RCADE

PLAY with DATA

ArcadeDB Manual

Version 0.8

Table of Contents

Chapter 1. Introduction

1.1. 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. ArcadeDB is the fastest DBMS in all the benchmarks we have measured against RocksDB, MongoDB, Cassandra, OrientDB and Neo4j*.

1.1.1. How can it be so fast?

ArcadeDB is written in LLJ ("Low-Level-Java"), that means it's written in Java (Java7+), but without using high-level API. The result is that ArcadeDB does not use the Heap (and therefore the Garbage Collection), but still runs on pretty much every sw/hw configuration. Furthermore the kernel is built to be efficient on multi-core CPUs by using novel Mechanical Symphaty techniques.

1.1.2. Cloud DBMS

ArcadeDB was born on the cloud. Even though you can run ArcadeDB as embedded and in an on-premise setup, if your application is on the Cloud, you can spin an ArcadeDB cluster in a few seconds from the online dashboard.

1.1.3. Is ArcadeDB FREE?

ArcadeDB Community Edition is FREE for any usage, while the Enterprise Edition is released under a Commercial license.

1.2. Run ArcadeDB

You can run ArcadeDB in the following ways: - On the cloud, we support Amazon AWS, Microsoft Azure and Google Cloud Engine - On-premise, on your servers, any OS is good. - Embedded, if you develop with a language that runs on the (Java* Virtual Machine) JVM*

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.

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.

Chapter 2. API

2.1. Java API

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.

2.1.1. 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.

2.1.2. 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.

2.1.3. 10 Minutes 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>.asynch()` object.

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

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

A `DatabaseFactory` object doesn't keep any state and its only goal is creating a `Database` instance.

Create a new database

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

Syntax:

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

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

Open an existent database

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

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

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

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

Using `READ_ONLY` denys any changes to the database. This is the suggested method if you're going to execute reads and queries only. 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(new Database.TransactionScope() {
    @Override
    public void execute(Database db) {
      // YOUR CODE
    }
  });
}
```

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

```
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:

```
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:

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

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

```
try( Database db = databaseFactory.open(); ){
  db.transaction( () -> {
    // CREATE A CUSTOMER INSTANCE
    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:

```
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:

```
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:

```
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":

```
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:

	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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Traverse via SQL

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

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

Or with the more powerful MATCH statement:

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

Traverse via Apache GREMLIN

Since ArcadeDB is 100% compliant with Gremlin 3.x, you can run this query against an Apache Gremlin Server:

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

For more information about Apache Gremlin:

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

Reference

2.1.4. `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:

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

close()

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

Syntax:

```
void close()
```

exists()

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

Syntax:

```
boolean exists()
```

Database create()

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

Example:

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

Database open()

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

Example:

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

Database open(MODE mode)

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

Example:

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

2.1.5. `Database` Interface

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

Methods (Alphabetic order)

asynch()	begin()	close()	commit()	deleteRecord()
drop()	getSchema()	isOpen()	iterateBucket()	iterateType()
query() positional parameters	query() (parameter map)	command() positional parameters	command() (parameter map)	lookupByKey()
lookupByRID()	newDocument()	newEdgeByKeys()	newVertex()	rollback()
scanBucket()	scanType()	transaction() default	transaction() with retries	

Methods (By category)

Transaction	Lifecycle	Query	Records	Misc
transaction() default	close()	query() positional parameters	newDocument()	asynch()
transaction() with retries	drop()	query() (parameter map)	newVertex()	command() positional parameters

Transaction	Lifecycle	Query	Records	Misc
begin()	isOpen()	lookupByKey()	newEdgeByKeys()	command() (parameter map)
commit()		lookupByRID()	deleteRecord()	getSchema()
rollback()		iterateType()		
		iterateBucket()		
		scanBucket()		
		scanType()		

asynch()

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

Syntax:

```
DatabaseAsyncExecutor asynch()
```

Example:

Execute an asynchronous query:

```
db.asynch().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()` method is called. ArcadeDB supports ACID transactions. Before the commit, no other thread/client can see any of the changes contained in the current transaction.

Syntax:

```
begin()
```

Example:

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

close()

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

Syntax:

```
void close()
```

Example:

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

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

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

drop()

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

Syntax:

```
void drop()
```

Example:

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

getSchema()

Returns the Schema instance for the database.

Syntax:

```
Schema getSchema()
```

Example:

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

isOpen()

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

Syntax:

```
boolean isOpen()
```

Example:

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

query(language, command, positionalParameters)

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

Syntax:

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

Where:

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

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

Examples:

Simple query:

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

Query passing positional parameters:

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

query(language, command, parameterMap)

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

Syntax:

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

Where:

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

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

Examples:

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

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

command(language, command, positionalParameters)

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

Syntax:

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

Where:

- `language` is the language to use. Only "SQL" is supported
- `command` is the command to execute. If the language supports prepared statements (SQL does), you can specify parameters by using `?` for positional replacement or by name by using `:<arg-name>`
- `positionalParameters` optional variable array of parameters to execute with the query

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

Examples:

Create a new record:

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

Create a new record by passing position parameters:

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

command(language, command, parameterMap)

Executes a command that could change the database. This is the equivalent to `query()`, but allows

the command to modify the database. Only "SQL" language is supported, but in the future multiple languages could be used.

Syntax:

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

Where:

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

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

Examples:

Create a new record by passing a map of parameters:

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

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

commit()

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

Syntax:

```
commit()
```

Example:

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

deleteRecord(record)

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

Syntax:

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

Examples:

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

iterateBucket(bucketName)

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

Syntax:

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

Example:

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

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

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

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

Example:

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

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

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

iterateType(className, polymorphic)

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

Syntax:

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

Example:

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

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

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

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

lookupByKey(type, properties, keys)

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

Syntax:

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

Where:

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

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

Examples:

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

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

lookupByRID(rid, loadContent)

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

Syntax:

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

Where:

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

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

Examples:

Load the vertex by RID and its content:

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

newDocument(typeName)

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

Syntax:

```
MutableDocument newDocument( typeName )
```

Where:

- `typeName` type name

It returns a `MutableDocument` instance.

Examples:

Create a new document of type "Customer":

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

newVertex(typeName)

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

Syntax:

```
MutableVertex newVertex( typeName )
```

Where:

- `typeName` type name

It returns a `MutableVertex` instance.

Examples:

Create a new document of type "Customer":

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

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

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

Syntax:

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

Where:

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

It returns a `MutableEdge` instance.

Examples:

Create a new document of type "Customer":

```
Edge likes = db.newEdgeByKeys( "Account", new String[] {"id"}, new Object[] {322323},
                               "Song", new String[] {"title"}, new Object[] {"Chasing
Cars"},
                               false, "Likes", true);
likes.save();
```

rollback()

Aborts the thread's active transaction by rolling back all the pending changes. Usually the transaction rollback is executed in case of errors. If an exception happens during the call `commit()`, the transaction is roll backed automatically. Once rolled backed, the transaction cannot be committed anymore but it has to be re-started by calling the `begin()` method.

Syntax:

```
rollback()
```

Example:

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

scanBucket(bucketName, callback)

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

Syntax:

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

Example:

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

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

scanType(className, polymorphic, callback)

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

Syntax:

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

Example:

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

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

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

  return true;
});
```

transaction(txBlock)

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

transaction(txBlock, retries)

Executes a transaction block as a callback or a clojure. Before calling the callback in `TransactionScope`, the transaction is begun and after the end of the callback, the transaction is

committed. In case of any exceptions, the transaction is rolled back. In case a `NeedRetryException` exceptions is thrown, the transaction is repeated up to `retries` times

Syntax:

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

Examples:

Example by using Java8+ syntax:

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

Example by using Java7 syntax:

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

2.1.6. `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

query() positional parameters	query() parameter map	command() positional parameters	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:

```
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:

```
db.asynch().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:

```
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:

```
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:

```
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:

```
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.asynch().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:

```
db.asynch().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:

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

Where:

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

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

Examples:

Create a new record by passing a map of parameters:

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

db.asynch().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);
```

Chapter 3. Tools

3.1. 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 `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.

3.1.1. 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`.

```
./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
- Any MongoDB Driver
- Any Redis Driver

3.1.2. Plugins

MongoDB Protocol Wrapper

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:

```
./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 Redis protocol, like CRUD operations and queries.

To start the Redis plugin, enlist it in the `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]
```

3.2. 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) 2018 Arcade Analytics
(https://arcadeanalytics.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
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 type Profile

+-----------+--------+
|operation  |typeName|
+-----------+--------+
|create 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.

3.3. 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
```

3.3.1. 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)

3.4. HTTP/JSON Protocol

3.4.1. Execute a command (POST)

Syntax: `/command/{database}`

3.4.2. Create a database (POST)

Syntax: `/create/{database}`

3.4.3. Create a document (POST)

Syntax: `/document/{database}`

3.4.4. Load a document (GET)

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

3.4.5. Drop a database (POST)

Syntax: `/drop/{database}`

3.4.6. Execute a query (GET)

Syntax 1: `/query/{database}/{language}/{command}` Syntax 2: `/query/{database}`

3.4.7. Get server information (GET)

Syntax: `/server`

Chapter 4. Settings

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

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

To change the same setting via Java code:

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

Available Settings:

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

Name	Description	Type	Default Value
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

Name	Description	Type	Default Value
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

Name	Description	Type	Default Value
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	

Chapter 5. 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.

5.1. OrientDB

- ArcadeDB "types" are the "classes" in OrientDB
- 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 db 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
- 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
- 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 has the record. This allows the RID to be immutable without the complexity of managing split records
- ArcadeDB has a Leader/Replica replication model, no sharding. Instead OrientDB has a Multi-Master + sharding. For this reason, the ArcadeDB complexity is 100X less than OrientDB with some limitations that in practice are even less
- ArcadeDB replicates the pages across servers, so all the databases are identical at binary level

5.1.1. What Arcade does not support

- ArcadeDB doesn't support storing records with a size major than the page size. You can always create a bucket with a larger page size, but this can be done only at creation time
- ArcadeDB remote server supports only HTTP/JSON, no binary protocol is available
- ArcadeDB doesn't provide a dirty manager, so it's up to the developer to mark the object to save by calling `.save()` method. This makes the code of ArcadeDB smaller without handling edge cases

5.1.2. What Arcade has more than OrientDB

- ArcadeDB saves every type and property name in the dictionary to compress the record by storing only the names ids
- ArcadeDB asynchronous API automatically balance the load on the available cores

- ArcadeDB is much Faster, on single server it's easy to see 10X-20X improvement in performance, with 3 nodes it's about 50X-200X. With 10 servers it's >500X!
- ArcadeDB uses much less RAM. With the right tuning with settings, it's able to work with only 4MB of JVM heap
- ArcadeDB allows to execute operation in asynchronously way (by using `.async()`)
- ArcadeDB is lightweight, the engine is <200Kb

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