose the optimal execution plan.

**Choice of the optimal execution plan**

If the query executor produces multiple execution plans, then it has to choose the more convenient one to actually execute the query. This choice is made based on the execution cost: the execution plan with the minimum cost is chosen.

**Actual execution**

After choosing the optimal execution plan, it is just executed.

The execution of an execution plan is just the execution of all the steps that it represents.

**Query Execution Plan**

As described above, an execution plan is a sequence of steps that have to be executed to calculate a query result.

Different queries will have different execution plans.

The typical execution plan is made of the following steps:

- fetch from query target (that can be a type, a bucket, an index and so on)
- evaluate LET expressions
- calculate query projections
- filter results
- aggregate data (eg. aggregate functions + GROUP BY)
- unwind projections
- sort result (ORDER BY)
- SKIP
- LIMIT

Obviously, a simple query like `SELECT FROM Person` will have a very simple execution plan made of a single step (the fetch from `Person` type), while a complex query will have an execution plan made of multiple steps.

To display the execution plan of a query, without executing it, you can just execute the query prefixing it with `EXPLAIN`, eg.
EXPLAIN SELECT FROM Person

Pagination

ArcadeDB supports pagination natively. Pagination doesn’t consume server side resources because no cursors are used. Only Record ID’s are used as pointers to the physical position in the bucket.

There are 2 ways to achieve pagination:

Use the SKIP-LIMIT

The first and simpler way to do pagination is to use the SKIP/LIMIT approach. This is the slower way because ArcadeDB repeats the query and just skips the first X records from the result. Syntax:

```
SELECT FROM <target> [WHERE ...] SKIP <records-to-skip> LIMIT <max-records>
```

Where: - records-to-skip is the number of records to skip before starting to collect them as the result set - max-records is the maximum number of records returned by the query

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.

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,TRAVERE versus SELECT) below.

**Syntax**

```
TRAVERSE [<type.]|field|]*|any()|all()
[FROM <target>]
[  MAXDEPTH <number>
  | WHILE <condition>
] [LIMIT <max-records>]
[STRATEGY <strategy>]
```

- `<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 <<=Traversal-Strategies,BREADTH_FIRST)
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 *, any() or all() 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 this, 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, typees 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) as the type $V$ and the Edge (or Arc, Connection, Link, etc.) as the type $E$. 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 `outE().inV()`.
- **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'
```

**SQL - TRUNCATE BUCKET**

Deletes all records of a bucket. This command operates at a lower level than the standard `DELETE` command.

Truncation is not permitted on vertex or edge types, but you can force its execution using the `UNSAFE` keyword. Forcing truncation is strongly discouraged, as it can leave the graph in an
inconsistent state.

**Syntax**

```
TRUNCATE BUCKET <bucket>
```

- `<bucket>` Defines the bucket to delete.
- `UNSAFE` Defines whether the command forces truncation on vertex or edge types, (that is, sub-types that extend the types \(V\) or \(E\)).

**Examples**

- Remove all records in the bucket `profile`:

  ```
  ArcadeDB> TRUNCATE BUCKET profile
  ```

For more information, see:

- `DELETE`
- `TRUNCATE TYPE`

---

**SQL - TRUNCATE TYPE**

Delete 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 command 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 typees, 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 truncation on vertex or edge types, (that is, sub-types that extend the types \(V\) or \(E\)).
Examples

• Remove all records of the type Profile:

ArcadeDB> TRUNCATE TYPE Profile

For more information, see:

• DELETE
• TRUNCATE BUCKET
• CREATE TYPE

8.5. SQL - UPDATE

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 <timeout>]
```

- **SET** Defines the fields to update.
- **REMOVE** Removes an item in collection and map fields.
- **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 `<<SQL-Where,`WHERE`>>` clause and a type target. There are further limitations on `UPSERT`, explained below.

- **`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 maximum number of records to update.
- `LIMIT` Defines the time you want to allow the update run before it times out.

The RID must have a # prefix. For instance, #12:3.

Examples:

- Update to change the value of a field:

  ```
  ArcadeDB> UPDATE Profile SET nick = 'Luca' WHERE nick IS NULL
  Updated 2 record(s) in 0.008000 sec(s).
  ```

- 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'
  ```

- 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
  ```
• Update to remove a value from a map

ArcadeDB> UPDATE Account REMOVE addresses = 'Luca'

• 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.

**SQL - PROFILE**

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 SQL-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
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.

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+KOVmcwNUq8Eq1w2/j8O
VnEkkVjzkeg=$2q2u4rjUL1jgoKBX9sG0rV0b0h6aHo+RhHs0kXneGkM=","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"
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:

```json
{
    "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.

### 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 \*.

---

227
**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

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:

```json
{
  "types": {
    "*": {
      "access": []
    },
    "Blog": {
      "access": [
        "readRecord"
      ]
    }
  }
}
```
The default settings for the admin group are:

```json
{
  "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.

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.
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.

OrientDB

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 “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. Much easier developing with ArcadeDB than with OrientDB with multi-threads applications. With OrientDB you have to use a pool of Database 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. Appendices

11.1. Data Types

ArcadeDB supports several data types natively. Below is the complete table.

<table>
<thead>
<tr>
<th>Type</th>
<th>SQL type</th>
<th>Description</th>
<th>Java type</th>
<th>Minimum - Maximum</th>
<th>Auto-conversion from/to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>BOOLEAN</td>
<td>Handles only the values True or False</td>
<td>java.lang.Boolean</td>
<td>0 - 1</td>
<td>String</td>
</tr>
<tr>
<td>Integer</td>
<td>INTEGER</td>
<td>32-bit signed Integers</td>
<td>java.lang.Integer</td>
<td>-2,147,483,648 - +2,147,483,647</td>
<td>Any Number, String</td>
</tr>
<tr>
<td>Short</td>
<td>SHORT</td>
<td>Small 16-bit signed integers</td>
<td>java.lang.Short or short</td>
<td>-32,768 - 32,767</td>
<td>Any Number, String</td>
</tr>
<tr>
<td>Long</td>
<td>LONG</td>
<td>Big 64-bit signed integers</td>
<td>java.lang.Long or long</td>
<td>-2&lt;sup&gt;63&lt;/sup&gt; - +2&lt;sup&gt;63&lt;/sup&gt;-1</td>
<td>Any Number, String</td>
</tr>
<tr>
<td>Float</td>
<td>FLOAT</td>
<td>Decimal numbers</td>
<td>java.lang.Float or float</td>
<td>2&lt;sup&gt;-149&lt;/sup&gt; - (2&lt;sup&gt;-127&lt;/sup&gt;)&lt;sup&gt;*2&lt;/sup&gt;</td>
<td>Any Number, String</td>
</tr>
<tr>
<td>Double</td>
<td>DOUBLE</td>
<td>Decimal numbers with high precision</td>
<td>java.lang.Double or double</td>
<td>2&lt;sup&gt;-1074&lt;/sup&gt; - (2&lt;sup&gt;-1023&lt;/sup&gt;)&lt;sup&gt;*2&lt;/sup&gt;</td>
<td>Any Number, String</td>
</tr>
<tr>
<td>Datetime</td>
<td>DATETIME</td>
<td>Any date with the precision up to milliseconds. To know more about it, look at Managing Dates</td>
<td>java.util.Date</td>
<td></td>
<td>Date, Long, String</td>
</tr>
</tbody>
</table>

233
<table>
<thead>
<tr>
<th>Type</th>
<th>SQL type</th>
<th>Description</th>
<th>Java type</th>
<th>Minimum - Maximum</th>
<th>Auto-conversion from/to</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>STRING</td>
<td>Any string as alphanumeric sequence of chars</td>
<td>java.lang.String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>BINARY</td>
<td>Can contain any value as byte array</td>
<td>byte[]</td>
<td>0 - 2,147,483,647</td>
<td>String</td>
</tr>
<tr>
<td>Embedded</td>
<td>EMBEDDED</td>
<td>The Record is contained inside the owner. The contained Record has no RIDs</td>
<td>EmbeddedDocument</td>
<td></td>
<td>EmbeddedDocument</td>
</tr>
<tr>
<td>Embedded list</td>
<td>LIST</td>
<td>The Records are contained inside the owner. The contained records have no RID and are reachable only by navigating the owner record</td>
<td>List&lt;EmbeddedDocument&gt;</td>
<td>0 - 41,000,000 items</td>
<td>String</td>
</tr>
<tr>
<td>Embedded map</td>
<td>MAP</td>
<td>The Records are contained inside the owner as values of the entries, while the keys can only be Strings. The contained ords e no RIDs and are reachable only by navigating the owner Record</td>
<td>Map&lt;String, EmbeddedDocument&gt;</td>
<td>0 - 41,000,000 items</td>
<td>Collection&lt;? extends EmbeddedDocument&lt;?&gt;&gt;, String</td>
</tr>
<tr>
<td>Type</td>
<td>SQL type</td>
<td>Description</td>
<td>Java type</td>
<td>Minimum - Maximum</td>
<td>Auto-conversion from/to</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>----------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Link</td>
<td>LINK</td>
<td>Link to another Record. It's a common one-to-one relationship</td>
<td>RID, <code>&lt;? extends Record&gt;</code></td>
<td>1:1 - 32767:2^63-1</td>
<td>String</td>
</tr>
<tr>
<td>Byte</td>
<td>BYTE</td>
<td>Single byte. Useful to store small 8-bit signed integers</td>
<td><code>java.lang.Byte</code> or <code>byte</code></td>
<td>-128 - +127</td>
<td>Any Number, String</td>
</tr>
<tr>
<td>Decimal</td>
<td>DECIMAL</td>
<td>Decimal numbers without rounding</td>
<td><code>java.math.BigDecimal</code></td>
<td></td>
<td>Any Number, String</td>
</tr>
</tbody>
</table>

### 11.2. Settings

ArcadeDB allows changing settings at JVM (server or embedded) and per database level.

<table>
<thead>
<tr>
<th>Server/Embedded (JVM) Level</th>
<th>Database Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 ...```  

**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:

```java
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):

```java
GlobalConfiguration.BUCKET_DEFAULT_PAGE_SIZE.setValue(1048576);
```

**11.2.1. Available settings (in alphabetic order):**

The table that follows contains all the available settings in ArcadeDB.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>asyncOperationsQueueImpl</td>
<td>Queue implementation to use between 'standard' and 'fast'. 'standard' consumes less CPU than the 'fast' implementation, but it could be slower with high loads</td>
<td>String</td>
<td>standard</td>
</tr>
<tr>
<td>asyncOperationsQueueSize</td>
<td>Size of the total asynchronous operation queues (it is divided by the number of parallel threads in the pool)</td>
<td>Integer</td>
<td>1024</td>
</tr>
<tr>
<td>asyncTxBatchSize</td>
<td>Maximum number of operations to commit in batch by async thread</td>
<td>Integer</td>
<td>10240</td>
</tr>
<tr>
<td>asyncWorkerThreads</td>
<td>Number of asynchronous worker threads. 0 (default) = available cores minus 1</td>
<td>Integer</td>
<td>15</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>---------------</td>
</tr>
<tr>
<td>bucketDefaultPageSize</td>
<td>Default page size in bytes for buckets. Default is 65536</td>
<td>Integer</td>
<td>65536</td>
</tr>
<tr>
<td>command.timeout</td>
<td>Default timeout for commands (in ms)</td>
<td>Long</td>
<td>0</td>
</tr>
<tr>
<td>commitLockTimeout</td>
<td>Timeout in ms to lock resources during commit</td>
<td>Long</td>
<td>5000</td>
</tr>
<tr>
<td>cypher.statementCache</td>
<td>Max number of entries in the cypher statement cache. Use 0 to disable. Caching statements speeds up execution of the same cypher queries</td>
<td>Integer</td>
<td>1000</td>
</tr>
<tr>
<td>dumpConfigAtStartup</td>
<td>Dumps the configuration at startup</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>dumpMetricsEvery</td>
<td>Dumps the metrics at startup, shutdown and every configurable amount of time (in seconds)</td>
<td>Long</td>
<td>0</td>
</tr>
<tr>
<td>flushOnlyAtClose</td>
<td>Never flushes pages on disk until the database closing</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>freePageRAM</td>
<td>Percentage (0-100) of memory to free when Page RAM is full</td>
<td>Integer</td>
<td>50</td>
</tr>
<tr>
<td>ha.clusterName</td>
<td>Cluster name. By default is ‘arcadedb’. Useful in case of multiple clusters in the same network</td>
<td>String</td>
<td>arcadedb</td>
</tr>
<tr>
<td>ha.enabled</td>
<td>True if HA is enabled for the current server</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>ha.k8s</td>
<td>The server is running inside Kubernetes</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>ha.k8sSuffix</td>
<td>When running inside Kubernetes use this suffix to reach the other servers. Example: arcadedb.default.svc.cluster.local</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>ha.quorum</td>
<td>Default quorum between ‘none’, 1, 2, 3, ‘majority’ and ‘all’ servers. Default is majority</td>
<td>String</td>
<td>MAJORITY</td>
</tr>
<tr>
<td>ha.quorumTimeout</td>
<td>Timeout waiting for the quorum</td>
<td>Long</td>
<td>10000</td>
</tr>
<tr>
<td>ha.replicationChunkMaxSize</td>
<td>Maximum channel chunk size for replicating messages between servers. Default is 16777216</td>
<td>Integer</td>
<td>16777216</td>
</tr>
<tr>
<td>ha.replicationFileSize</td>
<td>Maximum file size for replicating messages between servers. Default is 1GB</td>
<td>Long</td>
<td>1073741824</td>
</tr>
<tr>
<td>ha.replicationIncomingHost</td>
<td>TCP/IP host name used for incoming replication connections. By default is 0.0.0.0 (listens to all the configured network interfaces)</td>
<td>String</td>
<td>0.0.0.0</td>
</tr>
<tr>
<td>ha.replicationIncomingPorts</td>
<td>TCP/IP port number used for incoming replication connections</td>
<td>String</td>
<td>2424-2433</td>
</tr>
<tr>
<td>ha.replicationQueueSize</td>
<td>Queue size for replicating messages between servers</td>
<td>Integer</td>
<td>512</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>ha.serverList</td>
<td>Servers in the cluster as a list of &lt;hostname/ip-address:port&gt; items separated by comma. Example: localhost:2424,192.168.0.1:2424</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>indexCompactionMinPagesSchedule</td>
<td>Minimum number of mutable pages for an index to be schedule for automatic compaction. 0 = disabled</td>
<td>Integer</td>
<td>10</td>
</tr>
<tr>
<td>indexCompactionRAM</td>
<td>Maximum amount of RAM to use for index compaction, in MB</td>
<td>Long</td>
<td>300</td>
</tr>
<tr>
<td>initialPageCacheSize</td>
<td>Initial number of entries for page cache</td>
<td>Integer</td>
<td>65535</td>
</tr>
<tr>
<td>maxPageRAM</td>
<td>Maximum amount of pages (in MB) to keep in RAM</td>
<td>Long</td>
<td>4096</td>
</tr>
<tr>
<td>network.socketBufferSize</td>
<td>TCP/IP Socket buffer size, if 0 use the OS default</td>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>network.socketTimeout</td>
<td>TCP/IP Socket timeout (in ms)</td>
<td>Integer</td>
<td>30000</td>
</tr>
<tr>
<td>ssl.keyStore</td>
<td>Path where the SSL certificates are stored</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ssl.keyStorePass</td>
<td>Password to open the SSL key store</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ssl.trustStore</td>
<td>Path to the SSL trust store</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ssl.trustStorePass</td>
<td>Password to open the SSL trust store</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ssl.enabled</td>
<td>Use SSL for client connections</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>pageFlushQueue</td>
<td>Size of the asynchronous page flush queue</td>
<td>Integer</td>
<td>512</td>
</tr>
<tr>
<td>postgres.debug</td>
<td>Enables the printing of Postgres protocol to the console. Default is false</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>postgres.host</td>
<td>TCP/IP host name used for incoming connections for Postgres plugin. Default is '0.0.0.0'</td>
<td>String</td>
<td>0.0.0.0</td>
</tr>
<tr>
<td>postgres.port</td>
<td>TCP/IP port number used for incoming connections for Postgres plugin. Default is 5432</td>
<td>Integer</td>
<td>5432</td>
</tr>
<tr>
<td>profile</td>
<td>Specify the preferred profile among: default, high-performance, low-ram, low-cpu</td>
<td>String</td>
<td>default</td>
</tr>
<tr>
<td>queryMaxHeapElementsAllowedPerPageOp</td>
<td>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)</td>
<td>Long</td>
<td>500000</td>
</tr>
<tr>
<td>redis.host</td>
<td>TCP/IP host name used for incoming connections for Redis plugin. Default is '0.0.0.0'</td>
<td>String</td>
<td>0.0.0.0</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>---------------</td>
</tr>
<tr>
<td>redis.port</td>
<td>TCP/IP port number used for incoming connections for Redis plugin. Default is 6379</td>
<td>Integer</td>
<td>6379</td>
</tr>
<tr>
<td>server.databaseDirectory</td>
<td>Directory containing the database</td>
<td>String</td>
<td>${arcadedb.server.rootPath}/databases</td>
</tr>
<tr>
<td>server.databaseLoadAtStartup</td>
<td>Open all the available databases at server startup</td>
<td>Boolean</td>
<td>true</td>
</tr>
<tr>
<td>server.defaultDatabases</td>
<td>The default databases created when the server starts. The format is (&lt;database-name&gt;[(&lt;user-name&gt;:&lt;user-passwd&gt;[:&lt;user-group&gt;])][{import</td>
<td>restore:&lt;URL&gt;}][;]'. 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}</td>
<td>String</td>
</tr>
<tr>
<td>server.httpIncomingHost</td>
<td>TCP/IP host name used for incoming HTTP connections</td>
<td>String</td>
<td>0.0.0.0</td>
</tr>
<tr>
<td>server.httpIncomingPort</td>
<td>TCP/IP port number used for incoming HTTP connections. Specify a single port or a range &lt;from-to&gt;. Default is 2480-2489 to accept a range of ports in case they are occupied.</td>
<td>String</td>
<td>2480-2489</td>
</tr>
<tr>
<td>server.httpTxExpireTimeout</td>
<td>Timeout in seconds for a HTTP transaction to expire. This timeout is computed from the latest command against the transaction</td>
<td>Long</td>
<td>30</td>
</tr>
<tr>
<td>serverMetrics</td>
<td>True to enable metrics</td>
<td>Boolean</td>
<td>true</td>
</tr>
<tr>
<td>server.mode</td>
<td>Server mode between development, test and production</td>
<td>String</td>
<td>development</td>
</tr>
<tr>
<td>server.name</td>
<td>Server name</td>
<td>String</td>
<td>ArcadeDB_0</td>
</tr>
<tr>
<td>server.plugins</td>
<td>List of server plugins to install. The format to load a plugin is: &lt;pluginName&gt;:&lt;pluginFullClass&gt;</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>server.rootPassword</td>
<td>Password for root user to use at first startup of the server. Set this to avoid asking the password to the user</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>server.rootPath</td>
<td>Root path in the file system where the server is looking for files. By default is the current directory</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>server.securityAlgorithm</td>
<td>Default encryption algorithm used for passwords hashing</td>
<td>String</td>
<td>PBKDF2WithHmacSHA256</td>
</tr>
<tr>
<td>server.securitySaltCacheSize</td>
<td>Cache size of hashed salt passwords. The cache works as LRU. Use 0 to disable the cache</td>
<td>Integer</td>
<td>64</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>server.saltIterations</td>
<td>Number of iterations to generate the salt or user password. Changing this setting does not affect stored passwords</td>
<td>Integer</td>
<td>65536</td>
</tr>
<tr>
<td>server.eventBusQueueSize</td>
<td>Size of the queue used as a buffer for unserviced database change events.</td>
<td>Integer</td>
<td>1000</td>
</tr>
<tr>
<td>sqlStatementCache</td>
<td>Maximum number of parsed statements to keep in cache</td>
<td>Integer</td>
<td>300</td>
</tr>
<tr>
<td>test</td>
<td>Tells if it is running in test mode. This enables the calling of callbacks for testing purpose</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>txRetries</td>
<td>Number of retries in case of MVCC exception</td>
<td>Integer</td>
<td>3</td>
</tr>
<tr>
<td>txWAL</td>
<td>Uses the WAL</td>
<td>Boolean</td>
<td>true</td>
</tr>
<tr>
<td>txWalFlush</td>
<td>Flushes the WAL on disk at commit time. It can be 0 = no flush, 1 = flush without metadata and 2 = full flush (fsync)</td>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>typeDefaultBuckets</td>
<td>Default number of buckets to create per type</td>
<td>Integer</td>
<td>8</td>
</tr>
</tbody>
</table>

### 11.3. SQL Syntax

ArcadeDB Query Language is an SQL dialect.

This page lists all the details about its syntax.

**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

**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

```sql
/* 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**

*(draft)* In current version, type names are case **insensitive**, all the other identifiers are case sensitive.

**Reserved words**
In ArcadeDB SQL the following are reserved words

- AFTER
- AND
- AS
- ASC
- BATCH
- BEFORE
- BETWEEN
- BREADTH_FIRST
- BY
- BUCKET
- CONTAINS
- CONTAINSALL
- CONTAINSKEY
- CONTAINSTEXT
- CONTAINSVALUE
- CREATE
- DEFAULT
- DEFINED
- DELETE
- DEPTH_FIRST
- DESC
- DISTINCT
- EDGE
- FETCHPLAN
- FROM
- 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
```
• **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
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

```json
NULL
null
Null
nUll
...
```

### Numbers

ArcadeDB can store five different types of numbers - Integer: 32bit signed - Long: 64bit signed - Float: decimal 32bit signed - Double: decimal 64bit signed - BigDecimal: absolute precision

**Integers** are represented in SQL as plain numbers, eg. 123. If the number represented exceeds the Integer maximum size (see Java java.lang.Integer MAX_VALUE and MIN_VALUE), then it's automatically converted to a Long.

When an integer is saved to a schemaful property of another numerical type, it is automatically converted.

**Longs** are represented in SQL as numbers with L suffix, eg. 123L (L can be uppercase or lowercase). Plain numbers (without L prefix) that exceed the Integer range are also automatically converted to Long. If the number represented exceeds the Long maximum size (see Java java.lang.Long MAX_VALUE and MIN_VALUE), then the 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>) function, 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
the overflow follows Java rules.

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

**Collections**

ArcadeDB supports two types of collections: - **Lists**: ordered, allow duplicates - **Sets**: not ordered (?), no duplicates The SQL notation allows to create Lists with square bracket notation, eg.

```
[1, 3, 2, 2, 4]
```

A List can be converted to a Set using the `.asSet()` method:

```
[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 - `{ <expression> (, <expression>)* }`: a list, an ordered collection that allows duplicates, eg. `[{"a", "b", "c"}]` - `{ <expression>: <expression> (, <expression>: <expression>)* }`: the result is an ODocument, with `<field>:<value>` values, eg. `{"a":1, "b": 1+2+3, "c": foo.bar.size()}`). The key name is converted to String if it’s not. - `<expression> <modifier> ( <modifier> )*`: a chain of modifiers (see below) -
<json>: It is translated to an ODocument. Nested JSON is allowed and is translated to nested
ODocuments - <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" } } }` - a method invocation, eg. `foo.size()`.

Method invocations can be chained, eg. `foo.toLowerCase().substring(2, 4)` - a square bracket filter,
eg. `foo[1]` or `foo[name = 'John']`

**Square bracket filters**

Square brackets can be used to filter collections or maps.

`field[ ( <expression> | <range> | <condition> ) ]`

Based on what is between brackets, the square bracket filtering has different effects:

- <expression>: If the expression returns an Integer or Long value (i), the result of the square
  bracket filtering is the i-th element of the collection/map. If the result of the expression (K) is not
  a number, the filtering returns the value corresponding to the key K in the map field. If the field
  is not a collection/map, the square bracket filtering returns null. The result of this filtering is
  ALWAYS a single value.

- <range>: A range is something like M..N or M…N where M and N are integer/long numbers, eg.
  `fieldName[2..5]`. The result of range filtering is a collection that is a subset 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'`)

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

Math Operators

- `+` (plus): addition if both operands are numbers, string concatenation (with string conversion) if one of the operands is not a number. The order of calculation (and conversion) is from left to right, eg `'a' + 1 + 2 = 'a12', 1 + 2 + 'a' = '3a'`. 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 left 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
### Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>[ ] [&gt;&gt;]</td>
</tr>
<tr>
<td>bitwise AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive OR</td>
<td>^</td>
</tr>
<tr>
<td>bitwise inclusive OR</td>
<td></td>
</tr>
<tr>
<td>array concatenation</td>
<td></td>
</tr>
</tbody>
</table>

#### 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

- `%=(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]\n
\[[1, 2], [3, 4] \|\| [5, 6] = [[1, 2], [3, 4], [5, 6]]\n
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]\n
\(([1, 2] \|\| [2, 3]).asSet() = [1, 2, 3]\n
**Specific behavior of NULL**

Null value has no effect when applied to a `||` operation. eg.

\[[1, 2] \|\| null = [1, 2]\n
null || [1, 2] = [1, 2]\n
To add null values to a collection, you have to explicitly wrap them in another collection, eg.

\[[1, 2] \|\| [null] = [1, 2, null]\n
**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 collection, otherwise it returns FALSE. It's NOT the check of collection intersections, so `['a', 'b', 'c'] CONTAINS ['a', 'b']` will return FALSE, while `['a', 'b', 'c'] CONTAINS 'a'` will return TRUE.
- **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%'`'  
- **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. \( 5 \text{ BETWEEN } 1 \text{ AND } 10 \)
• **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'`

### 11.4. Storage Internals

**Edit this section**

#### 11.4.1. 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.

### 11.5. 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? 1. 2. 3.
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](https://github.com).

---

* 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.