ULB UNIVERSITÉ LIBRE DE BRUXELLES

OrientDB – The Multi-Model and Graph Database

Info-H-415: Advanced Databases

Alex Buleon - 000460570

Erasmus Mundus BDMA

Table of content

Introduction to NoSQL, graph databases and OrientDB

Introduction

OrientDB is a multi-model database. It is based on a NoSQL engine compatible with graph databases and document databases, borrowing features from object databases as well. Hence, in order to understand how it works in details, we shall understand what is NoSQL and what are the different existing models today.

Overview of NoSQL and its models

NoSQL refers to a family of database management systems (DBMS) that deviates from the traditional relational database paradigm. The "No" in NoSQL either stands for the negation of the traditional relational database system, or, and that is more the case here, for "not only".

Even if the exact definition remains open to debate, the term itself refers to multiple DBMS that emerged around 2010 to solve the problem of traditional databases scalability and flexibility. The main characteristic of a NoSQL DBMS is generally a horizontal salability which allows the management of big volumes of data.

Here is a glimpse to the most common NoSQL models:

The **column-oriented databases** are closer to the traditional approach, and their data is stored in several columns instead of rows. It allows to quickly add columns to the tables as the rows don't need to be resized. It also makes the aggregations between tables easier.

Product	
ID	Value
1	Beer
2	Beer
3	Vodka
4	Whiskey
5	Whiskey
6	Vodka
7	Vodka

Customer	
ID	Customer
1	Thomas
2	Thomas
3	Thomas
4	Christian
5	Christian
6	Alexei
7	Alexei

Date	
ID	Date
1	2011-11-25
2	2011-11-25
3	2011-11-25
4	2011-11-25
5	2011-11-25
6	2011-11-25
7	2011-11-25

Sale	
ID	Sale
1	2 GBP
2	2 GBP
3	10 GBP
4	5 GBP
5	5 GBP
6	10 GBP
7	10 GBP

Figure 1 Example of a column-oriented database

The **key/value databases** are following a paradigm designed with two entities: a dictionary (or a hash), which can contain a collection of objects and records containing the data, and the key, a unique identifier used to quickly get the associated data.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Figure 2 Example of a key/value database

Document databases are organized the same way as the key/value method. The value here is usually a collection of fields, themselves containing the records, as represented in Figure 3 bellow. Document databases have the particularity to be schema-less, which can be very useful in some situations. For instance, when using an external API with a relational database, there is a risk to crash the system if the API changes somehow in its structure. However, a document database management system will still correctly work, because it only finds its information with the same key, regardless of the other changes in the document.

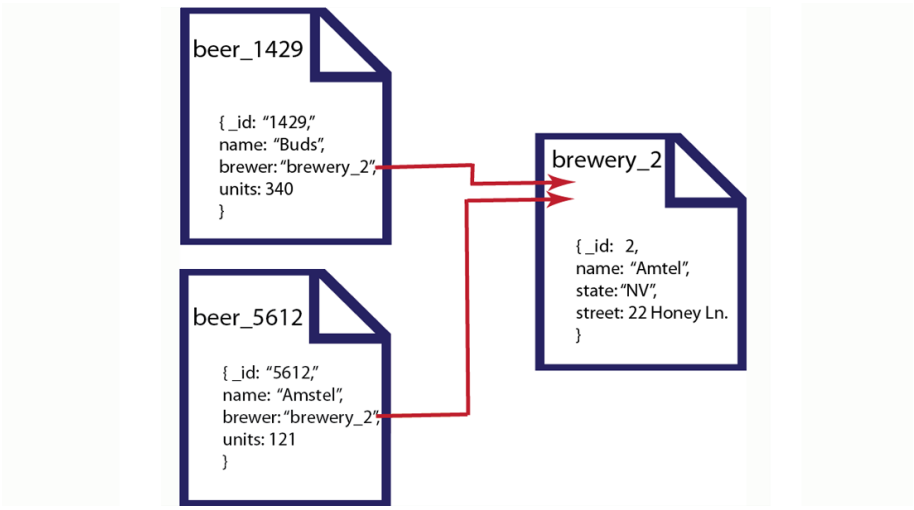

Figure 3 Example of a document database

Finally, the **graph databases** are similar to document databases, but with the addition of a relation system to navigate between documents. The model revolves around two key components: vertices and edges. A vertex is a node, where data is stored in the manner of a document, and the edges are the links representing concrete relation between vertices.

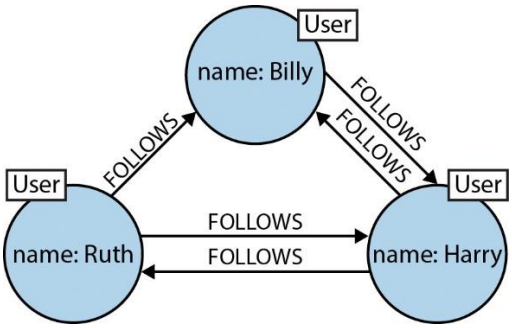

Figure 4 Graph database application with Twitter from the official Neo4J website

As it can be seen in figure 4, relations are usually direct actions between nodes, which result in simpler queries than more traditional databases, without the need to join tables.

However, once again the exact definition is vague[1]. Some say that the "graph database" nomenclature should only be given to the system which support the index-free adjacency implementation, where others think that this is more of a commercial move.

About OrientDB

OrientDB was released in 2010 as an open-source project. It means it can be used freely for any purpose as a community edition. The developers also provide an enterprise edition, which adds some synchronization tools to stay on par with an existing relational database, and some extensive support methods.

OrientDB respects the ACID model entirely that ensure that a transaction is executed reliably. The **atomicity** property guarantee that the database support complete transactions, **consistency** means that any transaction will never let the database in a corrupted state, **isolation** ensure that all the transactions are independent, so concurrent transactions execute as if they were in series, and finally the **durability** principle shows that any confirmed transaction has to stay recorded even if the system is unexpectedly shut down. Following the ACID model, although not mandatory, is a great proof of stability and this is also one of the features that helps giving credibility to the DBMS to be chose by an enterprise for a project. And OrientDB indeed manage to be used by several popular companies, like Dell, Cisco, Comcast, Lufthansa, Thales and Fox Sports among others.

Application databases

For the purpose of this project, I chose to work on a public database of Arda, the world of Tolkien and the Lord of the Rings. This database is used on a (relatively) popular website called Arda maps[2] which is an open-source and community driven project. It gathers most of characters (882), places (796) and event (58) for the three different ages.

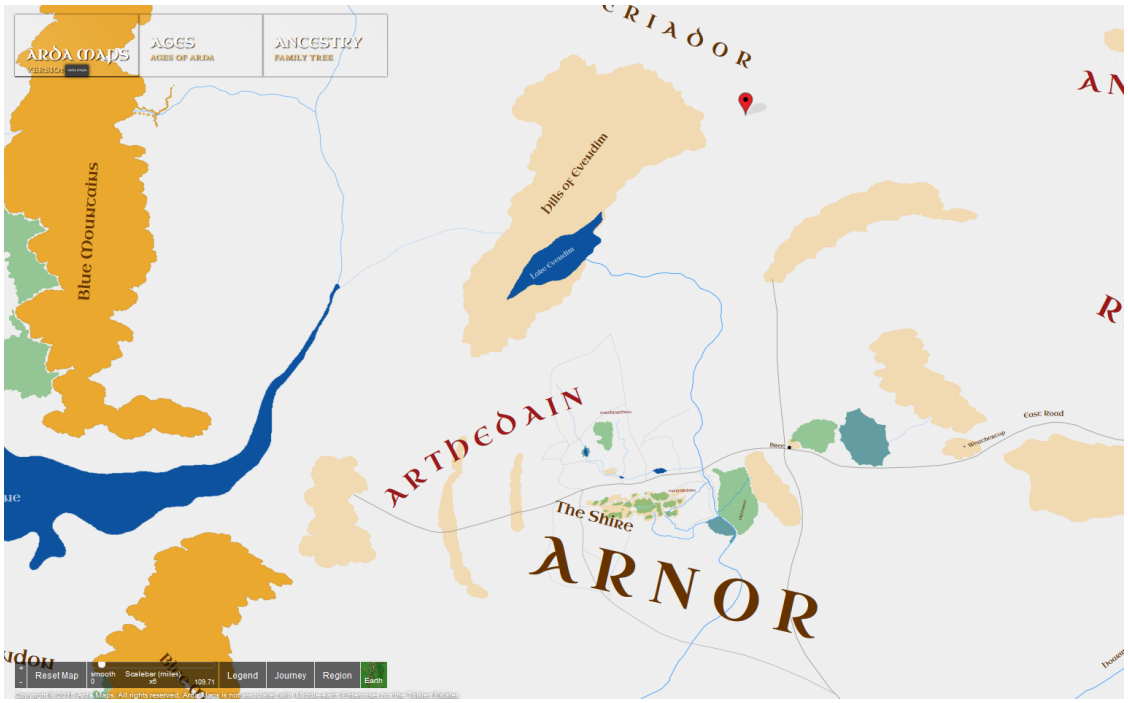

Figure 5 Use of the Tolkien-Arda database on the website arda-maps.org

I liked the fact that the relations were quite explicit and thus easier to understand (i.e. create family trees, see which character was at which event or location), however because of the small size of the database, I also worked, although to a lesser extent, with the Reactome pathways database[3]. Reactome is another open-source project, this time around biological pathways and human/organism processes. As opposed to the Tolkien-Arda database, Reactome has a total of 1,483,399 vertices and 7,819,353 edges. However, in my case, I didn't used it much because the dataset is less familiar and intuitive for me, and for the few queries that would return a much bigger dataset in result, some heavy performance issues were occurring (as the graph editor engine process every nodes and relations).

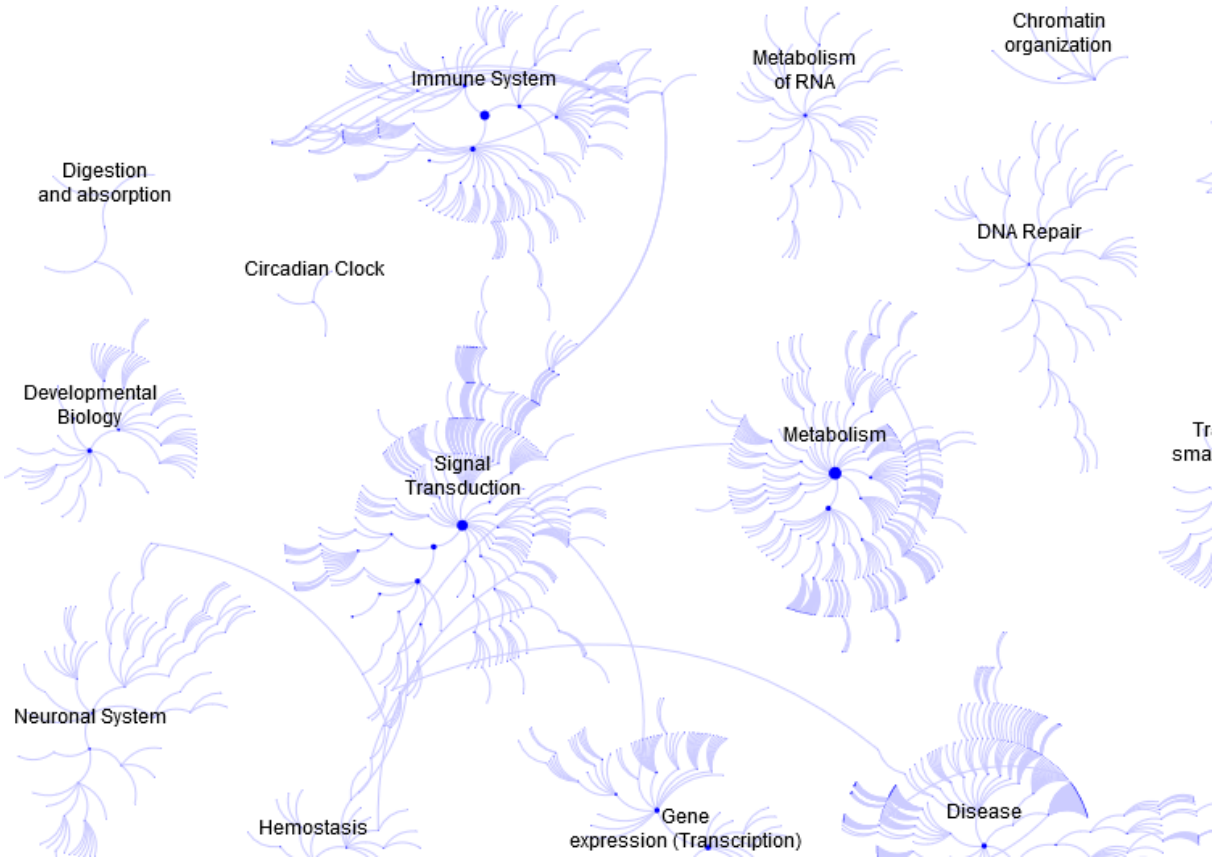

Figure 6 Graph of the Reactome database from the Pathway Browser of reactome.org

Installation and presentation

Setup

One of the selling point of OrientDB is its ease of setup and ease of use. Indeed, the process is straight-forward. The first thing we need to do is to download the compressed package from orientdb.com and extract it where we want.

Nom	Modifié le	Type	Taille
benchmarks	13/09/2017 15:58	Dossier de fichiers	
bin	13/09/2017 15:58	Dossier de fichiers	
config	13/09/2017 15:37	Dossier de fichiers	
databases	17/12/2017 14:44	Dossier de fichiers	
lib	13/09/2017 15:58	Dossier de fichiers	
log	17/12/2017 14:33	Dossier de fichiers	
plugins	13/09/2017 15:58	Dossier de fichiers	
www	13/09/2017 15:37	Dossier de fichiers	
history.txt	13/09/2017 15:37	Fichier TXT	60 Ko
license.txt	13/09/2017 15:37	Fichier TXT	12 Ko
readme.txt	13/09/2017 15:37	Fichier TXT	4 Ko

Figure 7 Community zip file content

We then have to start the local server, which can be done by executing server.bat or server.sh (respectively on Windows or on Linux) in the folder bin.

Figure 8 OrientDB console logo when starting the server

Once the local server successfully launched, we just have to go on the localhost web application. There, the login page allows a few options (see figure 9). You can either re-log to one of your projects, create a new one, download a public database, import one with one of the provided tools or drop the selected database.

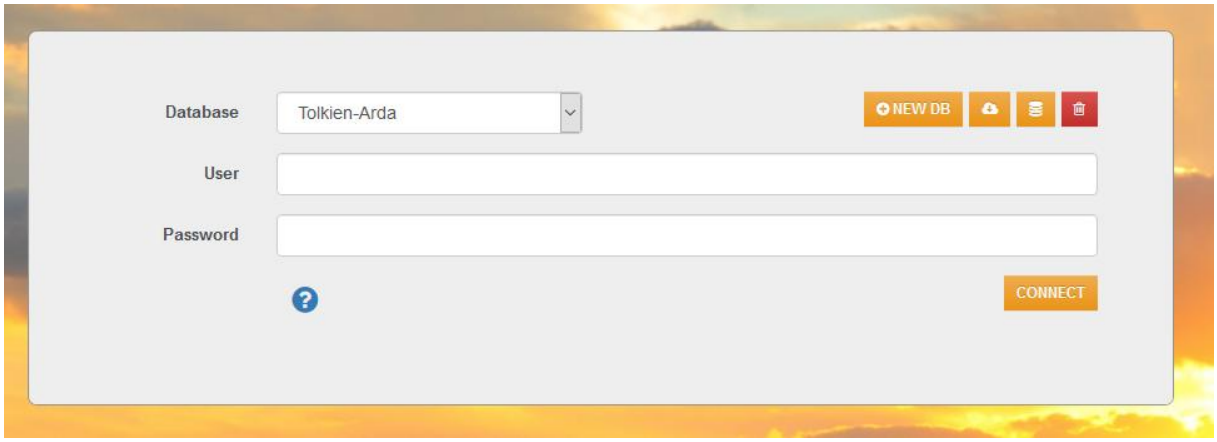

Figure 9 Login page

Web application

After connecting to a database, we arrive on the main application, divided in six tabs (see figure 10).

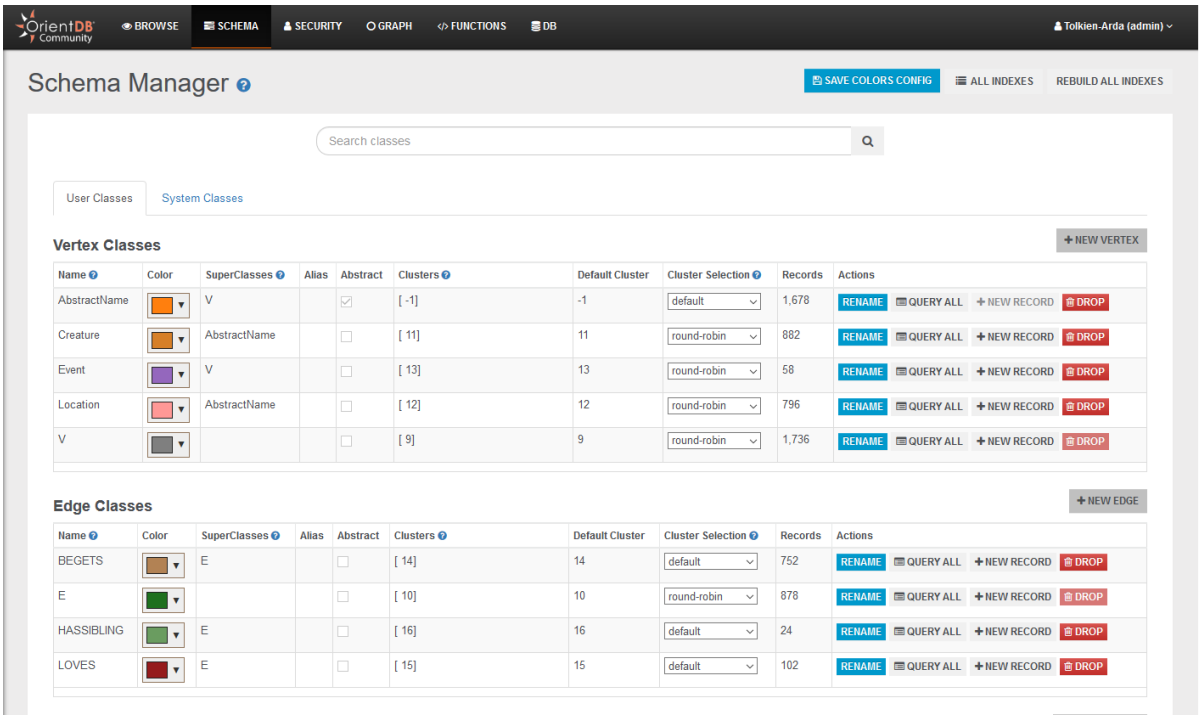

Figure 10 Schema management tab

- **Browse**: tab where it is possible to write the queries and get results in the form of tables. We can get every information from the returned vertices properties, as well as IDs of their relations, to and from each vertex (respectively called "in" and "out" relations). A history of the results is also kept.
- **Schema**: overview of every classes of the database. We can find the vertex classes and the edge classes, as well as generic classes and system classes. It includes the number of records for each class, some customizability options and shortcuts to rename, query, add a record or drop a class.
- **Security**: the security manager allows to specify roles to users of the database. A role can have multiple permissions: execute, delete, update, read and create. It can also be inherited from another role.
- **Graph**: the graph editor is another place to write queries, but here the result is returned as an interactive graph. As showed in figure 11, each node can be selected, edited or even dropped, in the manner of Neo4j.

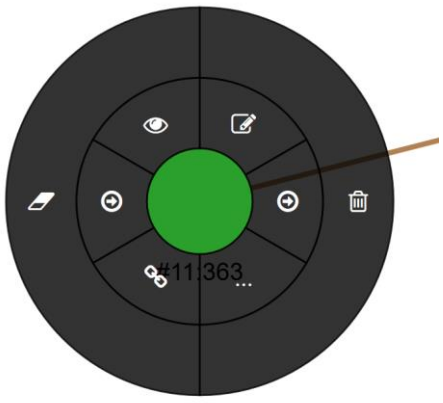

Figure 11 Node edition

- **Functions**: a manager that allows defining custom functions in either SQL or JavaScript. It differs to the queries by the possibility to process operations that would not be allowed natively by OrientDB. It is the equivalent of Stored Procedures in relational databases.
- **DB**: a database manager to notably get an overview of the clusters – with their names, number of records and conflict strategy types – the database configurations (like date and time format, language, country, time zone etc.) and an export option.

Query language

Background information

Historically, relational databases have been the most popular DBMS, hence, the SQL language has become the norm to learn for the developers. Today and since a few years, NoSQL databases have started to become really popular due to the high demand from big data related services like social networks for instance. Let's take a look at the db-engines.com popularity chart:

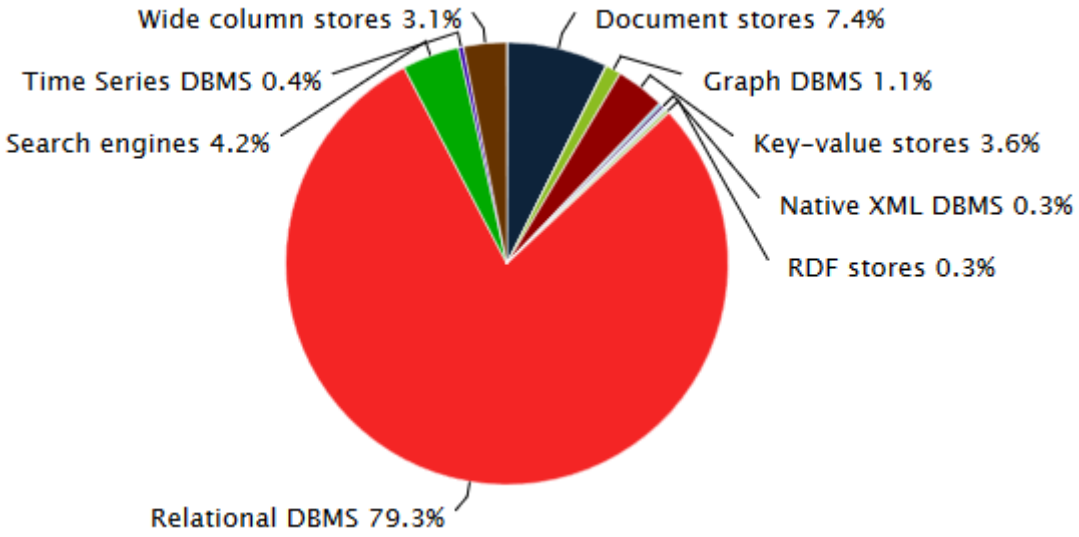

Figure 12 Ranking scores par DBMS category in December 2017

We can see that, although NoSQL databases grow in popularity, the world is still dominated at nearly 80% by the relational databases management systems. It means also means that in schools and Universities, those are the most taught DBMS, and SQL is the most taught query language. It also makes sense because most of the NoSQL DBMS each have a specific query language, or query method to generalize, different from the others. A few examples are CQL for Cassandra, Cypher for Neo4J, HQL for Hibernate, JSON, dynamic-object based language and MapReduce for a lot of Document Store, etc. Some of them are near from SQL, but there is still always a learning curve to take into account.

OrientDB development team wants to pay particular attention to simplicity and efficiency for their system. They built it around the idea of a swift adoption, which explain the little time required to put the DBMS in place. Thus, the team did not want to invent a new language and instead wanted to keep low learning requirements for anyone who wants to try it, so they kept SQL as their query language.

However, even with their desire to keep the most familiar version of SQL possible, the language still have to adapt for the NoSQL concepts.

OrientDB SQL

JOIN operator and Links

Because NoSQL DMBS don not work with tables, the biggest change with classic SQL is the removal of the JOIN operator. Instead of joining tables with IDs columns, the OrientDB version of SQL works with links. In practice, it acts the same as if it was two tables already joined, and it is only needed to add a "." between the two attributes. Figure 13 and figure 14 below show an example of the same query in 'classic' SQL and in 'OrientDB SQL'.

```
SELECT *
FROM Creature C, Location L
WHERE C.location = L.id
AND L.area = ["Middle-earth","Rohan"]
```

Figure 13 JOIN in a relational DBMS

```
SELECT * FROM Creature WHERE location.area = ["Middle-earth","Rohan"]
```

Figure 14 Link in OrientDB

For this query, we want to know who are the creatures that are located in the Rohan region. We can see that two tables are involved in the relational query, whereas only on link with a dot is executed in the OrientDB version of SQL.

Quality of life improvements

OrientDB SQL has multiple small changes, here is a few examples.

- When doing a projection, the star * character is not mandatory.
  So the following:
  ```
  SELECT * FROM Event
  ```
  Can become:
  ```
  SELECT FROM Event
  ```
- The SQL keyword DISTINCT become a function in OrientDB.
  So the following:
  ```
  SELECT DISTINCT illustrator FROM Location
  ```
  Can become:
  ```
  SELECT DISTINCT (illustrator) FROM Location
  ```
- The HAVING keyword has been removed, the counterpart of OrientDB can be obtained with nested queries.
  So the following:
  ```
  SELECT area, count(*) AS number_of_rivers
  FROM Location
  WHERE type='River'
  GROUP BY area
  ORDER BY number_of_rivers desc
  HAVING count(*) > 3
  ```
  Can become:

```
SELECT FROM (
    SELECT area, count(*) AS number_of_rivers
    FROM Location
    WHERE type='River'
    GROUP BY area )
WHERE number_of_rivers > 3
ORDER BY number_of_rivers desc
```

This query in particular returns the regions having more than 3 rivers, alongside the descending number of rivers. Here is the result:

PROPERTIES	
area	number_of_rivers
["Middle-earth","Beleriand"]	22
["Middle-earth","Rhovanion"]	15
["Middle-earth","Gondor"]	10
["Middle-earth","Eriador"]	9
["Middle-earth","Rohan"]	7
["Middle-earth","The Shire"]	4

Traversal

Because OrientDB is a graph database more than anything else, links and relationships represent a capital part of how the data is structured. The 'TRAVERSE' keyword has been created to recursively return certain nodes via the edges, a concept that is lacking in standard SQL. The operation itself is also much faster than the equivalent we would do on a relational database with a JOIN.

The Traverse operator has a few parameters:

- **Target**: the target represents your starting point. It can be represented by a class, a cluster (which can be compared to a collection in a relational database) or a set of records.
- **Fields**: represents the vertices and edges to traverse. It can come in the form of 'in(RelationName)' to show that it will traverse any relation that goes into the specified target; or in the form of 'out(RelationName)' to show that it will traverse any relation that goes from the specified target. (See examples later below).
- **Depth**: the maximum depth of the traversal.
- **Condition**: the core condition that indicates when will the traversal stop and what it returns.
- **Limit**: the maximum number of records to return.
- **Strategy**: there is two types of strategy supported by the traversal.

The default strategy is called **DEPTH_FIRST** and will focus and the depth. The ''reading head'' of the algorithm will read a node and then go to the next level of depth, until the end of the branch or until the maximum depth value is encountered. It will then go back to the last intersection and take the next possible path.

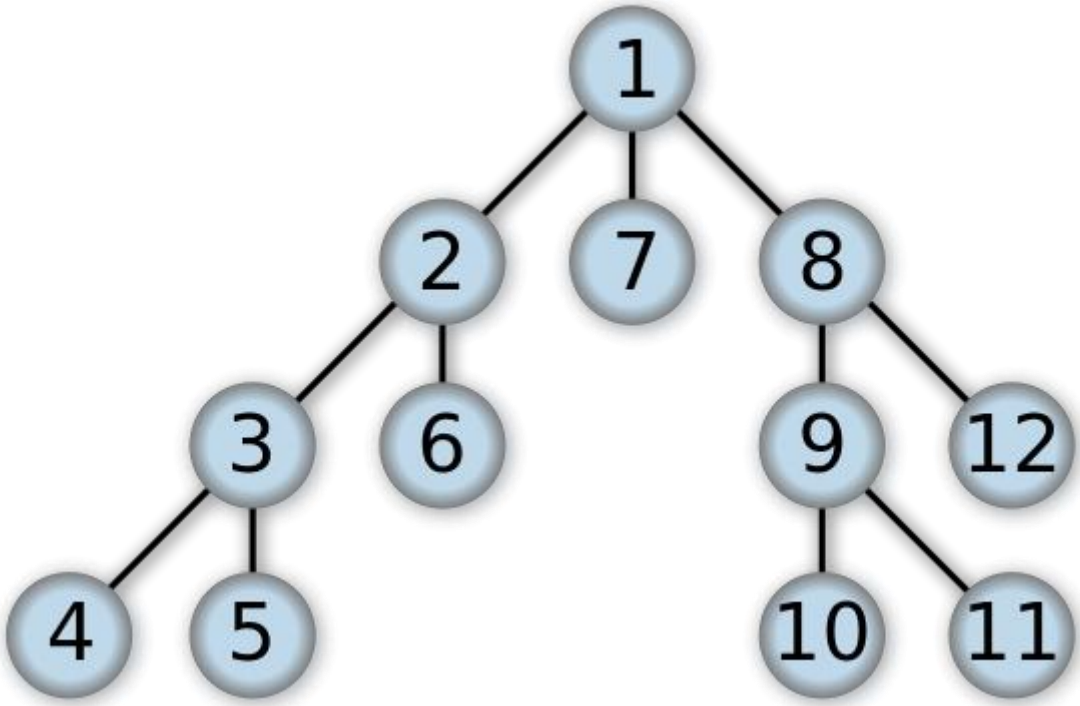

Figure 15 Depth-first strategy

The other strategy, the **BREADTH_FIRST** algorithm, is more "horizontal". Here the "reading head" reads one level of depth, entirely, before going to the next one. Even when a sorter branch comes to its end, the algorithm continues to read the others until the end of the longest branch, or the maximum number of allowed depth has been reached.

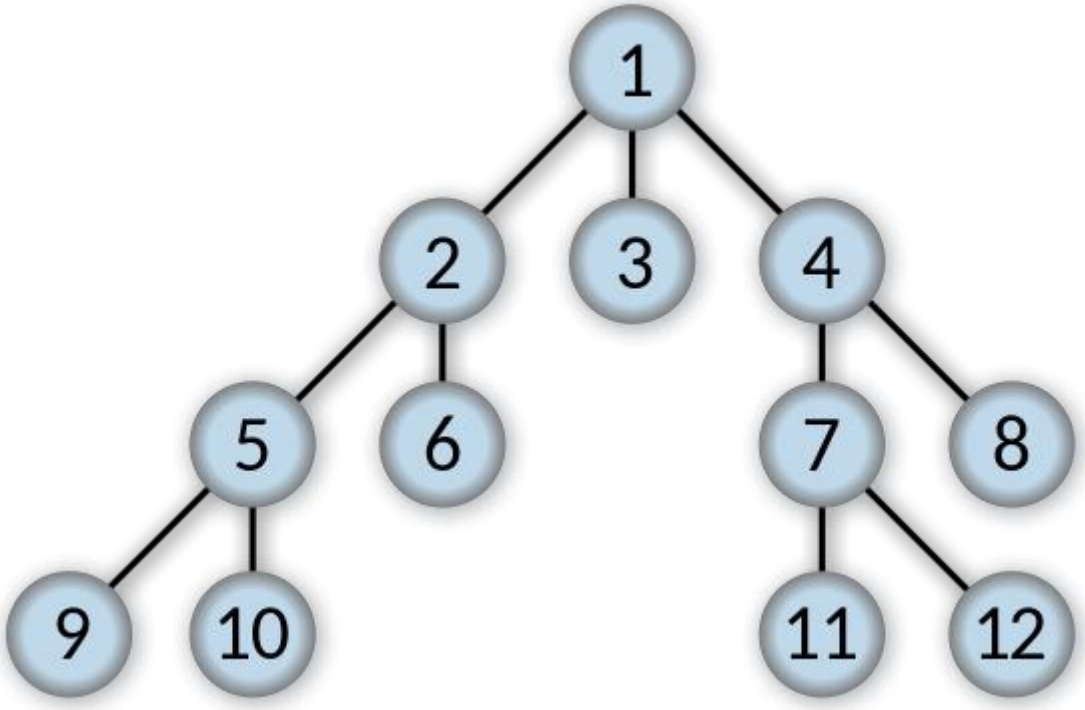

Figure 16 Breadth-first strategy

In the context of traversal, it is also possible to use the provided context variables. Those are dynamic and mainly use to refine queries conditions. They are recognizable with the '$' sign as follows:

- **$depth**: counts the level of depth from 0. This is the context variable used to get the maximum depth.
- **$path**: represent the current path as a string. For example: "#31:0.in.#236:0#.out".
- **$current**: returns the current record.
- **$parent**: returns the parent vertex if any.
- **$stack**: the stack of the current node traversed.
- **$history**: the history of every visited node.

All those concepts and variables stay quite familiar, and with the whole language staying the same, it was more intuitive to work with. An example of a query could be the following. I would like to get a simple family tree of the character Peregrin Took with every relation, whether it is a parent/children relation (called "BEGETS" in the dataset) or a love relation (simply called "LOVES").

The first step to simplify the last query would be to get the "@rid" metadata, which is the general ID common to the whole database in the form of "#XX:XXX".

```
select from Creature
where searchname like "%Peregrin%"
or altname like "%Peregrin%"
or altname like "%Pippin%"
```

Because we don't know the exact name under which he is registered, we try a few query on "searchname" which is a string, and on "altname" which is a collection of strings. We get the following result:

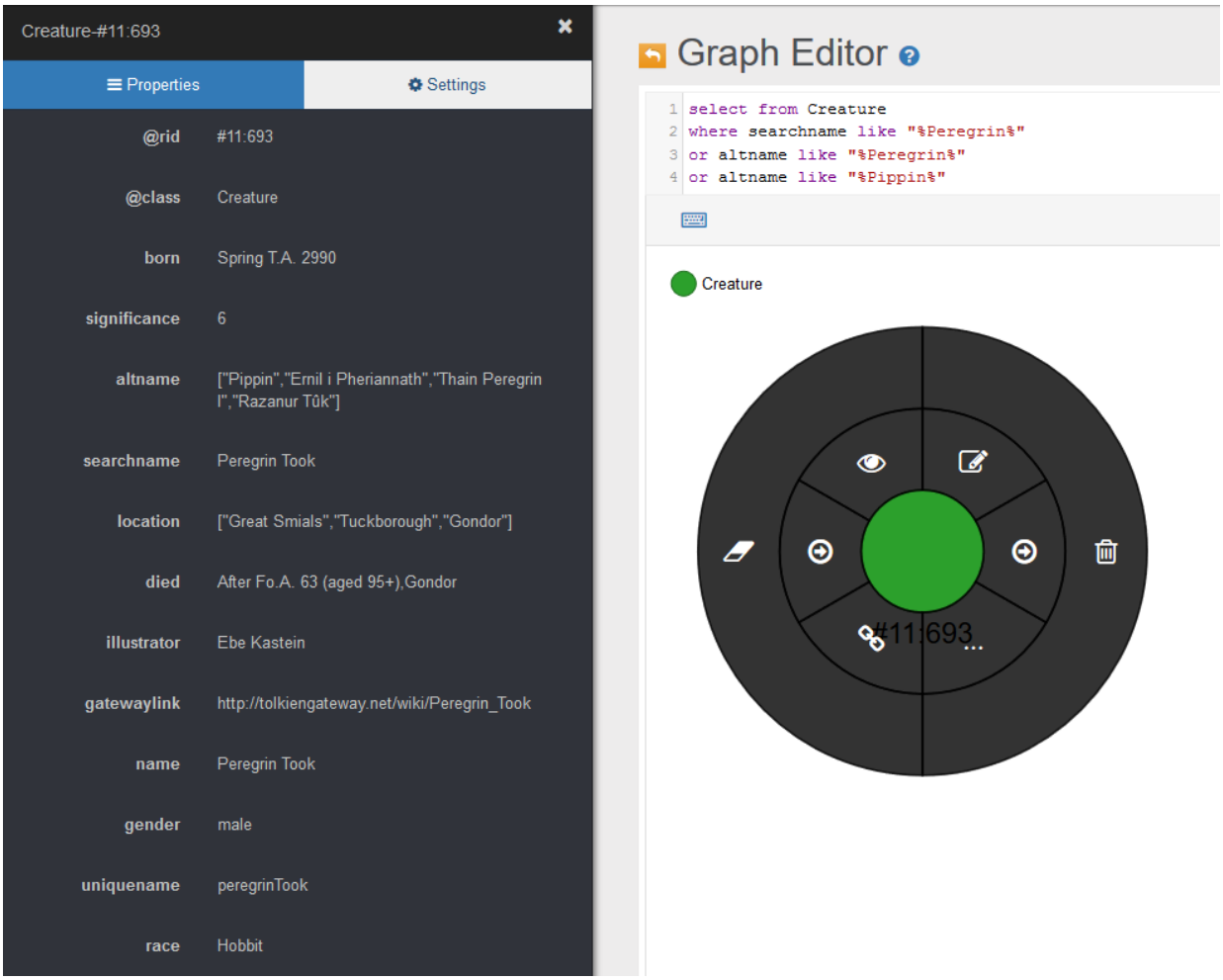

Figure 17 Result of the query to find to ID in the graph editor

The result is one node, on which we could do some actions and see its "in" and "out" relations. On the left panel, we get access to its properties, with among others the alt-names and the search name. We keep the ID so we can re-use it in the next query.

Please note that we could also do the query in the "browse" tab and get the same result. The only difference is that it is displayed as a table with metadata and the relations showed as different columns. Here is the result in this case:

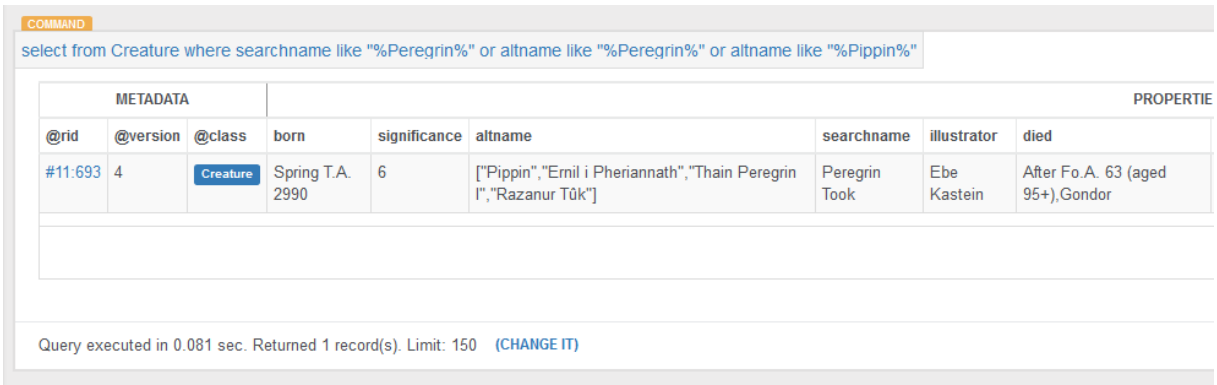

Figure 18 Result of the query to find to ID in the table editor (first part)

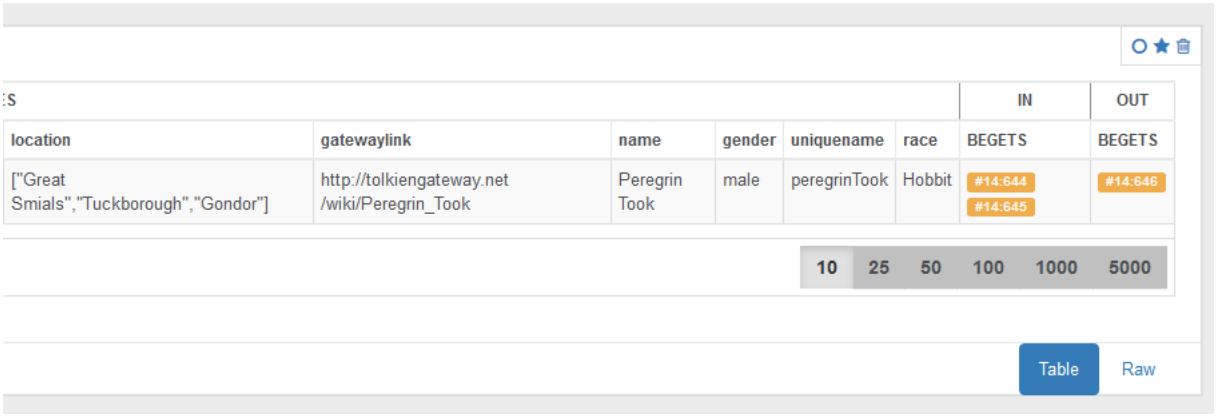

Figure 19 Result of the query to find to ID in the table editor (second part)

Once we get the correct ID, #11:693, we can do the actual query:

```
traverse in("BEGETS") from #11:693 while $depth <= 5 strategy breadth_first
```

Here, we use the traverse function:

- As a field, we use "in("BEGETS")" to describe literally that we want to return the nodes which have the Begets relation into Pippin.
- As a target, we use "from #11:693", the ID matching the character Pippin.
- As a condition, we use the context variable "$depth" to indicate that we return every edges and vertices until we reach a depth level of 5.
- As a strategy, we use the "breadth_first" strategy, which makes more sense when considering the fact that this is a family tree, so we may want all the parents firsts, then all the grand-parents and so on.

The result we get is a bit messy graphically at the beginning, because the engine of OrientDB returns the nodes and relations in the most compact way possible, but once sorted, we indeed get the tree we want.

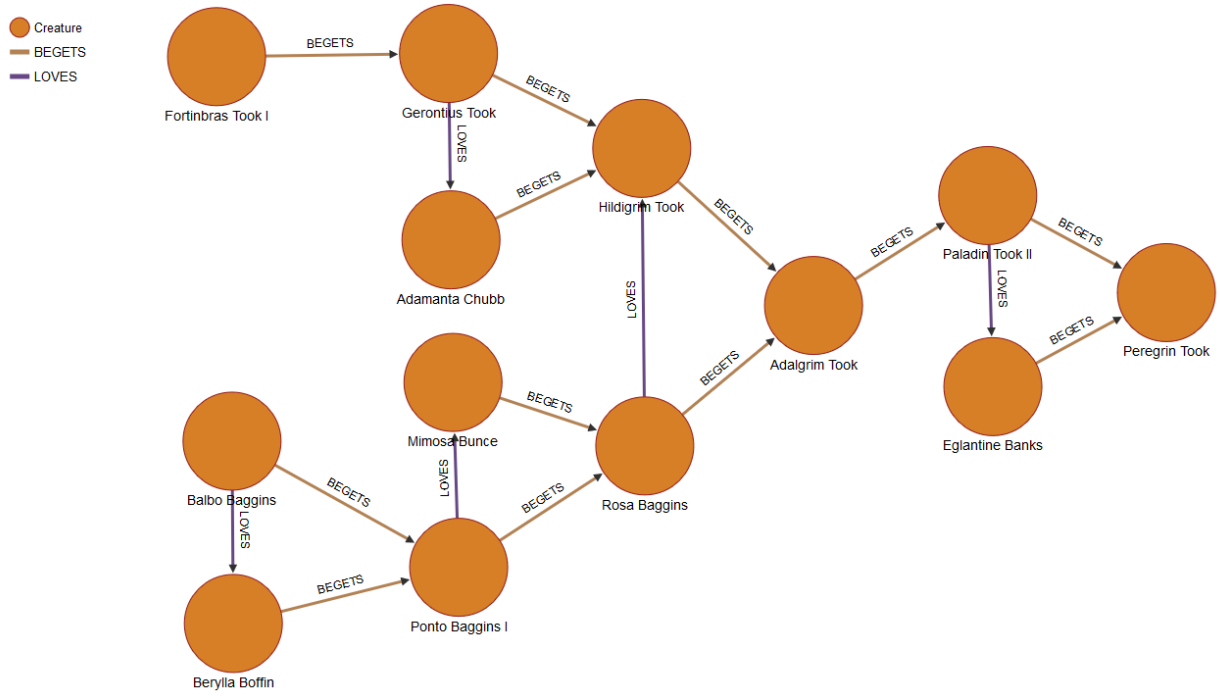

Figure 20 Family tree result

Performance

Neo4J and OrientDB

Another strong point for OrientDB is the claim of better performance than any other graph database. Their main competitor, far more popular as well, is Neo4J. It is understandable, as Neo4J has existed since the year 2000 (even if the first official release was in 2007), whereas OrientDB was created 10 years later in 2010.

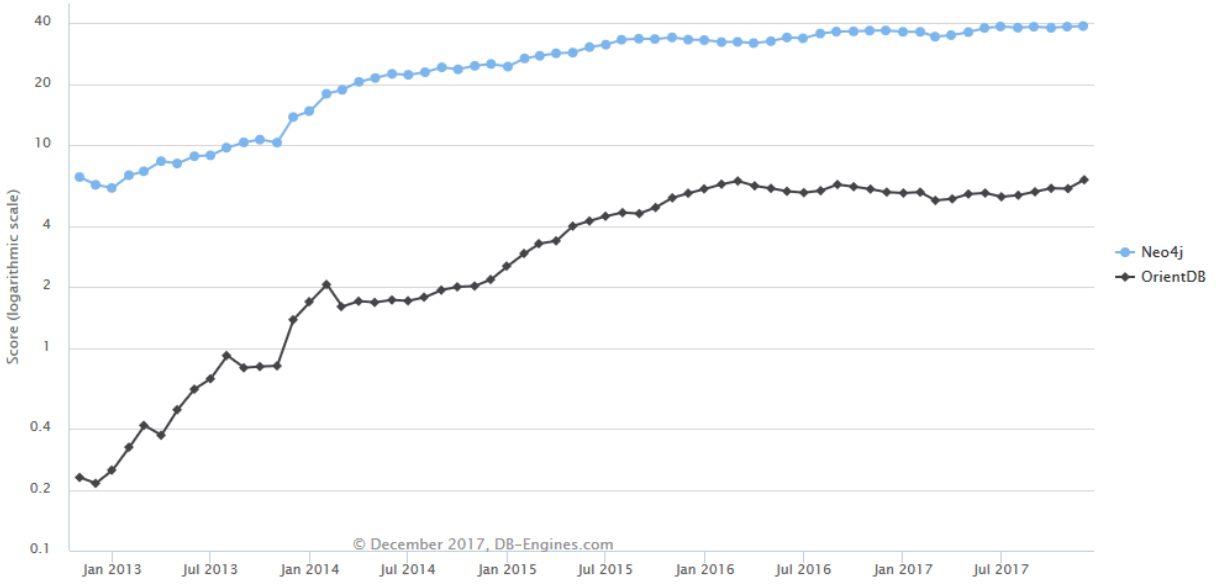

Figure 21 Popularity of both Neo4J and OrientDB over the years (with a logarithmic scale)

Still today, Neo4J stay the most used graph database. Another brake on the popularity development of OrientDB may also be its multi-model infrastructure. People that want to solve a problem with a graph database may prefer Neo4J, advertised as a graph database only, over OrientDB, advertised as a multi-model database.

XGDBench

To measure the performance, an independent team of two researchers, Miyuru Dayarathna, and Toyotaro Suzumura, from the Tokyo Institute of Technology and the IBM Research Lab in Japan, have created a benchmark tool called XGDBench[4] and wrote a paper about it. The tool put the DBMS under pressure with as much as operation they can handle with different workload, and then measure the average number of throughput (in operations/second).

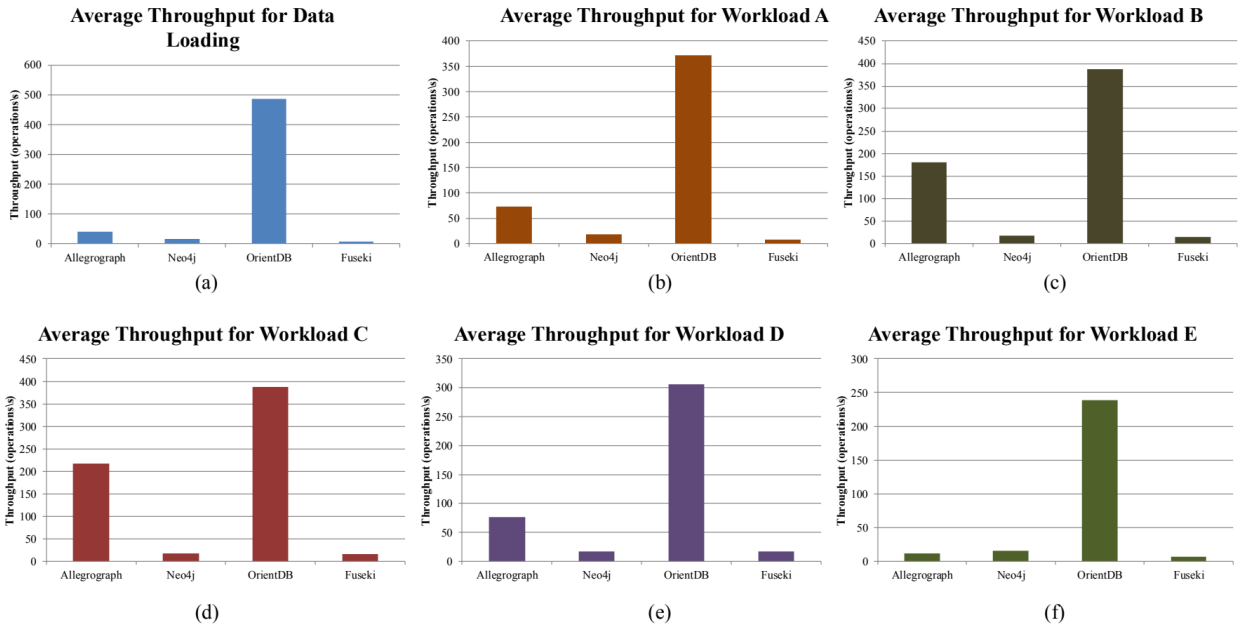

Figure 22 Result of XGDBench with Allegrograph, Neo4J, OrientDB and Fuseki

The results show that overall OrientDB manage to stands out with a higher throughput value compared to the others.

Scaling

Predicting a good scaling is always a challenge as a user of a DBMS during tests phases. Usually, when used in enterprise and fully deployed, OrientDB is meant to be used in a distributed architecture network. It means that several clusters of the database are used at the same time by multiple requestors, and the system must be able to process every query, read and write into the database and without never crashing. To achieve this goal, a replication system is set up.

From 2010 to 2012, OrientDB used the Master/Slave replication architecture. In this structure, every cluster has the role of a slave, except one defined as the master. The slaves contact the master for every CRUD operation (Create, Read, Update, Delete), and the master take their demands one by one to ensure the integrity of the main database. This solution was scaling up well with reads, yet, users were complaining about slow writes operations. This was due to a bottleneck problem: the slave clusters when scaled up where sending to many operations and only one master as not enough to process everything. It was easy to scale either the write *or* the read operation, but the challenge was to handle both at the same time.

In 2012, the development team switched the method to a multi-master replication system. In this case, every cluster is a master, meaning that they all have the rights to read and write on the database. Each master makes sure before operating to prevent any conflict. It also allows for the client to automatically reconnect to another cluster in case of failure. By default, one class of the database is associated to one cluster, however, for the safety and the fault tolerance of the system, OrientDB distributed configuration uses a solution similar to RAID (Redundant Array of Independent Disks) for hard drives. Today, OrientDB also uses the Hazelcast Open-Source project[5], a platform used to efficiently manage data and make parallel calculations, to discover nodes and synchronize certain operations between them.

This whole replication process works well for and allows a great scalability. It may be an important factor for its difference in throughput compared to the other relational or NoSQL DBMS, as it is one of the few to adopt this system.

Conclusion

The fact of being a multi-model database for OrientDB is both a strength and a weak point. It technically supports several models, but it really shines when it comes to graph databases. The developers themselves sometimes consider it more as a graph database in the official web documentation. The other extensively advertised goals, the easy-to-setup and easy-to-learn concepts have been genuinely true from my experience.

OrientDB wants to be the Swiss-knife of the DBMS, replace Neo4j as much as MongoDB or CouchDB. But as said earlier, those tools being more focused on one model, they tend to be, rightly, preferred by users.

Nevertheless, even with those few problems, OrientDB can be a really relevant tool, for example for a side-project in enterprise where you need something quick, efficient, and without high resources and time requirements.

End notes

[1] Discussions about the definition of graph database and its relation with the providing of index-free adjacency.
[2] arda-maps.org, a non-profit, open-source and community driven project regrouping accurate maps, timelines and family trees from the world of Arda (Tolkien). The project is using OrientDB as its main DBMS.
[3] Reactome is a database around biological pathways, human biology and organisms biological processes.
[4] XGDBench: A Benchmarking Platform for Graph Stores in Exascale Clouds – Research from 2012, directed by Miyuru Dayarathna and Toyotaro Suzumura at the Institute of Technology/IBM Research in Tokyo.
[5] Hazelcast Open-Source project

Additional references

Official website: http://orientdb.com/orientdb/

Official documentation: http://orientdb.com/docs/last/index.html

DB Engines: https://db-engines.com/en/