ArcadeDB Manual

Version 2021.09.01


```
[[Introduction]]
== Introduction

[[waht_is_arcadedb]]
=== 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.

[discrete]
==== 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.

[discrete]
==== 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, Kubernates, Amazon AWS
(coming soon) and Microsoft Azure consoles (coming soon).

[discrete]
==== 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.
=== 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, Kubernates 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 Kubernates to have an automatic setup of servers in HA with a load
balancer upfront.

[discrete]
==== 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.

[discrete]
==== Client-Server

This is the classic way people use a DBMS, like with Relational Databases.
The ArcadeDB server exposes <<HTTP/JSON Protocol,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.

[discrete]
==== 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,High Availability>>.

== Server

=== Server

To start ArcadeDB as a server run the script `server.sh` under the `bin` directory of
ArcadeDB distribution

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

<ArcadeDB_0> Starting ArcadeDB Server... [ArcadeDBServer]
<ArcadeDB_0> - JMX Metrics Started... [ArcadeDBServer]
<ArcadeDB_0> - Starting HTTP Server (host=0.0.0.0 port=2480)... [HttpServer]
XNIO version 3.3.8.Final [xnio]
XNIO NIO Implementation Version 3.3.8.Final [nio]
<ArcadeDB_0> - HTTP Server started (host=0.0.0.0 port=2480) [HttpServer]
<ArcadeDB_0> ArcadeDB Server started (CPUs=8 MAXRAM=1.92GB) [ArcadeDBServer]
```

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

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 <<#_settings,`server.name`>>,
example:

```
./server.sh -Darcadedb.server.name=ArcadeDB_Europe_0
```

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

==== 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 <<#_settings,`server.defaultDatabases`>>.

```
./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,HTTP/JSON>>
- Any <<#_mongodb-protocol-wrapper,MongoDB Driver>>
- Any <<#_redis-protocol-wrapper,Redis Driver>>

==== Plugins

===== MongoDB Protocol Wrapper

ArcadeDB Server supports a subset of the https://mongodb.com[MongoDB] protocol, like
CRUD operations and queries.

To start the MongoDB plugin, enlist it in the <<#_settings,`server.plugins`>>
settings. To specify multiple plugins, use the comma `,` as separator. Example:

```
./server.sh
-Darcadedb.server.plugins=MongoDB:com.arcadedb.mongodbw.MongoDBWrapperPlugin
```

The Server output will contain this line:

```
2018-10-09 18:47:01:692 INFO  <ArcadeDB_0> - Plugin MongoDB started [ArcadeDBServer]
```

===== Redis Protocol Wrapper

ArcadeDB Server supports a subset of the https://redis.io[Redis] protocol, like CRUD
operations and queries.

To start the Redis plugin, enlist it in the <<#_settings,`server.plugins`>> settings.
To specify multiple plugins, use the comma `,` as separator. Example:

```
./server.sh -Darcadedb.server.plugins=Redis:com.arcadedb.redisw.RedisWrapperPlugin
```

The Server output will contain this line:

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

                                        ``

=== High Availability

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

To start ArcadeDB Server in High Availability (HA) mode, modify the default setting <<#_settings,`ha.enabled`>> to `true`. Example:

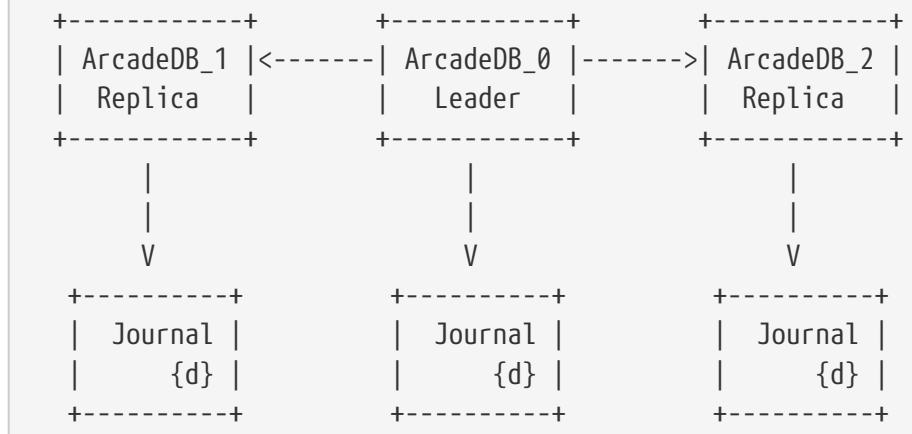
```
~/arcadedb $ cd bin
~/arcadedb/bin $ ./server.sh -Darcadedb.ha.enabled=true

<ArcadeDB_0> Starting ArcadeDB Server... [ArcadeDBServer]
<ArcadeDB_0> - JMX Metrics Started... [ArcadeDBServer]
<ArcadeDB_0> - Starting HTTP Server (host=0.0.0.0 port=2480)... [HttpServer]
XNIO version 3.3.8.Final [xnio]
XNIO NIO Implementation Version 3.3.8.Final [nio]
<ArcadeDB_0> - HTTP Server started (host=0.0.0.0 port=2480) [HttpServer]
<ArcadeDB_0> Listening Replication connections on 127.0.0.1:2424 (protocol v.-1)
[LeaderNetworkListener]
<ArcadeDB_0> Unable to find any Leader, start election (cluster=arcadedb
configuredServers=1 majorityOfVotes=1) [HAServer]
<ArcadeDB_0> Change election status from DONE to VOTING_FOR_ME [HAServer]
<ArcadeDB_0> ArcadeDB Server started (CPUs=8 MAXRAM=1.92GB) [ArcadeDBServer]
<ArcadeDB_0> Starting election of local server asking for votes from [] (turn=1
retry=0 lastReplicationMessage=-1 configuredServers=1 majorityOfVotes=1) [HAServer]
<ArcadeDB_0> Current server elected as new Leader (turn=1 totalVotes=1 majority=1)
[HAServer]
<ArcadeDB_0> Change election status from VOTING_FOR_ME to LEADER_WAITING_FOR_QUORUM
[HAServer]
<ArcadeDB_0> Contacting all the servers for the new leadership (turn=1)... [HAServer]
```

==== Architecture

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

[ditaa,ha-architecture]



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. In case a client is connected to a

Replica, the request is transparently forwarded to the Leader first to be replicated.

[ditaa,ha-architecture]
....
 +------------+
 | Client |
 +-----+------+
       |
       V
 +------------+   +------------+
 | ArcadeDB_1 |------->| ArcadeDB_0 |
 | Replica |forward | Leader |
 +------------+   +------------+
....

More coming soon.

==== Starting multiple servers in cluster

More coming soon.

==== Auto fail-over

More coming soon.

==== Auto balancing clients

More coming soon.

==== Kubernates (K8S)

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

```
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:

```
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.

=== Settings

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

```
$ java -Darcadedb.dumpConfigAtStartup=true ...
```

To change the same setting via Java code:

```java
GlobalConfiguration.findByKey("arcadedb.dumpConfigAtStartup").setValue(true);
```

Available Settings:

[%header,cols=4]
|===

|Name|Description|Type|Default Value
|dumpConfigAtStartup|Dumps the configuration at startup|Boolean|false
|dumpMetricsEvery|Dumps the metrics at startup, shutdown and every configurable amount of time (in ms)|Long|0
|test|Tells if it is running in test mode. This enables the calling of callbacks for testing purpose |Boolean|false
|maxPageRAM|Maximum amount of pages (in MB) to keep in RAM|Long|4
|initialPageCacheSize|Initial number of entries for page cache|Integer|65535
|flushOnlyAtClose|Never flushes pages on disk until the database closing|Boolean|false
|txWAL|Uses the WAL|Boolean|true
|txWalFlush|Flushes the WAL on disk at commit time. It can be 0 = no flush, 1 = flush without metadata and 2 = full flush (fsync)|Integer|0
|freePageRAM|Percentage (0-100) of memory to free when Page RAM is full|Integer|50
|asyncOperationsQueue|Size of the total asynchronous operation queues (it is divided by the number of parallel threads in the pool)|Integer|128
|asyncTxBatchSize|Maximum number of operations to commit in batch by async thread|Integer|10240
|pageFlushQueue|Size of the asynchronous page flush queue|Integer|128
|commitLockTimeout|Timeout in ms to lock resources during commit|Long|5000
|mvccRetries|Number of retries in case of MVCC exception|Integer|50
|sqlStatementCache|Maximum number of parsed statements to keep in cache|Integer|300
|indexCompactionRAM|Maximum amount of RAM to use for index compaction, in MB|Long|300
|indexCompactionMinPagesSchedule|Minimum number of mutable pages for an index to be schedule for automatic compaction. 0 = disabled|Integer|10
|network.socketBufferSize|TCP/IP Socket buffer size, if 0 use the OS default|Integer|0
|network.socketTimeout|TCP/IP Socket timeout (in ms)|Integer|30000
|ssl.enabled|Use SSL for client connections|Boolean|false
|ssl.keyStore|Use SSL for client connections|String|null
|ssl.keyStorePass|Use SSL for client connections|String|null
|ssl.trustStore|Use SSL for client connections|String|null
|ssl.trustStorePass|Use SSL for client connections|String|null
|server.name|Server name|String|ArcadeDB_0
|serverMetrics|True to enable metrics|Boolean|true

```
|server.rootPath|Root path in the file system where the server is looking for files.
By default is the current directory|String|.
|server.databaseDirectory|Directory containing the
database|String|${arcadedb.server.rootPath}/databases
|server.plugins|List of server plugins to install. The format to load a plugin is:
`<pluginName>:<pluginFullClass>`|String|
|server.defaultDatabases|The default databases created when the server starts. The
format is '(<database-name>[(<user-name>:<user-passwd>)[,]*])[;]*'. Pay attention on
using ';' to separate databases and ',' to separate credentials. Example:
'Universe[elon:musk];Amiga[Jay:Miner,Jack:Tramiel]'|String|
|server.httpIncomingHost|TCP/IP host name used for incoming HTTP
connections|String|0.0.0.0
|server.httpIncomingPort|TCP/IP port number used for incoming HTTP
connections|Integer|2480
|server.httpAutoIncrementPort|True to increment the TCP/IP port number used for
incoming HTTP in case the configured is not available|Boolean|true
|server.securityAlgorithm|Default encryption algorithm used for passwords
hashing|String|PBKDF2WithHmacSHA256
|server.securitySaltCacheSize|Cache size of hashed salt passwords. The cache works as
LRU. Use 0 to disable the cache|Integer|64
|server.saltIterations|Number of iterations to generate the salt or user password.
Changing this setting does not affect stored passwords|Integer|65536
|ha.enabled|True if HA is enabled for the current server|Boolean|false
|ha.quorum|Default quorum between 'none', 1, 2, 3, 'majority' and 'all' servers.
Default is majority|String|MAJORITY
|ha.quorumTimeout|Timeout waiting for the quorum|Long|10000
|ha.replicationQueueSize|Queue size for replicating messages between
servers|Integer|512
|ha.replicationFileMaxSize|Maximum file size for replicating messages between servers.
Default is 1GB|Long|1073741824
|ha.replicationIncomingHost|TCP/IP host name used for incoming replication
connections|String|localhost
|ha.replicationIncomingPorts|TCP/IP port number used for incoming replication
connections|String|2424-2433
|ha.clusterName|Cluster name. By default is 'arcadedb'. Useful in case of multiple
clusters in the same network|String|arcadedb
|ha.serverList|List of <hostname/ip-address:port> items separated by comma. Example:
localhost:2424,192.168.0.1:2424|String|

|===

== Tools

=== Console

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

```
~/arcadedb $ cd bin
~/arcadedb/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
```

==== Tutorial

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

```
> create database /temp/mydb

{mydb}>
```

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

```
> connect /temp/mydb

{mydb}>
```

Now let's create a "Profile" type:

```
{mydb}> create document type Profile

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

Check your new type is there:

```
{mydb}> info types

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

Finally, let's create a document of type "Profile":

```
{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:

```
{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.

=== Importer

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

- XML
- JSON
- CSV
- RDF

From file of types:

- Plain text
- Compressed with ZIP
- Compressed with GZip

Located on:

- local file system (just provide the path)
- and remote, by specifying `http` or `https`

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 $ cd bin
~/arcadedb/bin $ ./importer.sh -url http://commondatastorage.googleapis.com/freebase-public/rdf/freebase-rdf-latest.gz?

Analyzing url: http://commondatastorage.googleapis.com/freebase-public/rdf/freebase-rdf-latest.gz?... [SourceDiscovery]
Recognized format RDF (limitBytes=9.54MB limitEntries=0) [SourceDiscovery]
Creating type 'Node' of type VERTEX [Importer]
Creating type 'Relationship' of type EDGE [Importer]
Parsed 144951 (28990/sec) - 0 documents (0/sec) - 143055 vertices (28611/sec) - 144951 edges (28990/sec) [Importer]
Parsed 362000 (54256/sec) - 0 documents (0/sec) - 164118 vertices (5260/sec) - 362000 edges (54256/sec) [Importer]
...
```

If not specified, a database will be created under the "databases" directory, with name "imported". You can specify your own database (if existent) or the name of the new database must be created if not present:

Example of loading the Discogs dataset in the database on path "/temp/discogs":

```
~/arcadedb/bin $ ./importer.sh -database /temp/discogs -url https://discogs-data.s3-us-west-2.amazonaws.com/data/2018/discogs_20180901_releases.xml.gz
```

```
Note that in this case the URL is `https` and the file is compressed with `GZip`.

Example of importing New York Taxi dataset in CSV format. The first line of the CSV
file set the property names:

```
~/arcadedb/bin $ ./importer.sh -database /temp/nytaxi -url /personal/Downloads/data-
society-uber-pickups-in-nyc/original/uber-raw-data-april-15.csv/uber-raw-data-april-
15.csv
```


==== Configuration

- `url` as the URL to import. URLs can be local paths or from the Internet by using
`http` and `https`
- `database` as the database path/name to create (default=databases/imported)
- `forceDatabaseCreate` if the database doesn't exists it's created automatically
(default=false)
- `commitEvery` specifies the number of operations in a batch transaction. Higher is
better, but too high can consume too much RAM and increase the pressure of the JVM GC
(default=1,000)
- `parallel' specifies the number of parallel threads that execute the import.
(default=the available cores)
- `documentType` specifies the document type name to use during importing
(default=Document)
- `vertexType` specifies the vertex type name to use during importing (default=Node)
- `edgeType` specifies the edge type name to use during importing
(default=Relationship)
- `id` specifies the property that works as `id` (default=null)
- `idUnique` specifies if the property id is unique. (default=false)
- `idType` specifies the type of the property id. (default=String)
- `trimText` specifies if the imported text fields must be trimmed (removing leading
and tailing spaces). (default=true)
- `limitBytes` specifies the maximum bytes to read from the input source. (default=0
-> unlimited)
- `limitEntries` specifies the maximum number of lines to read from the input source.
(default=0 -> unlimited)

[[API]]

== API

=== Java API (Embedded)

NOTE: 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.

[discrete]
==== Synchronous API
```

```
The Synchronous API execute the operation immediately 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 need to work with multiple threads.

[discrete]
==== Asynchronous API

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.


==== 10-Minute Tutorial

In order to work with a database, the reference to the database to use must be taken. 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 `<<#_-code-databasefactory-code-class,DatabaseFactory>>` class that allows to create and open a database.

```java
DatabaseFactory arcade = new DatabaseFactory("/databases/mydb");
```

A `<<java-ref-database-factory.adoc#,DatabaseFactory>>` object doesn't keep any state and its only goal is creating a `<<java-ref-database.adoc#,Database>>` instance.

===== Create a new database

To create a new database from scratch, use the `.create()` method in `<<java-ref-database-factory.adoc#,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 `<<java-ref-database.adoc#,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:

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

Using `READ_ONLY` denys any changes to the database. This is the suggested method if
you're going to execute reads and queries only. 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:


```java
try( Database db = databaseFactory.open(); ){
  db.transaction(new Database.TransactionScope() {
    @Override
    public void execute(Database db) {
      // YOUR CODE
    }
  });
}
```

Or if you're using Java8+, you can simplify with a closure:

```java
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
````
````

```
    // 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 an control the transaction with explicit methods if you prefer. This code block is equivalent to the previous one:

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

  // YOUR CHANGES HERE

  db.commit();

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

Remember that every change in the database must be executed inside a transaction. ArcadeDB is a fully transactional DBMS, ACID compliant.
The usage of transactions is like with a Relational DBMS: `.begin()` starts a new transaction and `.commit()` commit all the changes in the database. In case you want to rollback the transaction, 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". In ArcadeDB it's mandatory to specify a type when you want tot create a document, a vertex or an edge.

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

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

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

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

Of course 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. 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 "IsFriend" 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:

[graphviz, dot-example, svg]
```
graph g {
    Elon -- Steve [label = "IsFriendOf" dir = "both"]
}
```

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

In the code snipped above, we have just created our first graph, made of 2 vertices and one edge that connects them. 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.

===== Traverse the Graph

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

You have basically three ways to do that (API, SQL, Apache GREMLIN) each one with its
pros/cons:

[cols=4]
|===
|
|API
|SQL
|Apache GREMLIN

|Speed|* * *|* *|*
|Flexibility|* * *|*|* *
|Embedded mode|Yes|Yes|No
|Remote mode|No|Yes|Yes (through the Gremlin Server plugin)
|===

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



====== Traverse via API

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

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

```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(LogN)`. Example:

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

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

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

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

Now that we have loaded the root vertex in memory, we're ready to do some traversal.
Before looking at the API, it's important to understand every edge has a direction: from vertex A to vertex B.
In the example above, the direction of the friendship is from "Elon" to "Steve".

```
While in most of the cases the direction is important, sometimes, like with the
friendship, it doesn't really matter the direction because if A is friend with B, it's
true also the opposite.

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

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

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

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

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

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


'''java
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    Vertex steve; // ALREADY LOADED VIA RID, KEYS OR SQL
    Iterable<Vertex> friends = steve.getVertices(DIRECTION.IN, new String[] {
"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, new String[] {
"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.x, you can run this query against the Apache Gremlin Server configured with ArcadeDB:

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

For more information about Apache Gremlin:

- http://tinkerpop.apache.org/gremlin.html[Introduction to Gremlin]
- http://tinkerpop.apache.org/docs/current/tutorials/getting-started/[Getting Started with Gremlin]
- http://tinkerpop.apache.org/docs/current/tutorials/the-gremlin-console/[The Gremlin Console]
- http://tinkerpop.apache.org/docs/current/recipes/[Gremlin Recipes]
- http://kelvinlawrence.net/book/Gremlin-Graph-Guide.html/[PRACTICAL GREMLIN: An Apache TinkerPop Tutorial]

==== 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 <<_getschema,`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);
```

==== 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();
```

==== 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.
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 https://en.wikipedia.org/wiki/Log-structured_merge-
tree[Log Structured Merge tree]
** `FULL_TEXT`, that uses https://lucene.apache.org/solr/guide/6_6/understanding-
analyzers-tokenizers-and-filters.html[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.

==== 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:

```java
TimeZone getTimeZone();
void setTimeZone(TimeZone timeZone);
String getDateFormat();
void setDateFormat(String dateFormat);
String getDateTimeFormat();
void setDateTimeFormat(String dateTimeFormat);
String getEncoding();
```
[discrete]
== Java Reference

==== `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 <<#_-
code-database-code-interface,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 te databases.
```

Syntax:

```java
void close()
```

====== exists()

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:

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

====== Database open(MODE mode)

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

Example:

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

==== `Database` Interface

It's the main class to operate with ArcadeDB. To obtain an instance of Database, use the class `<<#_-code-databasefactory-code-class,DatabaseFactory>>`.

===== Methods (Alphabetic order)

[cols=5]
|===
|<<_async,async()>>
|<<_begin,begin()>>
|<<_close,close()>>
|<<_commit,commit()>>
|<<_deleterecord-record,deleteRecord()>>
|<<_drop,drop()>>
|<<_getschema,getSchema()>>
|<<_isopen,isOpen()>>
|<<_iteratebucket-bucketname,iterateBucket()>>
|<<_iteratetype-classname-polymorphic,iterateType()>>
|<<_query-language_command-positionalparameters,query() positional parameters>>
|<<_query-language_command-parametermap,query() (parameter map)>>
|<<_command-language-command-positionalparameters,command() positional parameters>>
|<<_command-language-command-parametermap,command() (parameter map)>>
|<<_lookupbykey-type-properties-keys,lookupByKey()>>
|<<_lookupbyrid-rid-loadcontent,lookupByRID()>>
|<<_newdocument_typename,newDocument()>>
|<<_newedgebykeys-sourcevertextype-sourcevertexkey-sourcevertexvalue-
destinationvertextype-destinationvertexkey-destinationvertexvalue-
createvertexifnotexist-edgetype-bidirectional-properties,newEdgeByKeys()>>
|<<_newvertex-typename,newVertex()>>
|<<_rollback,rollback()>>
|<<_scanbucket_bucketname_callback,scanBucket()>>
|<<_scantype_classname_polymorphic_callback,scanType()>>
|<<_transaction_txblock,transaction() default>>
|<<_transaction_txblock_retries,transaction() with retries>>
|
|
|===

===== Methods (By category)

```
[%header,cols=5]
|===
|Transaction|Lifecycle|Query|Records|Misc

|<<_transaction-txblock,transaction() default>>
|<<_close,close()>>
|<<_query-language-command-positionalparameters,query() positional parameters>>
|<<_newdocument-typename,newDocument()>>
|<<_async,async()>>

|<<_transaction-txblock-retries,transaction() with retries>>
|<<_drop,drop()>>
|<<_query-language-command-parametermap,query() (parameter map)>>
|<<_newvertex-typename,newVertex()>>
|<<_command-language-command-positionalparameters,command() positional parameters>>

|<<_begin,begin()>>
|<<_isopen,isOpen()>>
|<<_lookupbykey-type-properties-keys,lookupByKey()>>
|<<_newedgebykeys-sourcevertextype-sourcevertexkey-sourcevertexvalue-
destinationvertextype-destinationvertexkey-destinationvertexvalue-
createvertexifnotexist-edgetype-bidirectional-properties,newEdgeByKeys()>>
|<<_command-language-command-parametermap,command() (parameter map)>>

|<<_commit,commit()>>
|
|<<_lookupbyrid-rid-loadcontent,lookupByRID()>>
|<<_deleterecord-record,deleteRecord()>>
|<<_getschema,getSchema()>>

|<<_rollback,rollback()>>
|
|<<_iteratetype-classname-polymorphic,iterateType()>>
|
|


|
|
|<<_iteratebucket-bucketname,iterateBucket()>>
|
|


|
|
|<<_scanbucket-bucketname-callback,scanBucket()>>
|
|


|
|
|<<_scantype-classname-polymorphic-callback,scanType()>>
```

```
|
|

|===

====== async()

It returns an instance of `<<#_-code-databaseasyncexecutor-code-
interface,DatabaseAsyncExecutor>>` to execute asynchronous calls.

Syntax:

```java
DatabaseAsyncExecutor async()
```

Example:

Execute an asynchronous query:

```java
db.async().query("sql", "select from V", null, null, new SQLCallback() {
  @Override
  public void onOk(ResultSet resultset) {
    while (resultset.hasNext()) {
      Result record = resultset.next();
      System.out.println( "Found record, name = " + record.getProperty("name"));
    }
  }

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

====== begin()

Starts a transaction on the current thread. Each thread can have only one active
transaction.
All the modification to the database become persistent only at pending changes in the
transaction are made persistent only when the `<<_commit,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:

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

====== isOpen()

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

Syntax:

```java
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:

```java
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:

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

Create a new record by passing position parameters:

```java
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:

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

Where:

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

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

Examples:

Create a new record by passing a map of parameters:

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

Syntax:

```java
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:

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

Examples:

```java
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_classname_polymorphic,iterateType()>> instead.
The result are not accumulated in RAM, but tather this method returns an
`Iterator<Record>` that fetches the records only when `.next()` is called.

Syntax:

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

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<>();

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.

```java
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_bucketname,iterateBucket()>>
method.
The result are not accumulated in RAM, but tather this method returns an
`Iterator<Record>` that fetches the records only when `.next()` is called.

Syntax:

```java
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":

```java
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:

```java
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 )
```

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,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,begin()>>` method.
````

Syntax:

```java
rollback()
```

Example:

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

====== scanBucket(bucketName, callback)

Scans all the records contained in a buckets. For each record found, the callback is called passing the current record.
To scan a type (with all its buckets), use the method <<_scantype_classname_polymorphic_callback,scanType()>> instead.
The callback method must return `true` to continue the scan, otherwise `false`.
Look also at the <<_iteratebucket_bucketname,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_bucketname_callback,scanBucket()>> method.
The callback method must return `true` to continue the scan, otherwise `false`.
Look also at the <<_iteratetype_classname_polymorphic,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 methods wraps a call to the method <<_transaction_txblock_retries,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:

```java
```

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

Examples:

Example by using Java8+ syntax:

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

Example by using Java7 syntax:

```java
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 `<<#_-code-database-code-interface,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

[cols=5]
|===
|<<_query-language-command-callback-positionalparameters,query() positional parameters>>
|<<_query-language-command-callback-parametermap,query() parameter map>>
|<<_command-language-command-callback-positionalparameters,command() positional parameters>>
|<<_command-language-command-callback-parametermap,command() parameter map>>

```
|
|===

====== 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
Resultset query( String language, String command, AsyncResultsetCallback callback,
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
- `callback`   is the callback to execute either if the query succeed
(method `onOk()` is called, or in case of error, where the method `onError()` is
called
- `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
db.async().query("sql", "select from V", new SQLCallback() {
  @Override
  public void onOk(ResultSet resultset) {
    while (resultset.hasNext()) {
      Result record = resultset.next();
      System.out.println( "Found record, name = " + record.getProperty("name"));
    }
  }

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

```

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, 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
Resultset query( String language, String command, AsyncResultsetCallback callback,
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>`
- `callback`   is the callback to execute either if the query succeed (method `onOk()` is called, or in case of error, where the method `onError()` is called
- `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, 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.
Only "SQL" language is supported, but in the future multiple languages could be used.

Syntax:

```java
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:

```java
db.async().command("sql", "insert into V set name = 'Jay', surname = 'Miner'", new
SQLCallback() {
  @Override
  public void onOk(ResultSet resultset) {
    System.out.println("Created new record: " + resultset.next() );
  }

  @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 SQLCallback()
{
  @Override
  public void onOk(ResultSet resultset) {
    System.out.println("Created new record: " + resultset.next() );
  }

  @Override
  public void onError(Exception exception) {
    System.err.println("Error on creating new record: " + exception );
  }
}, "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 the command to modify the database. Only "SQL" language is supported, but in the future multiple languages could be used.

Syntax:

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

Where:

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

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

Examples:

Create a new record by passing a map of parameters:

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

db.async().command("sql", "insert into V set name = :name, surname = :surname", new
SQLCallback() {
  @Override
  public void onOk(ResultSet resultset) {
```

```
    System.out.println("Created new record: " + resultset.next() );
  }

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

=== 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".

==== Tutorial

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

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

Now let's create the type "Class":

```
curl -X POST http://localhost:2480/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:

```
curl -X POST http://localhost:2480/command/school
     -d '{ "language": "sql", "command": "insert into Class set name = 'English',
location = "3rd floor"}'
     -H "Content-Type: application/json"
     --user root:root
```

Or by using the `document` API:

```
curl -X POST http://localhost:2480/document/school
     -d '{"@type": "Class", "name": "English", "location": "3rd floor"}'
     -H "Content-Type: application/json"
     --user root:root
```

==== Reference

===== Execute a command (POST)

Executes a non-idempotent command.

URL Syntax: `/command/{database}`

Where:

- `database` is the database name

Example to create the new document type "Class":

```
curl -X POST http://localhost:2480/command/school
     -d '{ "language": "sql", "command": "create document type Class"}'
     -H "Content-Type: application/json"
     --user root:root
```

===== Create a database (POST)

URL Syntax: `/create/{database}`

Where:

- `database` is the database name

Example to create a new database:

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

===== Create a document (POST)

URL Syntax: `/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":

```
curl -X POST http://localhost:2480/document/school
     -d '{"@type": "Person", "name": "Jay", "surname": "Miner", "age": 69}'
     -H "Content-Type: application/json"
     --user root:root
```

===== Load a document (GET)

URL Syntax: `/document/{database}/{rid}`

Where:

- `database` is the database name

Example of retrieving a document by RID:

```
curl -X GET http://localhost:2480/document/school/3:4
     --user root:root
```

The output will be:

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

===== Execute a query (GET)

This command allows to execute idempotent commands, like `SELECT` ad `MATCH`:

URL Syntax 1: `/query/{database}`

Where:

- `database` is the database name

The Payload is the JSON document to insert.

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

```
```

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

There is also this alternative syntax that takes the language and command in the URL:

URL Syntax 2: `/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

===== Drop a database (POST)

URL Syntax: `/drop/{database}`

Where:

- `database` is the database name

Example of deleting the database "school":

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

===== Get server information (GET)

Returns the current HA configuration.

URL Syntax: `/server`

Example:

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

Return:

```json
{ "leaderServer": "europe0", "replicaServers" : ["usa0", "usa1"]}
```

```
[[Comparison]]

== Comparison

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 getting 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.

==== Main similarities and differences

- Both can run on any platform
- Both can run SQL.
ArcadeDB and OrientDB both share the same SQL Engine
- ArcadeDB "types" are the "classes" in OrientDB
- Both ArcadeDB and OrientDB support inheritance
- ArcadeDB "buckets" are similar to the "clusters" in OrientDB
- ArcadeDB shares the same database instance across threads.
Much easier developing with ArcadeDB than with OrientDB with multi-threads
applications
- ArcadeDB uses thread locals only to manage transactions, while OrientDB makes a
strong usage of TL internally, making hard to pass the database instance across
threads and a pool is needed
- There is no base V and E classes in ArcadeDB, but vertex and edge are first type
citizens types of records.
Use `CREATE VERTEX TYPE Product` vs OrientDB `CREATE CLASS Product EXTENDS V`
- ArcadeDB saves every type and property name in the dictionary to compress the record
by storing only the names ids (varint)
- ArcadeDB keeps the MVCC counter on the page rather than on the record
- ArcadeDB manages everything as files and pages, for transactions and replication.
OrientDB has a mixed pages/record approach
- ArcadeDB allows custom page size per bucket/index
- ArcadeDB doesn't break record across pages, but rather create a placeholder pointing
to the page that contains the record.
This allows the RID to be immutable without the complexity of managing split records.
On the contrary, it is not possible to have objects larger than a page, so initial
setting of the page size is fundamental with ArcadeDB
- ArcadeDB supports light-weight edges, but they must be used with a different syntax.
This avoids automatic upgrade of edges and unexpected behavior
- ArcadeDB supports replication by using a Leader/Replica model without sharding for
now.
Instead, OrientDB is based on a Multi-Master model (the sharding was experimental,
```

```
never production ready) with a paxos style protocol not efficient on large volume of
transactions
- ArcadeDB replicates the pages across servers, so all the databases are identical at
binary level
- ArcadeDB Server supports HTTP/JSON, Postgres, MongoDB and Redis protocols, while
OrientDB supports only HTTP/JSON and a proprietary binary protocol

==== What ArcadeDB does not support

- ArcadeDB does not support storing records larger than the page size.
Initial setting of the page size is fundamental with ArcadeDB
- 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

==== 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!
- ArcadeDB uses much less RAM.
With the right tuning over the settings, it's able to work with only 4MB of JVM heap
- ArcadeDB codebase is much smaller and easier to maintain and improve
- ArcadeDB is lightweight, the engine is less than 1MB
- 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
```

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

If you would like to report an issue in the documentaton 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.