Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER IN COMPUTING

MASTER OF SCIENCE THESIS

Analysis of alternatives to store genealogical trees using Graph Databases

Lucía Pasarin Perea

Advisor/s: Enrique Mayol Sarroca, Pablo Casado Arias

June the 21st, 2013

Contents

1 Introduction

This master thesis is expected to give solutions to a current and open problem. Different aspects related to the genealogical tree storage using advanced databases are considered in this thesis. The first important point of the work is the application of software selection techniques to find the best DBMS or the most suitable to be used for a concrete domain. The next point is the use of current graph DBMS, some of which are still in early phases. Furthermore, the main purpose for this thesis is to state different alternatives to store this kind of information and to overview the previous contexts from which we depart.

Consequently, it is important to give a previous context for genealogical trees storage, on one side, but also an evolution of database management systems in a general way. This last approach allows for a better understanding of how graph DBMS emerge, in which context, with which needs and the same for older DBMS kinds, such as the relational one.

In this thesis the advantages of graph NoSQL databases for storing genealogical data will be defended. After that, one of the most important topics of the thesis will be tackled. This relevant topic is the implementation of some operations using some graph NoSQL databases applied to this problem. This will permit carrying out a comparison between all these systems that will be already used and then drawing conclusions about the operations' results in each of them. This last part will be interesting to support our decision of the best possible solution to this open problem.

1.1 Definition of the problem

Storing genealogical or family history data has been present since many years ago and still exists. The concrete problem that is tackled in this thesis and, then, motivates it is the one of finding the most suitable storage, in terms of DBMS comparison. This comparison will be done according to the performance of some operations, the effort for deploying, DBMS functionalities, etc.

Graph databases have been used since only a few years ago and they have been presented as a good alternative to store data whose structure is similar to a graph. In this context, they seem to be a better choice than a relational database for many reasons.

First of all, in a graph database, the performance of the system is improved as the data is participating in more relations. For a graph database, relating data is just following links from nodes, which is more efficient than joining tables as it is done in RDBMS. This is just the structure we are looking for, when storing family history data.

Concretely, for this domain, using a relational DBMS, we would need to first split the data into tables to later on putting it again in a graph structure. With graph DBMS, instead, we keep all the time the original graph structure without the need of a transformation. This, and more advantages like that, will be described and justified in the corresponding section (concretely, the one about databases history).

After considering some advantages of graph databases, what is aimed is to find the concrete DBMS that achieves these properties in the best possible way among some that will be first picked.

Finally, we would like to conclude that the main aim of this work consists of what follows. Originally family trees were stored in relational models in spite of being actually shaping a graph. As a consequence of this fact, these systems don't work in a way as efficient as it should be. Now, with the emergence of graph NoSQL DBMS, we see a clear opportunity for trying a new system that better suits our concrete domain needs.

1.2 Definition of the goals

The objectives we propose to solve the problem we address with this thesis are:

- Analyze the limitations of the relational model and the opportunities of Graph NoSQL model to store data structured in a similar way as a graph.
- Perform a first analysis of graph DBMS to select a subset of them and compare.
- Define adhoc comparison criteria for the selected graph DBMS in a detailed way.
- Build a prototype of each system in order to improve our comparison.
- Evaluate the systems according to the criteria in order to be able to recommend a good solution in this field.

Now we give a more detailed description of the main goals we expect to achieve:

- **Analyze the limitations of the relational model and the opportunities of Graph NoSQL model to store data structured in a similar way as a graph.**

  We have already noticed that relational database management systems have many disadvantages for storing this kind of data and query it. Then, the idea is finding the graph NoSQL DBMS whose features are the most appropriated. We will perform a concrete analysis applied to storing genealogical data. This process of finding the most suitable database will be based on the evaluation and comparison of several systems and based on criteria properly defined.

- **Perform a first analysis for getting some graph database management systems to compare.**

  This first analysis will be based on the first impressions and information of systems available on the Internet, without considering a prototype implementation. Concretely, we will base on social networks, graph database management systems' official web pages, etc. At the end, we will select a subset of graph DBMS to compare in more detail.

- **Define the comparison criteria to be used to compare the DBMS's in a detailed way.**

  In order to get these criteria to compare the DBMS already picked, we will use some tools and techniques of a specific software selection methodology that will be explained later. We consider that using specific indicators applied to the genealogical domain to classify graph DBMS may help us get a much fairer result.

- **Build a prototype of each system in order to be able to compare.**

  Another important goal is being able to build a prototype of genealogical tree storage, based on each one of these graph DBMS picked in the initial selection. One of the most important motivations of this thesis is finding a justification or proof that the graph DBMS that are finally chosen are the best ones. Therefore, a practical point of view with the implementation of such systems for storing this data using each DBMS was considered appropriated and necessary for the thesis.

- **Evaluate the systems according to the criteria in order to find the best possible solution.**

  We aim at drawing some conclusions about the suitability of the graph NoSQL database model to store genealogical tree information. Then, we will be able to defend which of the evaluated systems is the most appropriated. This is the reason why the comparison criteria are so important. With this thesis, it is intended to provide a solution to this open problem with a fixed context (mainly, a fixed data type to work with, i.e. genealogical data), using fixed comparison criteria. In this way, the result is a concrete solution and we reduce the future effort for getting an implementation given a concrete domain. Here the context is already set and this allows for getting relevant and concrete results.

2 History of Graph DB and NoSQL in general

2.1 Introduction

Database management systems have evolved over the years according to the industry concrete needs. Nowadays, we are in a moment in which the quantity of data and information to be managed by the enterprises has considerably increased and also these companies start realizing that their systems are slow, big and expensive.

Additionally, the need of performing analysis on data has become more important in the last years. Moreover, the use of this data put together with information from social networks became relevant too. All these facts must also be seen in the context of an important evolution of the Big Data[1] and Cloud Computing[2] trends.

In this context, NoSQL databases appear and present a way of storing data, which is cheaper than the relational, smaller, more flexible in many senses and faster. Many startups observed big companies' experiences with positive results (Google, Amazon, Facebook, etc.) and decided joining.

Among all these database trends, a small subgroup of these “new“ NoSQL DBMS called Graph Databases emerges. Then, it also starts growing slowly with the support of such big companies as Twitter, Deutsche Telekom or Cisco. Graph databases arise as a good option for storing social network, routing and recommendation data and, what is better for us, also genealogical data.

[1] Big Data refers to large datasets used to keep derived information including analysis, visualization and several operations over the data.

[2] Cloud Computing consists of storing many users' data in remote servers in accordance with a Software as a Service (SaaS) business model.

2.2 Relational Database Management Systems

Relational systems appeared around the 1970's when database researchers such as E. F. Codd [CODD] were willing to find a good way of storing and querying data according to a very concrete needs. First of all, they required a system that permits querying maintaining an adequate independence of the way the data was stored. A clear example of this is the fact that they wanted a differentiation of a physical ordering from a non-physical one. That means, being able to present a result sorted without needing to physically move records. This was really important to guarantee an efficient retrieval of the ordered data since physically moving records depending on the queries has a high cost in I/O and, therefore, in time.

Another important and needed separation between physical storage and application was in the use of indexes. This means that database clients should not change their way of querying the data depending on the existence or not of indexes. Indexes should make queries faster but applications or clients themselves should not directly use them. Also regarding indexes, they were required to be "non-essential" for the database, i.e. if they are removed, the database could still exist and be working. Then, indexes are seen as mere chains of indexed values copied from the database values.

Relational database management systems, then, contributed with an important concept that would be later on one of the basic ideas in the databases field. That relevant concept was the separation between logical and physical schema. This was, as has been said, an important evolution, since it allowed people working with databases not worry about the way the data was stored and work at a higher abstraction level.

Furthermore, the relational model, as its own name suggests, was intended at providing a new, different way of using the data: the relationships. Data relationships are defined as mathematical relations in which the sets that are related represent database domains, aka attributes. This implied an important evolution in the database fields since until that moment the data was queried as data files, in a tree structure. The previous approaches, for example, didn't provide effective ways to treat data redundancy and consistency that this new representation did.

There are many database management systems based on the relational model. Some of them are Oracle Database, MySQL, PostgreSQL, Microsoft SQL Server, MariaDB, etc. According to the ranking presented in [DBRANK], the most popular relational database management systems are Oracle Database and MySQL. So, we consider relevant explaining some main features about them.

Oracle Database was the first commercial relational DBMS using the SQL language which came on the scene in 1980. It is implemented in C and C++ and it is available to be run on many operating systems and using some different programming language like Java, C or C++. For managing an Oracle Database using these programming languages, the well-known JDBC (Java Database Connectivity) and ODBC (Open Database Connectivity[3]) access methods are available, among others. Oracle Database also permits server-side scripts using its own language, the PL/SQL or Procedural Language/Structured Query Language, considered as an extension for the SQL language. [ORACLEDB] [ORACLETIME]

MySQL, instead, is an open source project that was born in 1995 with the aim of obtaining a database system to access low-level routines. That initial database management system was called mSQL and they realized, after testing it, that it didn't meet the required need for being too slow or lack of flexibility. However, they still kept the syntax used for this older system to ease migrations to what we nowadays know as MySQL. In comparison with Oracle Database, MySQL provides compatibility for more programming languages that Oracle Database does not support, especially

[3] ODBC is a C middleware API. JDBC can be seen as a concrete implementation of ODBC for accessing a database from the Java programming language.

some script languages. Some examples of this fact are languages like Ruby, Python, PHP or Perl. [MYSQL] [MYSQLHIST].

To end up with this part about relational database management systems, we would like to summarize the main limitations they have to store and manage genealogical data. These limitations are, especially, the fact of being oriented to relations and the need for a data transformation into table. In the next point we revisit the characteristics and limitations cited here to talk about how they are treated in NoSQL systems.

2.3 The emergence of NoSQL systems

NoSQL databases have been used since only a few years ago, approximately at the end of the 1990's. They were born to deal with new technological needs and, concretely, with the fact that companies now have huger amounts of data to store in their systems, but they still need an efficient retrieval of this information.

The term NoSQL first appeared in 1998 to refer to some databases that were working without using the SQL language. Later on, on 2009, when Last.fm developer Jon Oskarsson organized a meetup in San Francisco, people attending the event could hear that term again. Little by little, NoSQL started to be used for systems, created by some startups, to deal with some problems that relational DBMS could not solve.

With the emergence of Amazon's Dynamo and Google's BigTable, some other companies also started creating their own systems to response their concrete needs. It is important to keep in mind that NoSQL systems were born in a moment when some other alternatives to the use of relational DBMS had been proposed before. Some examples of these previous approaches trying to be the alternative to RDBMS are object oriented DBMS or XML storages. However, in both cases, none of them achieved a minimum acceptance as NoSQL later on did [NOSQL].

We know that relational databases as MySQL or Oracle fulfill a set of properties called ACID (Atomicity, Consistency, Isolation and Durability) that guarantee that the database transactions can be executed in a reliable way. NoSQL databases instead break a little bit with this databases tradition and propose what is called eventual consistency, i.e. that the database will be consistent at some point if enough time passes. Note, however, that some NoSQL database systems fulfilling ACID properties also exist. An example of this is graph DBMS Neo4j.

Additionally, NoSQL systems provide better ways than RDBMS to scale in a horizontal direction. That means, they make it easy to add new machines to a database cluster and don't rely in highly available hardware. As we know hardware can fail, the system must be able to send requests to other nodes of the cluster that are available when a response is needed.

NoSQL databases can be classified into three main families according to the way they store the data: document store, key-value and graph. Furthermore, an additional category for DBMS implementing Google's BigTable system is sometimes included; as well as a category for column-store systems.

In key-value systems, data is kept by pairs of key-value in a map structure with the characteristic of being schema-less. Records are indexed and queried by key to retrieve the value and the system cannot respond to queries by values. Consequently, only one disk access is required, allowing fast data lookup. Some examples of key-value systems are Cassandra (developed by Facebook), Voldemort (used by LinkedIn) and Riak.

In document store systems aka document-oriented DBMS, data is stored in documents (records) that are part of a collection (the equivalent to a relational database table) and physically stored as JSON, BSON, XML, etc. Document-oriented DBMS provide, as an advantage with respect to key-value systems, the possibility of querying by a different value than the key or record identifier [COMPDOCKEY]. Some examples of document store systems are MongoDB, CouchDB, CouchBase and ArangoDB. We can consider, according to [TOPBIG], MongoDB as one of the leading big data storages together with Hadoop.

We also have graph database management systems, in which data is stored by nodes connected by relations. We talk about these DBMS in more detail in next section, where we give an overview of the features considered as the most important ones. Advantages with respect to both relational DBMS and other NoSQL systems are also provided.

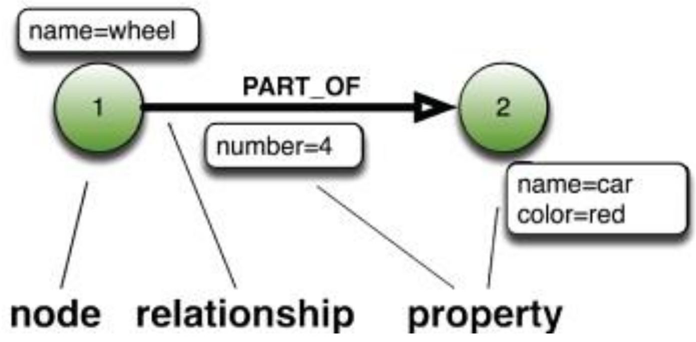
There also exist column-based NoSQL databases which store data in tables as the relational ones but instead of doing it by rows, they store the data by columns. This is a good approach for retrieving data, since only the needed attributes (rows) are retrieved. However these databases are not suitable for insertions or updates of the data because they require multiple accesses for inserting/updating each attribute [CST]. Some examples of column-based DBMS are HBase, MonetDB and Vertica Analytic Database. The last one, Vertica Analytic Database, was acquired by Hewlett Packard in 2011 [HPACQ]. HBase is the most popular among them and is an Apache project that can run on top of Hadoop.

2.4 Graph NoSQL Database Management Systems

2.4.1 Overview

As we said in last section, now we are going to give some main ideas about graph database systems. This includes some features and properties that also derive in advantages with respect to other systems.

First of all, as we know, graph DBMS store data in a graph structure in which the information is kept in nodes or vertexes that relate to each other by means of edges or relations. They provide, then, an efficient retrieval for highly connected data, which is an advantage with respect to both relational DBMS and document store systems.



Basic terminology for labeled property graphs. Source: [INFOQ]

With regard to document store systems, we have to note that, even though the graph approach is better for data that participates in many relations, document is better for storing JSON documents [GOODFOR].

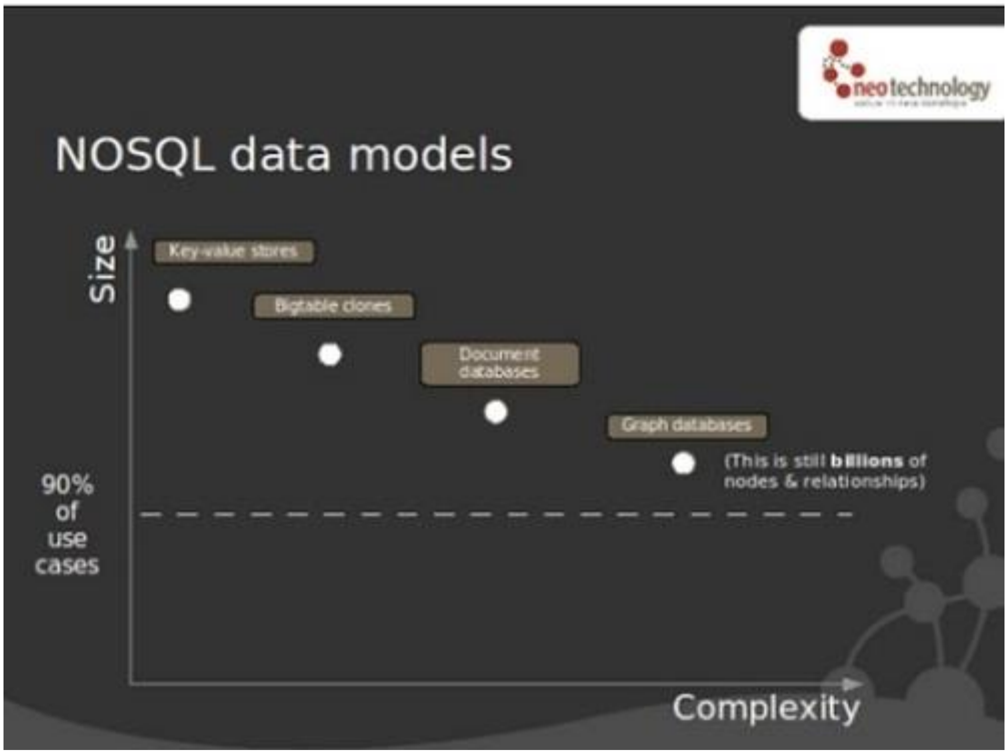
Moreover, graph DBMS provide an additional advantage which is a good support to tackle recursion. This is an important feature that is missing in relational DBMS and that is essential for retrieving all data that participate in a graph structure.

When storing graph data, a typically required operation is getting, given an initial node, all its data and, recursively, all the data of each target node obtained by navigating from the initial one. This is seldom supported by relational systems and, in case they provide it, it is not in a natural way.

Modelsoft Consulting Corporation member Michael Blaha describes this RDBMS problem in [GVSSQL] as "*Relational databases have poor handling of recursion. I will note that the vendor products have extensions for this but they aren't natural and are an awkward graft onto SQL. Graph databases, in contrast, are great with handling recursion. This is a big advantage of graph databases for applications where recursion arises.*"

Finally, we can summarize this comparison between graph databases and

other NoSQL systems with the following diagram. We can clearly observe how graph NoSQL DBMS are good for building complex systems or domains. They are, however, worse than other NoSQL DBMS at dealing with many database entries, but database sizes they are able to handle are of billions of nodes and relationships.



NoSQL data models compared in terms of Size and Complexity. Source: [INFOQ]

2.4.2 Justification

We focus this thesis on graph NoSQL databases, as for the kind of data to be stored (genealogical trees) is adequate. This is due to the fact that many relations between data are present and we know that graph databases are suitable for joining data.

First of all, in a graph database, the performance of the system is improved as the data is participating in more relations. This is just the contrary to a relational database and exactly the structure we are looking for when storing family history data.

Furthermore, if we used a relational database, a structure or format conversion should be carried out and this would be a loss of efficiency for many operations and, especially, for queries. Finally, relational database expected times for queries are affected as the database grows, which is expected not to be happening when using a graph database [APSTGC].

We also choose graph databases because they provide a flexible schema that would be interesting for modeling family relations. We have to take into account that some people have two brothers but some other don't have any, for example, and this is allowed in these systems.

Furthermore, we have to keep in mind that a family tree does not represent a tree structure. In fact, family trees are modeled as generic graphs since starting from a node we may have more than one way to arrive to a given ancestor. An example of this last fact could be going from a node to his father vs. going from a node to his uncle and then navigating to this last node’s brother. Thus, the brother of someone’s uncle may be the same as someone’s father. In conclusion, the fact of representing family trees as graphs may be an indicator that graph databases are adequate for storing this information.

3 State of the art

3.1 Analysis of previous works about graph DB comparison

Many studies have been carried out on the topic of trying to find the most adequate graph NoSQL databases. We can cite, for example, *Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark* [DPHSGAB]. It is a survey by some people from DAMA-UPC research group in which Neo4j, Jena, HypergraphDB and DEX graph databases are compared. Then, the conclusion is that Neo4j and DEX are the ones that, according to the experimental results, are more efficient than the other two databases.

Another interesting previous work comparing graph databases is *A Comparison of Current Graph Database Models* by Renzo Angles from the University of Talca in Chile [ACCGDM]. In this work the following graph databases are compared: AllegroGraph, DEX, Filament, G-Store, HyperGraphDB, InfiniteGraph, Neo4j, Sones and vertexDB. They are compared in terms of "Data storing features", "Operation and manipulation features" and "Graph data structures" compatible with each system.

Furthermore, the query language, API and data representation (nodes, relations, properties), as well as the presence of certain graph queries are compared. The result of this work is that, in terms of "Data storing features", the best database is HyperGraphDB. However, regarding "Operation and manipulation features", they prefer AllegroGraph and Sones.

Then, comparing "Graph data structures", the one with the biggest quantity of structures supported is Sones. When comparing the "Representation of entities and relations", the two DBMS considered as winners are DEX and InfiniteGraph. But, in terms of query support, AllegroGraph is the one supporting all the points considered, and only one of them with partial support. Finally, regarding the "Comparison of integrity constraints", DEX was the best positioned in this study. An important conclusion of this work is that, as the results obtained are based in a non-empirical study, they leave as future work the development of an implementation.

The same author of the last study coauthored with Claudio Gutiérrez also compares many ways of storing graph information in a theoretical point of view in *Survey of Graph Database Models* [SGDM]. In this work, the authors consider some models different from Graph NoSQL model, like GROOVY[4], XML[5], RDF[6] or OEM[7]. Some of the conclusions or results extracted from this study are that XML could be adequate for storing graphs as their structure is hierarchical. Furthermore, RDF is considered as a good model because of its "ability to interconnect resources in an extensible way".

Looking at all these previous works or studies we can observe one common thing. This is, that all of them are context-less and so the databases can only be compared in a theoretical way or without a concrete need for using them. Considering this, this thesis is aimed at analyzing some graph databases in the same way as it was done in previous studies. However, this time we try to get more useful and applicable results by making the context (data, operations, etc.) more concrete.

Consequently, in this work we don't use exactly the same graph DBMS as in the previous studies, and this is because here they have been picked according to our particular needs. We have noticed the importance of the kind of data and the operations to be performed over the data and then we have proceed according to it.

[4] GROOVY is an object-oriented database model that uses hypergraphs.

[5] XML format stands for eXtended Markup Language.

[6] RDF stands for Resource Description Framework, a format used for the semantic web.

[7] OEM means Object Exchange Model and it is used to exchange data between object-oriented databases.

4 The domain. Genealogical trees

As it has been said before, we consider that this thesis differs from some other previous studies because of including a concrete domain. Thus, this domain should be explained before starting with the main topic. In this way, we make sure that we are taking into account all the features and properties of these trees that will determine, later on, the selection criteria.

Genealogical trees are the graphical representation of a family relationships. One of the most popular representations is the ascendant or descendent tree, where oldest family members or ancestors are on the top of the tree, while descendants are in the bottom, hierarchically organized. Family trees have been built since long time ago, for example for representing members of a kings' dynasty.

From a graph theory point of view, we should highlight that family trees are not exactly trees, but graphs. This is due to the fact that there may be more than one path to arrive from a node to another one. However, we have to note that family trees have some tree properties: being acyclic (a person's descendant cannot be at the same time his/her ancestor) and directed (relationship directions are semantically important).

Another peculiarity of family trees is not having a regular or fixed structure: a family member can have whatever number of children. Thus, it may not be a good idea to represent them for example as binary trees. So, we may think about the need of a schema-less storage.

We also have to note that family trees can be of any size: we may include only the living members of a person's family which in general would be a rather small tree. However, we could also draw a family tree representing a whole dynasty of Chinese emperors which are usually quite large.

In this study we observe a previous context in which genealogical data is stored in a format called GEDCOM. GEDCOM is an acronym that stands for GEnealogical Data COMmunication and consists of storing the data in a plain text using certain tags or labels. These tags are used to represent people or individuals (INDI), families (FAM), some people properties like the name (NAME), sex (SEX), birth date (DATE), etc. GEDCOM files are written by lines and by hierarchical levels indicated with a number as a first character (e.g. “1 NAME Bob /Cox/”). They use the “.ged” extension and represent a standard for the exchange of genealogical data between different software.

Although GEDCOM is a standard format, we have to note that many genealogical software don’t make this treatment to it. As we can read in [GEDCOM_SYNC], there are genealogical programs that adapt GEDCOM in many different ways. Some of them may not use all the tags that are accepted by the standard or even add new tags that will be unknown by other programs. They may also interpret the same tags in different ways, assigning a different meaning compared with other genealogical software.

All these problems of not correspondence respect to the language being used have a direct influence in data migration. Migrating from a specific software to another one will possibly imply many compatibility problems even using both programs the GEDCOM “standard”.

Considering these issues related with the use of GEDCOM, it would also be interesting if, with this current work we could draw some conclusions about this topic applied to the resultant implementation. It would be relevant to point out if some standard storing language, possibly based on XML, is found for the storage and/or export/import is found for some of the implementations for each technology. It would be interesting to know about some storage alternatives or evolutions to other formats, such as XML. However, we should keep in mind that nowadays GEDCOM is still the most used and standard format in this field and this is why we base export/import operations on it. I.e., whatever new format different from GEDCOM found for storing data should be more or less easily convertible to GEDCOM, to guarantee a compatibility with previous family tree storage systems.

5 Methodology and strategies to solve the problem

The methodology followed to solve the problem is oriented to Software Selection [QMSSE]. This methodology provides guidelines to face the task of selecting software to be used e.g. by a company. It is important to remark the relevance of a good selection of the software since a bad selection is translated into a loss of performance or time. Performance can be lost as a result of the choice of an inefficient system and time wastage because of the choice of a too complex system. In a company, a bad selection will generally imply an important waste of time and money.

The steps to be followed to find the solution to this problem and according to the Software Selection methodology are:

- **Pick some software options that we could be finally choosing:**

  To do so, we pick these software options based on the technical specifications provided by the vendors. This represents the product/component technical specifications approach. We use this to create a list of them according to these criteria: completeness, technological features (e.g. operating system) and availability of a full version. We also consider the easiness to obtain information about them: use of them, publicity, existing comparisons or surveys.

- **Establish a set of comparison criteria to evaluate these software options:**

  To complete the selection methodology, the approach described in [UQMSPS] has been used for picking the comparison criteria that are used to evaluate the software options. These criteria are: functionality, reliability, usability, efficiency, maintainability and portability. They will be later explained in more detail.

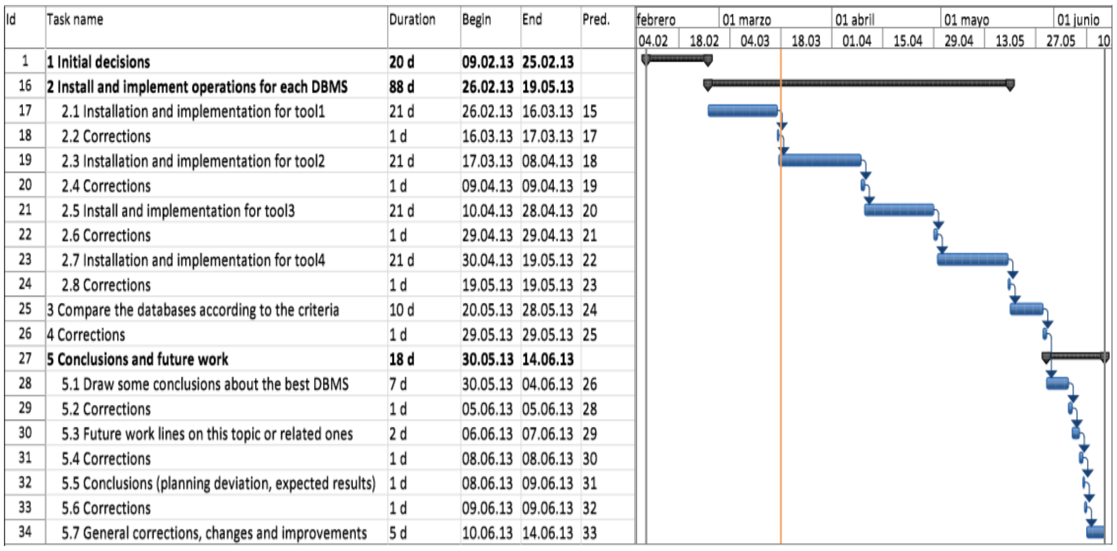
- **Draw a *Decision Analysis Spreadsheet* to visualize the comparison in a graphical way:**

For doing this evaluation, a *Decision Analysis Spreadsheet* is used, since it is a useful tool that allows for comparing different items. In this case, the items to be compared are software options and they are compared according to the score that they have on each of the comparison criteria. Additionally, a *Decision Analysis Spreadsheet* also allows for assigning a weight (in percentage) to each criterion to represent its importance. In this way, we can translate the comparison problem into a quantitative problem.

6 Planning of the work in tasks

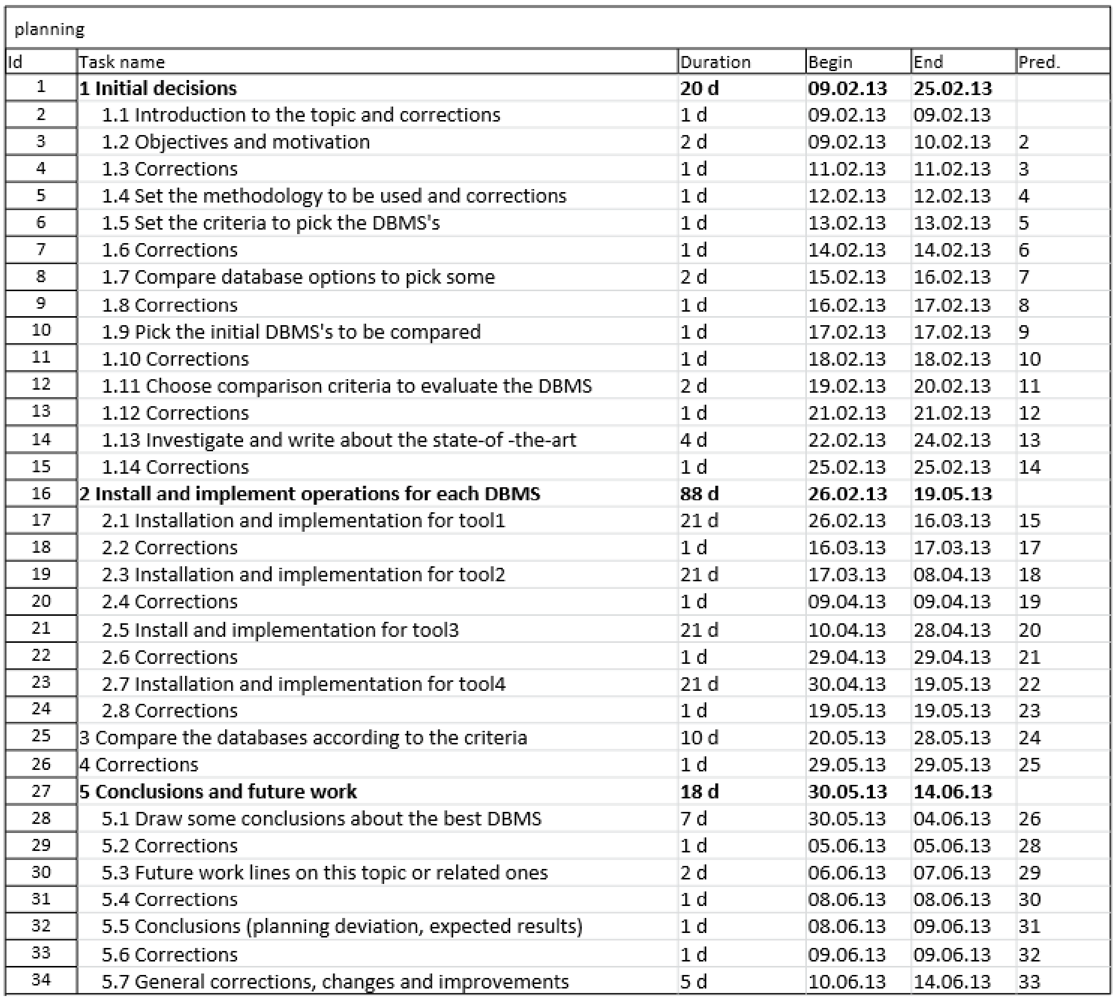
Here the work tasks are broken down, planned and presented in the following Gantt diagram. We can observe that the beginning of the project is the 9th of February and the expected ending of it is the 14th of June. We can also see that the tasks expected to take most of the time are the ones about deploying and installing. We consider that this part and also the final comparison are the most important ones. The reason of this is that they decide how to store the data and implement the operations using each one of the graph NoSQL DBMS. Consequently, they are crucial for getting the final results about which DBMS performed in the best way among them. Later, in this thesis, each one of the tasks will be validated to indicate if they were completed in the expected time or not.

This is the separation and planning of tasks represented as a Gantt diagram:



Id	Task name	Duration	Begin	End	Pred.
1	**1 Initial decisions**	**20 d**	**09.02.13**	**25.02.13**	
16	**2 Install and implement operations for each DBMS**	**88 d**	**26.02.13**	**19.05.13**	
17	2.1 Installation and implementation for tool1	21 d	26.02.13	16.03.13	15
18	2.2 Corrections	1 d	16.03.13	17.03.13	17
19	2.3 Installation and implementation for tool2	21 d	17.03.13	08.04.13	18
20	2.4 Corrections	1 d	09.04.13	09.04.13	19
21	2.5 Install and implementation for tool3	21 d	10.04.13	28.04.13	20
22	2.6 Corrections	1 d	29.04.13	29.04.13	21
23	2.7 Installation and implementation for tool4	21 d	30.04.13	19.05.13	22
24	2.8 Corrections	1 d	19.05.13	19.05.13	23
25	3 Compare the databases according to the criteria	10 d	20.05.13	28.05.13	24
26	4 Corrections	1 d	29.05.13	29.05.13	25
27	**5 Conclusions and future work**	**18 d**	**30.05.13**	**14.06.13**	
28	5.1 Draw some conclusions about the best DBMS	7 d	30.05.13	04.06.13	26
29	5.2 Corrections	1 d	05.06.13	05.06.13	28
30	5.3 Future work lines on this topic or related ones	2 d	06.06.13	07.06.13	29
31	5.4 Corrections	1 d	08.06.13	08.06.13	30
32	5.5 Conclusions (planning deviation, expected results)	1 d	08.06.13	09.06.13	31
33	5.6 Corrections	1 d	09.06.13	09.06.13	32
34	5.7 General corrections, changes and improvements	5 d	10.06.13	14.06.13	33

We can also see below the list of tasks again, but this time including the tasks in “1 Initial decisions” expanded:



planning

Id	Task name	Duration	Begin	End	Pred.
1	**1 Initial decisions**	**20 d**	**09.02.13**	**25.02.13**	
2	1.1 Introduction to the topic and corrections	1 d	09.02.13	09.02.13	
3	1.2 Objectives and motivation	2 d	09.02.13	10.02.13	2
4	1.3 Corrections	1 d	11.02.13	11.02.13	3
5	1.4 Set the methodology to be used and corrections	1 d	12.02.13	12.02.13	4
6	1.5 Set the criteria to pick the DBMS's	1 d	13.02.13	13.02.13	5
7	1.6 Corrections	1 d	14.02.13	14.02.13	6
8	1.7 Compare database options to pick some	2 d	15.02.13	16.02.13	7
9	1.8 Corrections	1 d	16.02.13	17.02.13	8
10	1.9 Pick the initial DBMS's to be compared	1 d	17.02.13	17.02.13	9
11	1.10 Corrections	1 d	18.02.13	18.02.13	10
12	1.11 Choose comparison criteria to evaluate the DBMS	2 d	19.02.13	20.02.13	11
13	1.12 Corrections	1 d	21.02.13	21.02.13	12
14	1.13 Investigate and write about the state-of -the-art	4 d	22.02.13	24.02.13	13
15	1.14 Corrections	1 d	25.02.13	25.02.13	14
16	**2 Install and implement operations for each DBMS**	**88 d**	**26.02.13**	**19.05.13**	
17	2.1 Installation and implementation for tool1	21 d	26.02.13	16.03.13	15
18	2.2 Corrections	1 d	16.03.13	17.03.13	17
19	2.3 Installation and implementation for tool2	21 d	17.03.13	08.04.13	18
20	2.4 Corrections	1 d	09.04.13	09.04.13	19
21	2.5 Install and implementation for tool3	21 d	10.04.13	28.04.13	20
22	2.6 Corrections	1 d	29.04.13	29.04.13	21
23	2.7 Installation and implementation for tool4	21 d	30.04.13	19.05.13	22
24	2.8 Corrections	1 d	19.05.13	19.05.13	23
25	3 Compare the databases according to the criteria	10 d	20.05.13	28.05.13	24
26	4 Corrections	1 d	29.05.13	29.05.13	25
27	**5 Conclusions and future work**	**18 d**	**30.05.13**	**14.06.13**	
28	5.1 Draw some conclusions about the best DBMS	7 d	30.05.13	04.06.13	26
29	5.2 Corrections	1 d	05.06.13	05.06.13	28
30	5.3 Future work lines on this topic or related ones	2 d	06.06.13	07.06.13	29
31	5.4 Corrections	1 d	08.06.13	08.06.13	30
32	5.5 Conclusions (planning deviation, expected results)	1 d	08.06.13	09.06.13	31
33	5.6 Corrections	1 d	09.06.13	09.06.13	32
34	5.7 General corrections, changes and improvements	5 d	10.06.13	14.06.13	33

7 Technical development: Selection and Evaluation processes

7.1 First selection of tools

In this section four database management systems are selected in order to be compared in the next section 7.2. First, the comparison criteria to be used to pick these four DBMS are described. Then, the comparison process is carried out and also explained. Finally, we talk about which are the chosen options.

7.1.1 Design of initial comparison criteria

In order to choose the databases that will be involved in the comparison about which is the best possible storage for genealogical data, we use the following selection criteria:

- **The availability of both community and commercial editions.**

  We would like to use a community edition for this thesis as we will work with sample data just for test and comparison purposes. So, we want to make it as cheap as possible. However, we know the interest of a company for using a full version in some cases: when the free version is not complete or to have certain services, like support.

- **The suitability for the kind of data that we want to store.**

  We know the context (basically, genealogical data with many relations and without a fixed schema) and we should decide according to it.

- **The reputation or support among nowadays companies towards the database management system.**

  Choosing already used and tested software that big companies use is interesting as, in this way, the probability of finding support and help in case of being in trouble increases. However, we should not forget the context: a really good DBMS used by many companies may be very bad for us if the data to be stored and the operations required are not the same.

The fact of being multiplatform was not considered as relevant for the selection of a DBMS. We consider that having either a Windows version or a Linux one is enough since both (especially the Linux) can be obtained in an easy way.

7.1.2 Evaluation and Comparison

The database management systems that are used in this initial comparison are all the ones that have been found investigating through Internet. We have excluded in this initial selection the databases having a proprietary license, since with these ones we would not be able download them to test them for the final comparison. The graph DBMS first used are:

- **Neo4j**: It's a leading graph database that fulfills ACID properties and stores the data in Property Graphs. It is considered as thousands of times faster than relational databases and it used by many successful companies such that Deutsche Telekom, Telenor, Mozilla, Cisco, etc. [NEO4J]

- **InfiniteGraph**: Distributed graph database that allows choosing between ACID fulfillment and a more relaxed consistency of the data. It also allows for physically keeping together elements that are frequently accessed. However, we don't choose it because the free version is only available for 60 days. [INFGRAPHDB]

- **FlockDB**: It is a graph database used by Twitter and created inside this company. It allows horizontal partitioning and also permits out of order and repeated writing to guarantee that the data is not lost. Although it is a database for storing graph data, it is not optimized for graph-traversal operations. This last indication, taken from the official web-site makes us discard this DBMS option. For keeping genealogical data, several traversals will be needed to know the ancestors/descendants or family relatives of a person [FLOCKDB].

- **Phoebus**: Implementation in Erlang of Google's Pregel graph DBMS [PREGEL]. As in Erlang distributed systems happens, vertices communicate between them through message passing [PHOEBUS]. No community support for this project has been observed, so this makes us think about better DBMS options than this one.

- **JPregel**: Quite new project also consisting of an implementation of Pregel [PREGEL], but this time in Java. It has been tested for problems like PageRank and Dijkstra's Shortest Path but it is a DBMS that is not very used at this moment. So, we keep looking at more supported options [JPREGEL].

- **ArangoDB**: A quite interesting DBMS option, since it allows for storing using a document model or a graph one. Additionally, it is starting to get some fame through social networks. However, this support is not very important for the graph version. In fact, there are only a few people talking about this DBMS in terms of graph data storage. [ARANGODB].

- **HyperGraphDB**: it is a database management system that allows for modeling data as hypergraphs (permitting N-ary relations/edges between vertices), which is quite interesting for this project. In this thesis we may need to keep more edges connecting not only two vertices but more (e.g.

a relation containing both ‘being son of’, and ‘being grandson of’). They also provide an interesting property when writing and reading the data: lock-free. This means that we can have concurrent writes and reads performed without locking data [HYPGRADB].

- **InfoGrid**: it’s a web graph DBMS with many components or projects that can be used together or separately. It allows for many graph database advantages as better scalability than relational ones. However, there is only few documentation about it and also few people talking about it on the Internet at this moment. It seems, simply looking at the official web page, like it is out of maintenance [INFOGRID].

- **DEX**: it is a database management system developed by Sparsity Techonlogies, a company associated with the UPC (Universitat Politècnica de Catalunya) implemented in C++ and Java. It is expected to allow for a high performance in both ideal and stress situations and it has mechanisms for minimizing I/Os. Moreover, it requires less space as it uses bitmaps for storing the data internally. As it seems to be a very efficient system in many senses, we are including it in our future analysis [DEX].

- **Bigdata**: it is a graph database management system that provides many graph DBMS features: high performance, horizontal scalability, etc. However, it is a DBMS without a notable support in the community. It was possibly difficult to find information about it because its name coincides with the Big Data movement for manipulating and visualizing huge quantities of data [BIGDATA].

- **OrientDB**: it is a graph DBMS that represents a mixture of both graph and document approach. We observe a particularly important support through social networks about this DBMS. Then, it could be thought as a good choice in this sense. Additionally, we notice that it has support for SQL queries, which can be an interesting feature in terms of usability, and concretely learnability for development. Moreover, there are many big companies using OrientDB in production environments (e.g. Sky, Spielo, etc.). [ORIENT] [ORIENT_SLIDES].

- **Titan**: it is a scalable graph DBMS optimized both for queries and store of a lot of vertices and edges and supporting both ACID and eventual consistency. It is typically used with Cassandra, HBase or Oracle BerkeleyDB as backend storage, so it is not a "pure" graph database management system, but requires for a non-graph system to work [TITAN].

- **VertexDB**: it is a graph DBMS written in C, that supports automatic garbage collection and uses JSON as response data format. It is the DBMS for which less information about use and support was found [VERTDB].

7.1.3 Selection

After having seen all these database management systems and, concretely, the properties of each one, we didn't pick some database management systems for many reasons. Some of them are because of their lack of correspondence with the domain for storing this kind of data (e.g. FlockDB). We also didn't include some other because of a poor support (e.g. VertexDB, Phoebus and JPregel). The support was evaluated through presence in social networks and how much they are used by companies, observed in the official web page of the DBMS and/or other official or specialized web pages

As a summary of this discard phase, we can illustrate this process with the following table:

Technology	Support in social networks	Support in specialized pages and forums	Correspondence with the domain
Neo4j	Yes	Yes	Yes
InfiniteGraph	Yes	Yes	Yes, but limited free version.
FlockDB	Yes	Yes	No
Phoebus	Not too much.	A little bit.	Yes
JPregel	Almost nothing.	No	Yes
ArangoDB	Yes	No	Yes
HyperGraphDB	Yes	Yes	Yes
InfoGrid	Not currently.	Not currently.	Irrelevant
DEX	Yes	Yes	Yes
Bigdata	No	No	Yes
OrientDB	Yes	Yes	Yes
Titan	Yes	Yes	Yes, but under-layer implementation is not graph.
VertexDB	Almost nothing.	Almost nothing.	Yes

For the "Support in social networks" we mainly based on Twitter [TWITTER]: presence or not of the corresponding hashtag and number of results when searching.

For the “Support in specialized pages and forums” we mainly based on the result of StackOverflow [STACKOVER]: presence or not of the corresponding tag and number of results retrieved in the search.

Additionally, the results of these web pages were contrasted by searching in Google [GOOGLE] to check that the conclusions taken from them were correct.

Then, as we can see in the table, the final decision is that we will keep **Neo4j**, considering all the enterprises using it successfully and so supporting it. We also keep **DEX**, as it's a promising project expected to be providing a good efficiency with the use of bitmaps. **OrientDB** is also included in this partial selection, for all the community supporting it through social networks. Finally, we are including **HyperGraphDB** with the possibility of storing hypergraphs.

7.2 Detailed comparison of Graph Databases

Once we have picked the four database management systems to be exhaustively compared, it’s time for starting this exhaustive comparison. First of all, the comparison criteria are given, then the DBMS are evaluated and compared according to the criteria and, to end up, the final decision is pose and explained.

7.2.1 Design of comparison criteria

The points or functionalities that have been chosen for the comparison are, as explained in the methodology part and according to [UQMSPS]: functionality, reliability, usability, efficiency, maintainability and portability. For the case of functionality, we consider, among others, some data export operations. For these concrete operations, it would be interesting to determine if we could use GEDCOM or a format easily convertible to GEDCOM.

Then, let's explain in more detail each one of these points:

- **Functionality**:
    - Suitability: if the DBMS is adequate in terms of the operations that we need for our system and the ones that the DBMS offers. For this particular case we consider the following operations:
        - Create family tree: create node/relationship.
        - Modify family tree: change node/relationship properties.
        - Remove family tree: delete node/relationship.
        - Create different relationship types (son, daughter, nephew, niece…) and get all relationship types.
        - Traverse family tree to get all the members and relationships.
        - Determine the relationship between two nodes (shortest path).
        - Get all data about a single node or a subset of all the data (e.g. birth date and place, gender, etc.).
        - Data about a given node's ancestors (e.g. father, grandfather, etc.).
        - Data about a given node's descendants (e.g. son, grandson, etc.).

    - For the two previous cases we have to set the maximum distance between the node and the descendant/ancestor. For example, we consider son as distance 1 and grandson as distance 2.
    - Number of descendants and ancestors of a given node.
    - Export the whole family tree.
    - Export of only a branch of the family tree.
    - Import data from an input file.
    - Query by one or more node/person field/s (e.g. all nodes with age greater or equal than 16).
    - Query for all the nodes corresponding to alive people.
    - Query for a list of all distinct surnames.
    - Query for a list of all distinct birth cities.
  - <u>Security</u>: Analyze if there are effective mechanisms for guaranteeing that the data will be kept in a secure way. Concretely, we require for having:
    - <u>Password authentication</u> capability.
    - <u>Backup of the database</u> data and structure feature and consequent restore.
    - Possibility to define <u>different levels of privacy</u> for the data. This consists of being able to set some information (nodes and/or relations) as public and some others as private by some means. This is particularly important considering data protection legislation (e.g. LOPD in Spain

or DPA in United Kingdom). We have to bear in mind that we can only publish data of people that had died at least 50 years ago, so we would need this differentiation.

- **Reliability:**
    - Maturity: the state of development of the DBMS.
        - If when developing the database we find problems related with features or operations still not available in the current version. This also includes the use of low-level elements and not support for higher ones, for example. We also include here, if any, the case of bugs caused by the DBMS.
    - Fault tolerance: the capability of recovering from an error in an elegant way.
        - We require our database to have mechanisms for a fast recovery after a failure occurs: e.g. ability to switch from one database to another.
- **Usability:**
    - Understandability: the easiness to understand how the DBMS works.
        - We evaluate this by comparing the time spent to read, understand and learn using the documentation resources available for each technology.
    - Learnability: how much time takes learning how to deploy it and implement operations.

        - We evaluate this by observing if the required time for implementing the database was according to the initial planning or not.

    - Attractiveness: although it is generally a subjective point, we measure it by the quality of the tools available for working with the DBMS.

        - Here we see if the system provide different types of query languages.

        - We also measure the quality, if they exist, of visualization tools.

    - About usability, we finally remark that we assume that the final user will not be aware of the graph DBMS used in a final application for managing family tree. This is particularly important for this domain, since these final application's users will be frequently people without technical knowledge and few experience using computers. Therefore, we assume this user oriented usability is fulfilled but, if we observed any case in which the desired DBMS transparency for the user is violated, we would document it here.

- **Efficiency:**

    - Time behavior: how much time performing some needed operations takes (contrasted with the frequency with which the operation will be used or the importance of it).

        - We will evaluate the operations listed in the Suitability point inside the Functionality part of this section. Then, the indicators for these operations will be the presence or not of the operation. Additionally, in some cases talking about the degree of inclusion of the operation would make sense. An example of this would be the number of export formats.

    - <u>Resources utilization</u>: measured in terms of memory consumed. We measure both memory used for queries (in general RAM memory) and memory to store data (in general disk memory), but we only use RAM measurement as relevant information to compare. This is due to the fact that RAM memory is still more expensive than hard disks, so we should try to make good use of it.

        - This will also be measured for each one of the operations in the list.

- **Maintainability:**

    - <u>Analyzability</u>: if we can analyze the database to extract statistics or relevant data about how it is working.

        - Concretely, we care about <u>storage size</u>, total <u>number of nodes, relationships and indexes</u> as general database features. We are also interested in knowing about <u>compaction operations.</u> Since we will be deleting and adding nodes in the database, we would like to guarantee that we can reuse the space occupied by a node that was deleted. Additionally, some mechanisms to guarantee that, in general, <u>the data is occupying as little space</u> as possible are required. This is due to the fact that, as we mention in the domain part, family trees can be of any size.

- Changeability: if it is possible to change the database structure once it is built without hurting it too much.
    - We are interested in knowing whether it is possible to change (extend) for example the relationship types or to add new indexes once the database has been built and/or started. This is particularly interesting in our context and allows us to check about the database flexibility. After all, we know some people can die and some other can be born. Thus, we cannot have a static immutable structure having, e.g. an exact number of indexed values.
- Stability: measured by observing the number of times each DBMS fails. In this way, we can use an approximated failure rate as a stability measure.
    - We include here errors related with the DBMS itself (e.g. if it returns a wrong result for any operation or if an operation returns a DBMS-specific error).
    - In this point we can also evaluate the quality of log data that can be extracted from the database.
- Testability: how easy the testing tasks are when using the DBMS.
    - We observe if the database management system provides or is compatible with some tools for debugging. In this way, we can find errors in the database operations in an easier way.
    - We also see if the system is compatible with some testing frameworks, such as JUnit and other DBMS-specific

frameworks that ease the task of testing the database operations.

- **Portability:**
    - Install-ability: how easy the installation and deployment tasks are. For doing this, we simply keep the time that we spent installing and deploying each DBMS and compare it with the initial prevision. We split these tasks into the following parts:
        - Time spent for downloading. Specially, we describe if a registration was required before starting the download.
        - Time required for installing the database.
        - Finally, we measure the time for getting the DBMS running to be able to execute a first simple operation.
    - Portability compliance: the easiness for exports and imports of the database.
        - We look at the tools or operations available for exporting and importing the database data: different formats, such as CSV, and time spent for doing the task. Here is when we also study the possibility of importing and exporting from/to GEDCOM format.

7.2.2 Evaluation and Comparison

Now we explain how this final comparison, evaluation and results display are going to be done. We explain all this in terms of e.g. how the different comparison elements will be split and auxiliary elements like tables that will be used. The information explained in this point will be later on useful, especially for point 9, in which we develop the comparison and evaluation.

First of all, we separate by technology used (Dex, HyperGraphDB, Neo4j and OrientDB). Then, for each technology, we compare each comparison point (functionality, reliability, usability, efficiency, maintainability and portability) splitting them correctly as explained in last section (Suitability, Security, Maturity, Fault tolerance, Understandability, Learnability, Attractiveness, Transparency for the user, Time behavior, Resources utilization, Analyzability, Changeability, Stability, Testability, Install-ability, Portability compliance). Concretely, we want to use a table in the form of the following one for the "Suitability" point:

	Feature 1	**Feature 2**	**……………**	**Feature N**
Weight or Importance (1, 2 or 3)				
Presence (%)				

Table 1. Sample Suitability table.

In the table above we can observe that we measure the weight or importance of the feature that is being evaluated with the following values: 1 means that the feature is a little bit important, 2 represents a feature that is more or less relevant and 3 represents a very important or crucial feature. The last value should be seen as the adequate for operations whose omission would make the whole system "useless". We have to note here that, of course, the first row (the weight one) will be the same in all Suitability tables. We just repeated these values to have them together with the specific technology results and get the overall punctuation. We also assign a specific color to each technology to make the task of finding the information about each DBMS visually easier.

For the other comparison points a part from the Suitability, we have to remember that some of them were subdivided again. Then, we consider a table like the one above for each one of these subcategories if there are more than one. However, we sometimes group comparison points according to the group they belong (first separation: functionality, reliability, etc.).

8 Technical considerations for the prototype

In this section we are going to explain some design and implementation decisions that have been taken into account in order to develop the prototype for the current thesis. We talk about tools or technologies to be used, class format, method signatures, etc.

First of all, the DBMS that are used in the comparison are accessed via the Java programming language. Among many available options, this was the best one to be chosen especially because of the previous knowledge about this programming language and because all four DBMS support it.

The class structure to be used consists of a PersonDao class that implements an IPersonDao interface. Then, there are as many PersonDao subclasses as database management systems used, that is four. That means, we have the following classes extending PersonDao: DexPersonDao, HyperGraphDbPersonDao, Neo4jPersonDao and OrientDbPersonDao.

Next, we can see the method signatures to know the parameters they use, the type they return and a little bit of their semantics in correspondence with the previous definition of the required operations in section 7.2.1.

For understanding the following signatures we have to note that some Generic types are used [GENERIC]: NI stands for Node Identifier, RI means Relation Identifier and RT refers to Relation Type. What is then done is creating each one of the classes specifying the type for NI, RI and RT for each one of them. The reason for doing this is that node or person ids may be (and in some cases are) of different types according to the DBMS used, and the same for the other generic types used.

Create node/ relation:

```
NI createPerson(long personId, String name, String surname, Date
      birthDate, String birthPlace, Gender gender, boolean alive);

RI createRelation(RT relationType, NI firstPerson, NI secondPerson);
```

Modify node/ relation:

```
void modifyPerson(NI personId, String name, String surname, Date
      birthDate, String birthPlace, Gender gender, boolean alive);

RI modifyRelation(RI oldRelationId, RT newRelationType,
      NI firstPerson, NI secondPerson);

Remove node/ relation:
void removePerson(NI personId);

void removeRelation(RI relationId);

Create relation type:
(We don't include this operation here because it is only implemented
as a Java method in DexPersonDao)

All relation types:
Map<Integer, String> getRelationTypes();

Traverse tree:
Map<String, Map<String, String>> getFamilyTree();

Shortest path:
Map<String, Map<String, String>> getPath(NI firstPerson, NI
      secondPerson, int maxDepth);

All node's data:
Map<String, String> getAllData(NI personId);

Get ancestors/ descendants:
Set<String> getAllDescendants(NI personId, int level);

Set<String> getAllAncestors(NI personId, int level);

Number of descendants/ ancestors:
int getNumberOfDescendants(NI personId);

int getNumberOfAncestors(NI personId);

Export tree:
void exportFamilyTree(ET exportType, String fileName);

Export tree's branch:
void exportFamilyBranch(ET exportType, String fileName, NI
startPerson);

Import data:
void importFamilyTree(String fileName);
```

```
Query by field (age):
List<String> getPeopleOlderThan(int years);

Get alive people:
List<String> getAlivePeople();

Get distinct surnames:
Set<String> getPeopleSurnames();

Get all birth cities:
Set<String> getPeopleBirthPlaces();
```

One of the tools that are used to develop this prototype is **yEd** program from yWorks [YWORKS]. yEd is a tool that allows for data visualization and both import and export. It is very useful here for importing family tree files from GEDCOM format (.ged) to GraphML format (.graphml). As we mentioned before, GEDCOM is the standard format for representing family trees. GraphML is an XML based format used by many of the current graph NoSQL database management systems to import and export data. In this way, they provide compatibility between different systems.

Furthermore, tools like **Maven** [MVN] and **Subversion** [SVN] were used to ease the prototype development. Maven was used as a way to "install" the DBMS by including them in the current project letting Maven resolve dependencies. Subversion was used through the **Subclipse** [SUBCLIPSE] plugin for the Integrated Development Environment (IDE) **Eclipse** [ECLIPSE]. As Maven plugin for Eclipse, **m2e** [M2E] was the one used.

We also have to note that the tests were made with the current last stable versions of each DBMS. Consequently, the results are completely conditioned to that fact: we cannot be sure, for example, if a feature not currently supported by a DBMS will or won't be included in the future.

Regarding the database that will be used, it contains 997 nodes and 1005 relations between nodes or edges. This database size was considered as adequate to test the features to be analyzed, since it was thought as a medium one. It is important to keep in mind this information when doing the comparison, especially when talking about performance or response time.

The data contained in the database comes from a GEDCOM file that was transformed into GraphML format. Then, this last file is read by importFamilyTree(…) database operation. In this operation we extract the information according to some properties or fields contained in the file. In order to perform this operation we use Tinkerpop Blueprints, which is a kind of driver (an analogous to JDBC, used for SQL databases with the Java programming language). Blueprints allows for performing graph operations on top of many current NoSQL databases, such as Neo4j, OrientDB, Dex, InfiniteGraph, Oracle NoSQL[8], Titan and MongoDB.

For the memory consumption measurement, the tool that was used is **Java VirtualVM** [JVIRTUALVM], which is a monitoring tool that provides a lot of information about the current execution. It shows the memory usage in both Java Heap and PermGen, CPU usage, total number of classes involved (including libraries), Garbage Collector activity and number of currently live threads. VirtualVM also provides a useful feature that consists of detecting and giving information about Java processes not running in console but in an Integrated Development Enovironment (IDE) such as Eclipse.

A final consideration that was seen as important for the comprehension of the next points is the way of measuring the resources utilization and performance. For doing this we chose a list of the main operations that were implemented for which we saw the signature before. We, however, excluded from this list the import and export operations, considering them as too long or slow to perform a fast measure and comparison.

We followed the same execution order for all DBMS and the same parameters whenever it was possible (except for the cases in which, e.g., the identifiers were of a different type). We are aware there are some systems that don't support some operations and thus they have

[8] A key-value database created by Oracle Corporation. [ORACLENOSQL]

one or more methods less than the others. However, our results show, in general, a big difference between systems so we consider these missing operations as not relevant. A good example to justify this situation is the case of HypergraphDB, as we will see in the next point.

9 Final Evaluation and Comparison in more detail

9.1 Evaluation and Comparison

In this part of the thesis, the experimental results obtained with the implementation of the prototype are presented.

With regard to the weights representing the level of importance for the operations, we would like to explain the reason why 1, 2 or 3 is assigned to each one of them.

First of all, for the Suitability operations the following is considered. A punctuation of 3 is assigned for the operations of **create nodes and relations** because it is considered that their omission would mean not being able to build the family tree. We also use a punctuation of 3 for **modifying nodes and relations** because we have to keep in mind that, for example, an error could be introduced in the previous operations (the create ones) and we have to be able to correct it. We have to note that the DBMSs that support modify operations without the need of deleting and creating a new node or relation will receive a better punctuation. We, of course, prefer implementations of the modify operations that change the existing nodes or relations.

Likewise, 3 is also used for representing the importance of the **remove operations for nodes and relations**. This is due to the same reason as the modify operations: we may make a mistake creating a person. Thus, the remove operation is useful for correcting errors in which, for example, we didn't want to create a node because it is not present in the current family (e.g. we looked at the wrong family tree or we received false information).

The operation of **creating different relation types** receives a punctuation or weight of 3 because we need these different types to represent the family tree. Without types we may not be able to differentiate, for example, a relation of type MARRIED from a relation of type COUSIN. Furthermore, we consider different node types are not required for this specific domain since we only represent nodes of type PEOPLE.

Traversing the family tree is also considered as a crucial or very important operation. We give a punctuation of 3 to this operation because we understand that being able to display the tree is a basic feature. Otherwise, we would have an adequate structure to store the data that we want, but for which we would not be able to see its content in a given moment.

Furthermore, as with the traverse operation we only would like to get people names and surnames and the relation types between them, we also consider an operation for **getting all data**. This is why get all data operation is also thought as a crucial operation with, thus, a punctuation of 3.

Getting the **shortest path**, the **ancestors**, **descendants** and the **count** of both are operations with a punctuation of 2. This is due to the fact that they are not crucial operations because the system would still work if we didn’t have them. However, they are quite important because, even though the information they provide can be retrieved from the output of the traverse operation, this information is not so immediate. Thus, we would say they are not crucial, but the information they provide is really useful for the system.

For the **query and projection operations** (i.e. query for older that a certain age, get alive people, people surnames, birth cities and all relation types) we assign a weight of 1. With this low value we wanted to represent that the information provided by these operations can be obtained, more or less easily, using the “traverse operation”. So they are not completely necessary.

Finally, for both **export operations**, we have to say that they deserve a punctuation of 2 because although the system can work without them, they are really necessary. This importance relies on the fact that the system is intended to provide compatibility with the standard family tree format (GEDCOM). Thus, a good way to migrate this information to GEDCOM is making sure that our system can support data exports to at least one format.

The same can be said with regard to the **import operation**: we need it to make the use of GEDCOM as much compatible as possible. As it is explained in the technical considerations section, we use yEd tool for that. Furthermore, we have to note the import operation is also important because it allows for having a minimum quantity of people in the system. Without the import operation the system could work but probably with a small size and/or with unreal data.

Regarding the Security comparison elements, we have considered what follows. **All three comparison elements** receive a punctuation of 2 because they represent important capabilities. However, they are not elements whose exclusion could mean an impossibility for working with the system.

For the Reliability part we consider negative weights or punctuations. This is due to the fact that these points are stated in a negative way. That means, we do not write them as the results we expect to maximize (e.g. lack of operations still not available), but in terms of what we want to avoid or minimize (e.g. presence of operations still not available).

Then, for the reliability table we consider a punctuation of -3 for points which, in case of being fulfilled, would make the **system not able to work** correctly or as expected. A weight of -2 represents something **relevant, but not essential** for the system, that is wrong. Finally, using a weight of -1 we mean that something **not very relevant** is missing or wrong.

For the Usability table we consider a weight of 2 for the following comparison points: **time to read and understand**, **adequate implementation time** and **different query languages**. This is due to the fact that they are important features to save time, considering that the development phase should be as fast as possible, in order to get the comparison data as soon as possible. However, they are not basic features whose omission would cause the impossibility to work with the system.

In the Usability table we also have the **visualization tools quality** comparison point. It receives a weight of 1 because it is a feature that is interesting to have, but not too relevant for the development process. Finally, the **user transparency** feature is the most important among the Usability points. It is important since it allows for making the users unaware of low level complexities that have no interest to them.

Time and **resource utilization** are the only points in the Efficiency table. They both receive a punctuation of 2 because they are both important to be taken into account as a general property for all or almost all projects. However, minimizing them is not a main aim of this work, this is rather a functional study. I.e. a work for trying to determine the functionalities that can be supported by certain systems in a given context.

For the Maintainability table, **all its points except for two** are weighted as 2 because they are general database features that we should provide. However, they are not crucial for this concrete work. The points that receive a punctuation of 3 are the possibility of **changing relation types** after having been defined and the **absence of specific errors** directly related with the DBMS being used.

The first one of them is important in order to guarantee that the system developed will be flexible and will allow modifications after creation, such as e.g. including the relation brother-in-law. The second one, instead, is relevant for having a minimum quality and adequate support for the features that are expected to be offered.

The same can be said, finally, for the last table, i.e. the Portability one. In this one, we also consider that most of its points or functionalities are considered as general database features that should be offered. In this case, we only assign a weight of 3 to the **format compatibility** capability, whereas **the resting ones** have a punctuation of 2. This is due to the fact that data import and export are important for the system since we want to provide GEDCOM compatibility in the easiest possible way.

9.1.1 Dex

After implementing the operations we got the results that are reflected in the following Suitability table belonging to the **Functionality** group:

	Create node/ relation	Modify node/ relation	Remove node/ relation	Create relation type	All relation types	Traverse tree
Weight or Importance (1, 2 or 3)	3	3	3	3	1	3
Presence (%)	70	80	100	90	70	60

The Create node and Create relation operations receive a score or percentage of inclusion for Dex system of 70%. This is due to the fact that they have to use an object belonging to *com.sparsity.dex.gdb.Value* class. This is a coupling to a technology specific class that could be easily avoid by providing higher level methods. Therefore, we can say that the way this functionality is supported is good but improvable.

For the Modify node and Modify relation operations the same can be said. The difference is that we give a little bit extra score to this modification ones because the system supports it without the need of removing the old object and creating a new one.

The Remove node and Remove relation operations represent examples of perfectly supported functionalities. They consist of just calling *com.sparsity.dex.gdb.Graph drop(…)* method with the node or relation id, respectively.

The Create relation type operation is supported in one of the best possible ways. In order to create a new relation type we need to call *com.sparsity.dex.gdb.Graph newEdgeType(…)* method, having checked previously that the type doesn't exist. However, as we consider that it could be improved a little bit, as we will see later on in Neo4j implementation, it receives a score of 90 out of 100.

For the operation for getting All relation types we consider a score of 70% because we have to iterate over all relation types identifiers. Then, we need to access the database as many times as relation types are, in order to retrieve the actual type using the identifier. Consequently, we can say that, from a performance point of view, this operation could considerably improve.

The Traverse tree operation receives a percentage of inclusion of 60% because it is fully supported but it could be better in terms of both performance and easiness. According to the implementation that we were able to find, what we need to do here is not a very efficient nor intuitive process. That process consists of iterating over the relation type identifiers to be able to search neighbors of each node of each type, so we end up having too many nested loops.

	Shortest path	All node's data	Get ancestors/ descendants	Number of descendants/ ancestors	Export tree
Weight or Importance (1, 2 or 3)	2	3	2	2	2
Presence (%)	100	80	80	80	100

The Shortest path operation receives the maximum punctuation, i.e. of 100 out of 100. The reason for that score is the fact that this operation is completely supported via a *SinglePairShortestPath* abstract class and *SimplePairShortestPathDijkstra* implementing class.

The Get all data operation has a score of 80% since it is supported but we need to use that Value class to get each one of the values. As we already penalized an operation implementation for this DBMS for the same reason, the penalty now is slightly below last time.

For Get descendants and Get ancestors operations, as also happened with a previous operation, we need to iterate over all edge types that exist in the system. The penalty assigned for that reason is less than before because it was already given.

The Get number of ancestors and Get number of descendants operations receive a rather good score, concretely 80%. This is due to the fact that it can be implemented like summing without the need of getting all values and counting them after that. I.e. the operation can be done without traversing the values twice, but only once. However, we note there is a small penalty because of having to iterate over all edge types in the system and then getting the neighbors of the starting node, for the edge type at each iteration.

For Exporting the whole family tree the score is 100 out of 100. Dex provides compatibility with multiple export types (i.e. GraphML, Graphviz and YGraphML) natively but none is useful. During the implementation process, we could observe that GraphML seems to be the most used and, then, a kind of "standard" for graph NoSQL DBMS. However, GraphML obtained with an export operation in Dex differs quite a lot in terms of format, compared with other DBMS used that support GraphML (i.e. Neo4j and OrientDB). However, Dex is compatible with Tinkerpop Blueprints as all these DBMS are too. Blueprints allows us to use some graph operations from many different systems and the export operation to GraphML format is one of them.

	Export tree's branch	**Import data**	**Query by field (age)**	**Get alive people**	**Get distinct surnames**	**Get all birth cities**
Weight or Importance (1, 2 or 3)	2	2	1	1	1	1
Presence (%)	100	100	80	80	90	90

For Export tree's branch operation the required support was found. The actions required to export only a branch with all systems compatible with Tinkerpop Blueprints are three. The first one is creating an empty *com.tinkerpop.blueprints.Graph* object. Next, we have to get all descendants (using *getAllDescendants(…)* operation from PersonDao) of the initial node for which we want to export a branch. And, finally, add all nodes, i.e. the initial node and all its descendants, to the empty graph.

Import data operation receives a score of 100% because it is one of those systems that support Tinkerpop Blueprints to perform graph operations. Blueprints allows for importing and exporting graphs (in this case the family tree) from or to, respectively, GraphML format.

The operation for querying by age receives a score of 80%, because it is supported, but it could be improved in at least two ways. The first one corresponds to the already cited problem with coupling to Value class. The second problem of this operation implementation is that we need to get the fields or attributes to be projected (i.e. people names) one by one, by iterating over the returned values. We cannot just indicate the fields that are interesting for us.

Get alive people operation has also the problem of having to iterate over the returned values to get the required results and the coupling to Value class. Get people surnames operation and Get all birth cities operation have also the "extra iteration" problem to get the fields. But this is the only improvable thing that was found for these implementations.

Now we present the Security table to finish with all the **Functionality** tables.

	Password authentication	**Backup of the database**	**Different privacy levels**
Weight (1, 2 or 3)	2	2	2
Presence (%)	0	100	100

As far as it could be found, Dex seems not to be compatible with Password authentication. This is why this database feature receives a score of 0%.

Dex supports the Backup feature by the following two methods in the following two different classes. The first one corresponds to the *com.sparsity.dex.gdb.Graph.backup(…)* method to get the "DatabaseFile.backup" file. After the backup we can restore from the previous file using *com.sparsity.dex.gdb.Dex.restore(…)*.

Finally, Dex supports both node and edge type definition. Then, Different privacy levels could be implemented by just creating, for example, some nodes/edges of type "private", some "public", etc.

The following table corresponds to the **Reliability** table, containing both comparison points corresponding to Maturity and Fault tolerance. Concretely all points are Maturity ones except for the last one (Ability to switch from one database to another) which is a Fault tolerance one.

	Operations still not available	**Use of low-level elements**	**Bugs caused by the DBMS**	**Ability to switch between databases**
Weight (-1, -2 or -3)	-3	-2	-2	-1
Presence (%)	0	30	0	0

For this table we must recall we aim at getting as least punctuation as possible as they punctuate in a negative way, i.e. using negative weights.

The Operations still not available point receives a score of 0 because we found 0 operations that were not supported by Dex. That means, all operations to be implemented were present.

For the Use of low-level elements we already indicated that this DBMS includes the use of a specific low-level element. This element consists of an object of the class *com.sparsity.dex.gdb.Value*. This is needed for assigning values in some operations, as it was said before when talking about each operation.

For the last two points, we would like to say that no Platform specific bugs were found when developing. Furthermore, no Problem for switching between different Dex databases was found.

The **Usability** table, containing Understandability (Time to read, understand and learn), Learnability (Adequate implementation time), Attractiveness (Different query languages and Visualization tools quality) and User transparency, is shown below.

	Time to read, and understand	Adequate implementation time	Different query languages	Visualization tools quality	User transparency
Weight (1, 2 or 3)	2	2	2	1	3
Presence (%)	80	70	60	0	100

For this DBMS, the Time to read, understand and learn the documentation was considered adequate. The documentation is complete and can be downloaded in pdf from the official web page.

The Implementation time was considered as adequate. The only thing that was found as improvable was the API. Many of the method names were considered not self-explanatory enough and this slows the development down.

The presence of Different kinds of query languages has a rather low score because only two query languages were found. The first one is Dex API itself, which is quite complete and permits most of the required operations. The second one is the already cited Tinkerpop Blueprints API. In our case, Blueprints is only used for the import and export operations, but we have to note that many others are supported too.

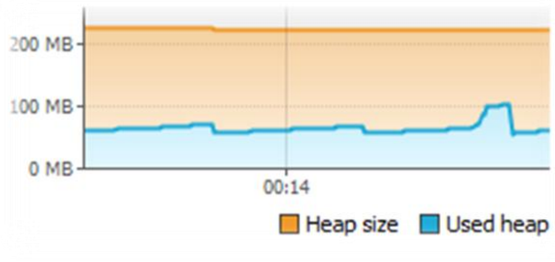
Finally, the Visualization tools point receives a score of 0 because none was found for this system.

The **Efficiency** table only contains the comparison points that can be observed below:

	Time behavior	**Resources utilization**
Weight (1, 2 or 3)	2	2
Presence (%)	100	100

For the Time behavior, Dex was considered as the fastest with a rather big difference from the second fastest one.

Dex is the system among the ones being compared that consumes the minimum quantity of Java heap memory. In this case, this can be seen as a consequence of the fact of being the fastest system too. Below we can see the VisualVM screenshot for the heap memory consumption while executing the system with Dex:



The **Maintainability** table is split according to its subparts as follows. There are two Maintainability tables where the first one corresponds to Analyzability and Changeability; then the second one contains Stability and Testability.

This is the first Maintainability table where all points, except for the last two, are Analyzability points.

	Storage size	Number of nodes, relations and indexes	Compaction operations	Data occupying as little space	Change relation types	Add new indices
Weight (1, 2 or 3)	2	2	2	2	3	2
Presence (%)	100	66.67	0	0	90	100

For Dex, an operation for getting information about the Storage size is included. This operation is called *getData()* and belongs to *com.sparsity.dex.gdb.DatabaseStatistics* class. The result of calling it is a long number indicating the size of the database in KBytes.

For getting the Number of edges and nodes, the two methods in *com.sparsity.dex.gdb.Graph* class were found; they are *countEdges()* and *countNodes()*, respectively. However, for getting the total number of indexes, no method was found. Consequently, we can say that 2 out of the 3 features were present, and this is why we have a score of 66.67 ≈ 2/3.

Regarding the Compaction operation, no support for it was found in this system, so a score of 0 is given to it. For the presence of mechanisms to guarantee that the Data is occupying as less space as possible, no possibility in this sense was found.

We consider that Changing the current relation types is possible. We simply need to use *createRelationType(…)* method in DexPersonDao class, created for this system. Another possibility to edit the system relation types is using PersonDao methods to manage relations (*createRelation(…)*, *modifyRelation(…)* and *removeRelation(…)*). However, these methods are only valid if, apart from editing relation types, we also want to change relations.

For Adding indexes feature we found the required support. The process consists of changing the attribute or field definition, i.e. transform it into an *Indexed* one. Then, to do that, we should call *indexAttribute(…)* method inside *com.sparsity.dex.gdb.Graph* class.

This is the second **Maintainability** table where the first two points are Stability ones and the resting ones are Testability capabilities.

	Lack of DBMS specific errors	**Quality of log data**	**Tools for debugging**	**Testing frameworks**
Weight (1, 2 or 3)	3	2	2	2
Presence (%)	100	100	70	70

As far as this DBMS was tested, no Errors specific of this technology were detected, so we consider it 100% free of DBMS specific errors.

With regard to Log data quality, there is a file auto-generated with the execution of the system using Dex database. This file is called dex.log and contains information about each error or warning that occurred during all executions, i.e. it is an incremental file. Moreover, using *com.sparsity.dex.gdb.LogLevel* enumeration, we can set the log level to *Off* or disable, *Fine* to log everything that happens, etc.

No specific Dex Debugging tools were found. However, this is not a problem since it is compatible with all debugging tools that the Java programming language is, such as the Eclipse IDE debugger.

For the Testing tools or frameworks, it is the same situation as for the debugging tools. No specific Dex tools were found, but all testing systems compatible with Java are suitable to be used, e.g. JUnit.

To end up with all the tables, we have the **Portability** one containing both Install-ability and Portability compliance features. In the following Portability table we can see the first three points as Install-ability ones and the resting ones related with the Portability compliance:

	Time for downloading	**Time for installing**	**Time to get it running**	**Format compatibility**	**Fast import and export**
Weight (1, 2 or 3)	2	2	2	3	2
Presence (%)	70	80	90	100	70

For this DBMS, the Time for downloading it was adequate. The only things that were required to be specified were, mainly, the database size that we require and the purpose of its usage (commercial, research, etc.). This previous step to download the DBMS could be thought as an extra time to start installing it.

The Time for installing the DBMS was fairly short. The only needed action was using the .jar file available from the official web page. This .jar was able to be used with Eclipse IDE, just adding it as an external library. We have to note, however, that adding the dependency using m2e Maven plugin for Eclipse was not possible, so the installation was slightly longer than for other DBMS.

The Time to get the system running was adequate. We cannot mention any special important difficulty found during the first steps using the DBMS.

Regarding the Format compatibility, Dex was able to show a good support. Apart from the already cited GraphML compatibility via Tinkerpop Blueprints, Dex also supports exports to CSV format.

Considering that the Export and import operations are generally not very fast, we would say that this is not the exception. The time for these operations in this case was not too long, but we have to keep in mind the size of the sample with which we performed the tests is not too big.

9.1.2 HyperGraphDB

	Create node/ relation	Modify node/ relation	Remove node/ relation	Create relation type	All relation types	Trave rse tree
Weight or Importance (1, 2 or 3)	3	3	3	3	1	3
Presence (%)	100	80	80	80	70	70

For the operation of Creating a node or a relation, we consider a score of 100% because it is completely supported. Furthermore, creating a node we can specify whatever kind of node created by us, i.e. it is like the domain layer is contained in the persistence one.

This has many advantages like, for example, that we avoid converting objects retrieved from the database into domain ones. Another important advantage is that we know exactly what we are storing at each moment and we can extend or remove the attributes stored whenever we want it by simply changing the class.

For creating relations we also have as an advantage the possibility of creating new kinds of relations by simply creating a class that implements org.hypergraphdb.HGLink interface.

Regarding the operations for Modify a node or a relation, we split it into two parts to analyze both node and relation support. For modifying a node we consider the same advantages seen for the create operation. In this case we can use the node class setter to change an object values. Consequently, it is fully supported and thus it receives a punctuation of 100%.

For the modify relation operation we also have the advantage of using a class implementing the HGlink interface, which can be adapted to our concrete needs. However, for this operation we consider a score of 60% because of the presence of an important problem. This problem is the fact that no support for this operation without removing the old relation was found. This means, in order to modify an existing relation we need to remove the old one and create a new one.

Finally, the punctuation of 80% that we see for this operation as a general one is the result of the average of both sub-operations. Then, 80% could be obtained by summing 100% for the node operation and 60% for the relation operation and then dividing this result by 2.

For the <u>Remove node and relation</u> operations, we consider a punctuation of 80% because they are completely supported. However, we include a penalty for removing a node because a problem with this operation was found that didn't allow us to perform it correctly. As this error seems to be a platform specific error, it will be explained in more detail in the adequate point, i.e. the one about maturity and, concretely, when talking about bugs.

For <u>Creating relation types</u> we assign a score of 80 out of 100 because it can be done by creating a new relation. For example, if we use the HGValueLink class, the value of the relation or relation types is an element of type Object that is given as a first parameter of the constructor. However, a small penalty is assigned to this operation's score since it doesn't support creation of relation types without creating relations.

The operation for getting <u>All relation types</u> receives a score of 70% because it is supported but the implementation that was achieved is not very efficient. This implementation consists of getting all identifiers for both nodes and relations and then filtering them.

The operation for Getting the family tree has the same problem that the previous one also has. In this case, what our implementation does is filtering to obtain just relations and then getting the relation extreme points.

	Shortest path	**All node’s data**	**Get ancestors/ descendants**	**Number of descendants/ ancestors**	**Export tree**
Weight or Importance (1, 2 or 3)	2	3	2	2	2
Presence (%)	0	100	80	90	0

No support was found for the operation of getting the Shortest path between two nodes. There is a method provided by the DBMS API that is called *dijkstra(…)* and belongs to *org.hypergraphdb.algorithms.GraphClassics* class. But the problem with this method is that it only obtains the distance between the two nodes as a numerical value and what we want is the list of nodes and edges in the path.

The operation for getting All node’s data is fully supported by using getter methods of the node class.

The Get ancestors and get descendants operations are supported in a more or less good way. The only problems or improvements that were detected are two. The first one is the lack of support for setting a maximum level at which the ancestors/descendants search should stop. The second one consists of an implementation improvement in which we get all relations from a given node and so on recursively, but we need to filter inside the loop to get only descendants or ancestors, respectively.

The operations for Getting the number of ancestors and descendants have one of the problems that the previous ones had. This problem consists of the retrieval of all relations from a given node and all of them recursively, but without a filter to get only ancestors or descendants. We have to note that the problem observed before (of not supporting a maximum level) is not present now as we don't require it for getting the total number of ancestors/descendants.

Finally, no support for the Export operation was found using this DBMS. No way for implementing this operation was found looking at both native API and Tinkerpop Blueprints. In fact, the last one is not even compatible with HyperGraphDB.

	Export tree's branch	**Import data**	**Query by field (age)**	**Get alive people**	**Get distinct surnames**	**Get all birth cities**
Weight or Importance (1, 2 or 3)	2	2	1	1	1	1
Presence (%)	0	0	100	100	90	90

The Export tree's branch operation is not supported as explained in the last operation in the last table. For the Import operation we can say the same since this DBMS is not compatible with Tinkerpop Blueprints, which is the system that provides support for GraphML format import.

The Query for people older than some age operation was fully supported by using *hg.lt(…)* (lt = less than) operator to query objects of the node class using *hg.getAll(…)* method. The same was done to implement the Query for alive people with the only difference that the operator used in this case was *hg.eq(…)* with "alive" value as a parameter.

The queries for Getting people surnames and Getting people birth places receive a score of 90% because they are fully supported in a similar way as the last ones but they could be improved. The way that was thought to improve them is a support that was not found for getting just the attributes or fields that are meant to be projected.

Now we present the Security table to finish with all the **Functionality** tables.

	Password authentication	**Backup of the database**	**Different privacy levels**
Weight (1, 2 or 3)	2	2	2
Presence (%)	0	0	100

For the Password authentication feature, no support was found using this DBMS. Also, no support for performing a Backup of the database was found too. Finally, for the compatibility with Different privacy levels, we could implement then system using different node and relation types which would be suitable to support this feature.

The following table corresponds to the **Reliability** table, containing both comparison points corresponding to Maturity and Fault tolerance. Concretely all points are Maturity ones except for the last one (Ability to switch from one database to another) which is a Fault tolerance one.

	Operations still not available	**Use of low-level elements**	**Bugs caused by the DBMS**	**Ability to switch between databases**
Weight (-1, -2 or -3)	-3	-2	-2	-1
Presence (%)	20	0	30	0

Among all the operations that are supposed to be supported by all systems, approximately the 20% are Operations still not available for this concrete DBMS. These non-supported operations are the import and export features (including both exporting the whole tree and only one branch) and the shortest path.

Regarding the Use of low-level elements in the implementation, no example of this fact was found while developing. The level at which the development was done was adequate to what we previously expected using Java programming language.

The Bug that was found during the development phase and that was mentioned before in one of the operations' explanation is with the feature to remove a node. The problem with this operation is that the operation seems to be done correctly for removing the internal DBMS identifier for the node but, however, the node object still exists and is available to be retrieved. This issue seemed to be occasioned by the DBMS itself and the workaround we used to solve it was adding a boolean attribute in the node class called removed.

Finally, with respect to the support for Being able to switch between different databases, no problem that would make this feature impossible was found.

The **Usability** table, containing Understandability (Time to read, understand and learn), Learnability (Adequate implementation time), Attractiveness (Different query languages and Visualization tools quality) and User transparency, is shown below.

	Time to read, and understand	Adequate implementation time	Different query languages	Visualization tools quality	User transparency
Weight (1, 2 or 3)	2	2	2	1	3
Presence (%)	50	60	0	70	100

The Time to read and understand HyperGraphDB documentation doesn't receive a good mark because there is only few documentation and only a few examples. Regarding pdf documentation, only a slides were found [HYPER_SLIDES], but no official documentation to be downloaded.

The Implementation time was affected by the lack of documentation and support for many operations. Also, a difficulty for fitting the domain in the storage system was found. This last fact is due to a not adequacy of the domain with hypergraphs, i.e. graphs containing hyperedgess; an interesting feature that we finally could not take advantage of.

With regard to the Support for different kinds of query languages, HyperGraphDB is not compatible with any query language apart from the native API itself.

For the Visualization tools, we have to note that this system includes some classes belonging to a package called "viewer" that allow for creating a system for visualizing HyperGraphDB graphs. This is a good option to have a graphical representation of our graph, in this case a family tree. However, this support for visualization could be improved by having a system already created for having this graphical representation of our graph.

When developing using this graph DBMS, we didn't note any case in which the User transparency would be violated. It doesn't require for having a specific software system, for example, in the client side.

The **Efficiency** table only contains the comparison points that can be observed below:

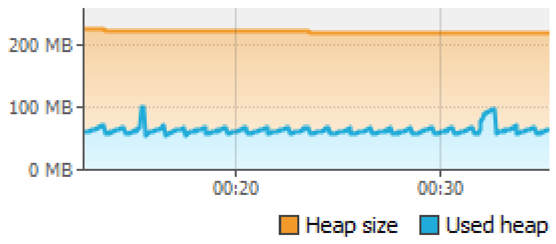
	Time behavior	**Resources utilization**
Weight (1, 2 or 3)	2	2
Presence (%)	80	80

HyperGraphDB receives a score of 80% for the Time behavior feature as it is the third fastest DBMS among the ones that are compared. This can be seen as almost the worst but we don't consider it like this because HyperGraphDB obtained time is rather closed to the second best one and really far from the forth one.

Here we can see an example of why we didn't consider a disadvantage or advantage the fact of measuring performance for different systems with the same operations, some of them not supported. Here we can note that HyperGraphDB is a system with many unsupported operations and it is not the fastest one.

For the Resources utilization, we have to say that in the case of HyperGraphDB it was very low, as we can see in the following VisualVM screenshot

(HyperGraphDB value is the one on the right side). However we know HyperGraphDB's consumption must be greater than Dex one because HyperGraphDB is slower. In fact we can observe this too in this picture since the value on the left side corresponds to Dex.



The **Maintainability** table is split according to its subparts as follows. There are two Maintainability tables where the first one corresponds to Analyzability and Changeability; then the second one contains Stability and Testability.

This is the first Maintainability table where all points, except for the last two, are Analyzability points.

	Storage size	Number of nodes, relations and indexes	Compaction operations	Data occupying as little space	Change relation types	Add new indices
Weight (1, 2 or 3)	2	2	2	2	3	2
Presence (%)	0	90	40	0	80	100

No mechanism for getting the Total current storage size of the database was found. This is the reason why this operation receives a score of 0%.

For both Getting the number of nodes and Getting the number of relations, we can use *count(…)* method from *org.hypergraphdb.HyperGraph* class.

However, for Getting the number of indexes there is no direct support. We should get all indexes either by type or by value using, respectively, *getIndexByType(…)* or *getIndexByValue(…)*, both from *org.hypergraphdb.HGIndexManager* class. Then with this result, we could count the number of indexes.

There was no support found for a Data compaction operation. However, there exists an operation for executing maintenance tasks, but the API doesn't specify if it fulfills our requirements. It is a method called *runMaintenance()* and is part of *org.hypergraphdb.HyperGraph* class.

Furthermore, no operation for guaranteeing that the Data is occupying as few space as possible was found. Then we consider this feature as a not supported one (at least in the current version).

In order to Change the existing relation types we need to use the relation methods in PersonDao. These relation methods are: *createRelation(…)*, *modifyRelation(…) and removeRelation(…)*. No way to change the existing relations types without changing a relation was found.

Finally, the Addition of new indexes is fully supported in HyperGraphDB by means of *HGIndex* interface and, concretely, the *addEntry(…)* method.

This is the second **Maintainability** table where the first two points are about Stability and the resting ones are Testability capabilities.

	Lack of DBMS specific errors	Quality of log data	Tools for debugging	Testing frameworks
Weight (1, 2 or 3)	3	2	2	2
Presence (%)	70	50	70	70

With regard to DBMS specific errors, we can say that the system is 70% free of them. The only error apparently caused by the DBMS itself is the already cited problem with the implementation of the *removePerson(…)* method in *HyperGraphDbPersonDao* class.

For the Log data we only found a reference to them in *HGLogger* class. This class contains methods for *exception(…)*, *warning(…)*, etc. However, no way of automatically keeping a file with this log data was found. This is the reason why this feature receives a low score for this concrete system.

No specific HyperGraphDB Tools for debugging nor Testing frameworks were found. However, this DBMS supports the use of all both debugging tools and testing frameworks compatible with the Java programming language. Consequently, this last fact would be the required response to our needs.

To end up with all the tables, we have the **Portability** one containing both Install-ability and Portability compliance features. In the following Portability table we can see the first three points as Install-ability ones and the resting ones related with the Portability compliance:

	Time for downloading	Time for installing	Time to get it running	Format compatibility	Fast import and export
Weight (1, 2 or 3)	2	2	2	3	2
Presence (%)	100	100	90	0	0

The Time for downloading HyperGraphDB was really fast. Both the .jar file and Maven dependency can be obtained from the official web page.

For installing, it was fast too because of having both the .jar file option and Maven dependency.

For the Format compatibility capability, we were not able to find any format with which this system would be compatible for neither import nor export operations.

Finally, as no support for import nor export operations was found, we cannot, of course, consider that the required Fast import and export capability is fulfill.

9.1.3 Neo4j

	Create node/ relation	Modify node/ relation	Remove node/ relation	Create relation type	All relation types	Traverse tree
Weight or Importance (1, 2 or 3)	3	3	3	3	1	3
Presence (%)	90	80	100	100	100	90

The Create node and Create relation operations are completely supported, so they receive a high punctuation. The reason why they don't receive the highest possible mark is that they could be improved by allowing, as HyperGraphDB does, for using different kinds of node and relation objects.

For the Modify node operation we consider a score of 100% since it is fully supported and we don't require for removing the old node and creating a new one, we can just modify the existing one. For the Modify relation operation we consider a punctuation of 60% since the only possible way of modifying it that was found consists of removing the old relation and creating a new one.

Both Remove node and Remove relation operations are fully supported using Neo4j DBMS. Moreover, we couldn't find any problem nor improvement for the obtained implementation with this system.

For Creating new relation types, Neo4j provides an interface called *org.neo4j.graphdb.RelationshipType* that we can use to implement our own relation types. Then, the result would be an enumeration in which we can add as many values as relation types we want our system to have. It is a good system for many reasons. First of all, the use of an enumeration has the advantage, compared with Strings, of preventing from possible typos. Furthermore, it avoids having to guess all relation types that we have or having to ask the DBMS each time we want to know it. Finally, another positive point for this system is the possibility of managing relation types independently of relation management.

For Getting all relation types we use an implementation for which we couldn't find any problem nor improvement. It consists of calling *getAllRelationshipTypes()* method inside *GlobalGraphOperations* class and iterating over the *RelationshipType* objects to get their name value.

Finally, the Get family tree operation is fully supported by retrieving all relation types, iterating over them and, for each relation, getting the start and end nodes, a part from the relation type name.

	Shortest path	**All node's data**	**Get ancestors/ descendants**	**Number of descendants/ ancestors**	**Export tree**
Weight or Importance (1, 2 or 3)	2	3	2	2	2
Presence (%)	100	100	100	80	100

The Shortest path operation is fully supported with Neo4j by means of the *shortestPath(…)* method inside *org.neo4j.graphalgo.GraphAlgoFactory* class. It allows us to provide all necessary conditions before iterating without requiring for filtering anything inside the loop.

The Get all data operation is provided by the getPropertyKeys() method inside *org.neo4j.graphdb.Node* class. It allows us to get all properties, fields or stored values for a given node by only one query.

The operation for Getting all ancestors and the one for Getting all descendants of a given node or person is completely supported. The way of implementing them is using *org.neo4j.kernel.Traversal* class and then applying different methods to get an *org.neo4j.graphdb.traversal.TraversalDescription* object.

Then we use this object to call *traverse(…)* method and *nodes()* using the resultant object. Finally, the last object is an *Iterable<Node>* which we can use to iterate over all descendants to get the required properties, such as their names.

The main advantages of this implementation is the fact of using a clear API with many different options to apply over the Traversal and, especially, the fact of avoiding filtering inside the final loop. We start iterating with only the needed information and this reduces the total number of iterations and thus increases performance.

For the Number of descendants and the Number of ancestors operations, we have almost the same implementation than for getting the descendants, respectively. The only difference is that, instead of keeping some ancestor/descendant properties, what we do is counting them. However, the clear improvement that was detected for this implementation is the lack of an operation that just returns the count or sum of them and not the whole *org.neo4j.graphdb.Node* objects.

Finally, for the Export operation, it is the same case as for Dex. As Neo4j is also compatible with Tinkerpop Blueprints, the export can be done in an adequate way using GraphML format.

	Export tree's branch	**Import data**	**Query by field (age)**	**Get alive people**	**Get distinct surnames**	**Get all birth cities**
Weight or Importance (1, 2 or 3)	2	2	1	1	1	1
Presence (%)	100	100	90	90	90	90

The Export tree's branch operation is compatible with this system as it can be implemented using Blueprints in the way already described for Dex DBMS in the corresponding section.

Importing data from a GraphML file is the same case as the export operations. They all three can be implemented in an adequate way since the current analyzed system is compatible with Tinkerpop Blueprints.

Finally, for all Query operations in general, we observe an adequate support but that could be improved for all of them in the same way. It is a small improvement that consists of supporting for only getting the required field/s or properties which is a basic feature that we expect and have using SQL language but that we couldn't find here.

Now we present the Security table to finish with all the **Functionality** tables.

	Password authentication	**Backup of the database**	**Different privacy levels**
Weight (1, 2 or 3)	2	2	2
Presence (%)	60	20	100

For the Password authentication feature we have to say that it is not completely supported yet, as it could be found. What exists right now is a support for protecting the web server with a password by using a proxy, as can be seen in [SERVER_INSTALL] or use an extension for authentication [AUTHENTICATION_EXT]. Apart from that, there is another security feature which is quite useful and consists of allowing or banning access to certain servers [SECURITY_SERVER].

To be able to perform the Database backup operation, we would need to use an Enterprise version of the system. However, as we are using the Community version, we are not allowed to use this feature. For this reason, it receives a very low punctuation.

Finally, to have Different privacy levels, we didn't find any problem that could make this requirement impossible to be done. We would need to create different node types, which is an available feature when using Neo4j.

The following table corresponds to the **Reliability** table, containing both comparison points corresponding to Maturity and Fault tolerance. Concretely all points are Maturity ones except for the last one (Ability to switch from one database to another) which is a Fault tolerance one.

	Operations still not available	Use of low-level elements	Bugs caused by the DBMS	Ability to switch between databases
Weight (-1, -2 or -3)	-3	-2	-2	-1
Presence (%)	0	0	20	0

With regard to Operations still not available for this system, we have to say that none of them were found. All operations that were required to be implemented were finally achieved.

For the Use of low-level elements, we didn't find any case of it, so we consider a score of 0% as the minimum one in a positive sense (for negative points, a low score ends up being positive).

For the Bugs occasioned by the platform or system itself, we only found a small one that was possibly because of Neo4j or maybe because of the lack of experience using it. The problem was an error that appeared after changing some data elements (e.g. node fields, etc.) that make the system not work until we create a new empty database and start working with the new one. We are aware this may be a DBMS maturity problem, but it can also be a misunderstanding of the system.

Finally, for the Ability to switch between different databases, we didn't find any problem that makes this requirement impossible to be provided. We would just need to add a condition to use one database in some cases and another in other cases, for example.

The **Usability** table, containing Understandability (Time to read, understand and learn), Learnability (Adequate implementation time), Attractiveness

(Different query languages and Visualization tools quality) and User transparency, is shown below.

	Time to read, and understand	Adequate implementation time	Different query languages	Visualization tools quality	User transparency
Weight (1, 2 or 3)	2	2	2	1	3
Presence (%)	100	100	100	100	100

About the Time to read, understand and learn the documentation, we have to say it was really short due to the inclusion of a lot of examples, as well as pdf documentation and many extensive documentation sections. Therefore, as the documentation was really complete, this phase was seen as easier as if we have had worse documentation.

For the Implementation time, we consider that it was also short. Everything we needed was supported and Neo4j API is very complete and clear. Another helpful fact is that it includes support for many different ways of doing operations (query languages, classes, extensions such as Lucene indexes, etc.). Then, this last fact makes the development phase easier.

For the support of Different kinds of query languages, Neo4j provides the maximum support among all DBMS compared here. Apart from Neo4j native API, we can query by using Gremlin language, which is a graph traversal scripting language [GREMLIN]. Furthermore, developing with Neo4j we can also use Cypher graph query language [CYPHER]. Finally, we can also use the already cited Tinkerpop Blueprints driver for graph operations such as importing and exporting from/to GraphML format, etc.

With regard to the Visualization tools, Neo4j Web UI provides a visualization environment to display the nodes, properties and relationships added in the system throughout the time. The server can be accessed by executing the Neo4j.bat file in Windows or the neo4j one in Linux and Mac, all of them in the bin directory of the corresponding version. Then, after executing it with the “start” option, we can access the UI in a web browser by typing http://localhost:7474/. It is a really good tool for both visualization and also query or statement execution.

A part from the Web UI, for visualizing Neo4j data we can also use Gephi, which is an open source visualization software that can be used together with Neo4j [GEPHI].

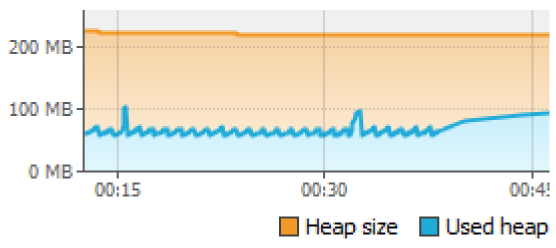
Finally, for the User transparency, we didn’t find anything that would compromise it. No need for having, for example, a specific software or operating system installed was found.

The **Efficiency** table only contains the comparison points that can be observed below:

	Time behavior	**Resources utilization**
Weight (1, 2 or 3)	2	2
Presence (%)	40	50

During the execution of the obtained implementations for all systems, Neo4j was seen as the slowest one. Comparing the time that each one was taking to execute the same list of operations, we observed Neo4j’s response time was approximately 3 times slower than the second slowest system. And it was 10 times slower than the fastest among all four systems.

Neo4j is the DBMS that consumed the maximum quantity of Java heap memory. We can observe in the following screenshot how it is arriving to its maximum value (100 MB) and it is maintained for quite long time. We can see Neo4j's value on the right side of the screen and we can also see how it compares with Dex (the first value starting from the left) and HyperGraphDB (the one in the middle).



The **Maintainability** table is split according to its subparts as follows. There are two Maintainability tables, where the first one corresponds to <u>Analyzability</u> and <u>Changeability</u>; and the second one contains <u>Stability</u> and <u>Testability</u>.

This is the first Maintainability table where all points, except for the last two, are Analyzability points.

	Storage size	**Number of nodes, relations and indexes**	**Compaction operations**	**Data occupying as little space**	**Change relation types**	**Add new indices**
Weight (1, 2 or 3)	2	2	2	2	3	2

Presence (%)	0	70	0	0	100	100

No way of Getting the total storage size of the system was found. We are aware this could be because it doesn’t exist, but it can also be because of not having found it although it exists.

For the Number of nodes and the Number of relations we found the methods *getAllNodes()* and *getAllRelationships()*, both inside *GlobalGraphOperations* class would allow us to obtain them by iterating over the Iterable returned by calling the methods and increasing a counter. The operation for getting the Number of indexes would be supported by calling both *nodeIndexNames()* and *relationshipIndexNames()* both in *IndexManager* class. Then we would need to get the *length* of both String[] returned with the method calls and, finally sum both values.

We didn’t find information about Compaction operations nor mechanisms to guarantee that the Data is occupying as few space as possible.

The way of Changing relation types is quite simple and is done through an Enum implementing Neo4j interface RelationshipType.

Finally, for Adding new indexes and index entries we need to use org.neo4j.graphdb.Index<T extends PropertyContainer> interface. Basically, we can have both Relationship and Node objects indexed.

This is the second **Maintainability** table where the first two points are Stability ones and the resting ones are Testability capabilities.

	Lack of DBMS specific errors	Quality of log data	Tools for debugging	Test frameworks
Weight (1, 2 or 3)	3	2	2	2
Presence (%)	100	90	70	80

For Neo4j we didn't detect any DBMS specific error. Furthermore, the Quality of the log data seemed to be right. The system allows for keeping in a "messages.log" file everything that occurred in the system during the execution.

For the Debugging tools we can say the already cited Gephi project is also suitable for Java class debugging. Apart from that, Neo4j also provides options for remote debugging sessions using Neo4j server. More information on that can be found in [REMOTE_DEBUG].

We didn't find any Neo4j specific Tools for testing. However, as also happens with the other systems, it is compatible with all test frameworks that the Java programming language is. Additionally, Neo4j provides a small tutorial about how to use JUnit or Hamcrest in the official web page [BASIC_UNIT_TEST].

To end up with all the tables, we have the **Portability** one containing both Install-ability and Portability compliance features. In the following Portability table we can see the first three points as Install-ability ones and the resting ones related with the Portability compliance:

	Time for downloading	**Time for installing**	**Time to get it running**	**Format compatibility**	**Fast import and export**
Weight (1, 2 or 3)	2	2	2	3	2
Presence (%)	100	100	100	90	70

Using Neo4j, the time for both Downloading and Installing it was really short. For the download part we didn't have to register and we only needed to get the package with all files. For installing it we had both .jar file version and Maven dependency options, so it was quite easy. The same can be said for the Time to get the system running, especially because of the simplicity and clearness of the API.

For the Format compatibility, we can say Neo4j is compatible with GraphML format to perform both import and export operations through Tinkerpop Blueprints driver. No other compatible format was found, but this one ends up being more than enough as it is the standard for graph NoSQL DBMS.

For the Fast import and export operations point we have to say that the import operation was quite slow for the quantity of nodes to be retrieved and the export one was quite fast.

9.1.4 OrientDB

	Create node/ relation	Modify node/ relation	Remove node/ relation	Create relation type	All relation types	Traverse tree
Weight or Importance (1, 2 or 3)	3	3	3	3	1	3
Presence (%)	90	60	100	80	100	100

For the Create node and the Create relation operations we consider a punctuation of 90%, which is a high value, since we consider they are fully supported.

For both Modify node and Modify relation operations we consider a score of 60% which is not very high. This is due to the fact that both are supported but by removing the old node and creating a new one. No option for modifying the existing node was found.

Both Remove operations are fully supported by means of an adequate implementation, so they receive a 100% score.

For Creating relation types, the only way that was found was by using the methods to manage relations (create, modify and remove relation operations). No way of managing relation types in an independent way than relations was found. This is not a big problem, but the fact of having this feature allows us for a more flexibility because of having more options for developing.

The operation for getting All relation types is considered as 100% supported as it can be implemented in a single query and without having to filter results inside the loop that fills the result object. Concretely, in this implementation we used an *OSQLSynchQuery<ODocument>* object that permits creating a query in an OrientDB SQL language. This language is like standard SQL but with some language extensions for working with graphs. In this case the query that was used is *"select distinct(label) as label from E"*, where E represents all the edges in the system.

Finally, the Traverse tree operation is also fully supported. For this operation we use *browseEdges()* method inside *OGraphDatabase* class. Then, iterating over the results which is an *Iterable<ODocument>* we can get relation labels or types and the relation extreme nodes.

	Shortest path	**All node's data**	**Get ancestors/ descendants**	**Number of descendants/ ancestors**	**Export tree**
Weight or Importance (1, 2 or 3)	2	3	2	2	2
Presence (%)	0	100	80	80	100

In OrientDB the Shortest Path operation is not supported at all. Then, it receives a punctuation of 0%.

The Get all data operation is completely supported by using the *field()* method applied to an *ODocument* object. This *ODocument* can be obtained by using the Node identifier.

For both Get all ancestors and Get all descendants operations, we assign a high value because both are supported, including the maximum level parameter. However, we include a small penalty due to a performance improvement due to the fact that the current implementation consists of getting all nodes and edges without filtering.

For the Number of Ancestors and Descendants operations, we achieved an implementation that is quite similar to the get all ancestors/descendants operations, respectively. The only difference is that for getting the number or count what we do is increasing a counter, instead of adding results to a result Set.

The Export tree operation is fully supported by using Tinkerpop Blueprints in the same way as Neo4j and Dex. So, it receives a score of 100%.

	Export tree's branch	**Import data**	**Query by field (age)**	**Get alive people**	**Get distinct surnames**	**Get all birth cities**
Weight or Importance (1, 2 or 3)	2	2	1	1	1	1
Presence (%)	100	100	90	90	90	90

Both Export tree branch and Import data operations are fully supported by means of the Tinkerpop Blueprints driver for performing graph operations, as it was already explained for Dex and Neo4j.

For the operations for Getting all elements fulfilling a certain query we have the same situation that was observed before. They are 100% supported but they receive a small penalty for having to retrieve the whole Node and then filtering to get the required field/s. This is the reason why they receive a punctuation of 90%.

Now we present the Security table to finish with all the **Functionality** tables.

	Password authentication	**Backup of the database**	**Different privacy levels**
Weight (1, 2 or 3)	2	2	2
Presence (%)	100	90	100

OrientDB has 100% Support for authentication by means of a name and password. This is done by indicating it as parameters to the *open(…)* method applied to a *OGraphDatabase* object.

For the Backup of the database, what is recommended in OrientDB Google Groups is just copying the whole database folder. We can see the discussion in the following link: [ORIENT_BACK]. We can also perform an automatic backup as it is explained in [ORIENT_AUTO_BACK].

Finally, for the Different privacy levels, we didn't find anything that could make this operation incompatible. What we could do is simply create different node types or, what is the same, with different labels. In the current implementation we only have nodes with label "PERSON", but we could have, for example, "PUBLIC" and "PRIVATE" nodes.

The following table corresponds to the **Reliability** table, containing both comparison points corresponding to Maturity and Fault tolerance. Concretely all points are Maturity ones except for the last one (Ability to switch from one database to another) which is a Fault tolerance one.

	Operations still not available	Use of low-level elements	Bugs caused by the DBMS	Ability to switch between databases
Weight (-1, -2 or -3)	-3	-2	-2	-1
Presence (%)	5	0	5	0

For OrientDB only one operation out of 22 was found as Not available operations. This operation was the one for getting the shortest path between two nodes to know the relationship between them. Then, 1 out of 22 represents approximately 5%.

No Use of low-level elements was found for this DBMS, so we consider a "negative" score of 0% for this point.

The only Bug due to the DBMS itself was the one that caused the not availability of the shortest path operation. The method for getting this exists in the documentation but the operation is not working. It was expected to be supported by using the SQL language for OrientDB with extensions for working with graphs. The query for the shortest path would be something like *"select flatten(shortestpath("+ firstNode + ", " + secondNode + ").out)"* but this didn't work.

For Being able to switch between databases with OrientDB, we didn't find any limitation. We would simply need to add a condition in order to use one database for some cases and another/s for other cases.

The **Usability** table, containing Understandability (Time to read, understand and learn), Learnability (Adequate implementation time), Attractiveness (Different query languages and Visualization tools quality) and User transparency, is shown below.

	Time to read, and understand	Adequate implementation time	Different query languages	Visualization tools quality	User transparency
Weight (1, 2 or 3)	2	2	2	1	3
Presence (%)	80	70	90	80	100

The Time to read, understand the documentation was quite short. Although there is not too much documentation and only some slides as pdf documentation, it was quite easy to understand.

The Implementation time was considered as adequate. The only special problem that made implementation slower was when the problem in the operation for getting the shortest path was detected. Many different queries were tried and none of them worked, also many forums and specialized pages were consulted.

OrientDB has support for many Different kinds of query languages. First of all, developing with the graph version of OrientDB we can use the native API or GraphDatabaseRaw with classes such as *OGraphDatabase*, etc. Also, we have support for OrientDB's SQL extended language by using an *OSQLSynchQuery<ODocument>* object, indicating the query as a String parameter. Finally, as we said before, OrientDB is compatible with Tinkerpop Blueprints for performing graph operations such as import and export database tasks.

Regarding the Visualization tools for OrientDB, a visualizer called OrientDB Studio is provided. The first step to get it is installing the OrientDB server that uses the OrientDB HTTP REST protocol and is able to respond REST JSON requests. After having the server, we can open it to visualize by typing http://localhost:2480/ in a web browser as explained in [ORIENT_STUDIO].

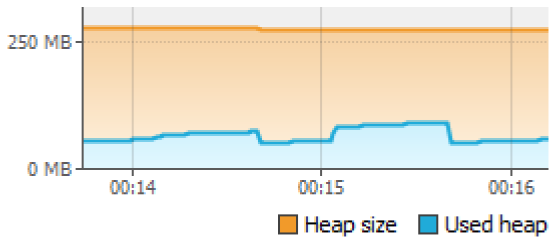
For the User transparency requirement, we didn't find any limitation or situation for which this wouldn't be possible.

The **Efficiency** table only contains the comparison points that can be observed below:

	Time behavior	**Resources utilization**
Weight (1, 2 or 3)	2	2
Presence (%)	90	80

OrientDB response time was seen as the second shortest after Dex which is the fastest. Then, the punctuation for this system is 90 out of 100.

OrientDB resources utilization seems to be quite similar to HyperGraphDB one or even better. We can see its graphical representation in the following VisualVM screenshot and, concretely, the value on the right. We have to keep in mind that here the picture is more expanded than it was before. We can appreciate it uses approximately 100MB or a little bit less for less than a minute. That is, as we said, quite similar to HyperGraphDB.



The **Maintainability** table is split according to its subparts as follows. There are two Maintainability tables where the first one corresponds to Analyzability and Changeability; then the second one contains Stability and Testability.

This is the first Maintainability table where all points, except for the last two, are Analyzability points.

	Storage size	Number of nodes, relations and indexes	Compaction operations	Data occupying as little space	Change relation types	Add new indices
Weight (1, 2 or 3)	2	2	2	2	3	2
Presence (%)	100	95	70	10	80	100

For Getting the storage size in OrientDB we simply need to call *getStorage()* method applied to an *OGraphDatabase* object. Then, with the *OStorage* object obtained we call *getSize()* method.

For the Number of nodes and Number of edges, we found 100% support by calling *countVertexes()* and *countEdges()* methods, respectively. For the Number of indexes, we found that it was supported but we needed to get the *OIndex* objects and then use *size()* method, because no operation for directly getting the size was found. This last case represents a semantic improvement although not a performance one.

For the Compaction operation, only the following information was found. They said some time ago that it was in general not required because OrientDB automatically recycles holes. However, a defragmentation operation was supported but in alpha version. The source of this information can be found in [ORIENTDB_COMPACT].

For the possibility of having Data occupying as little space as possible, we didn't find it exactly. The only thing that was found was an operation for indicating the "data segment strategy" to be used. This can be done by calling the method *setDataSegmentStrategy(…)* applied to an *OGraphDatabase* object. This allows for indicating, for example, an *ODefaultDataSegmentStrategy* object with which we can call *assignDataSegmentId(…)* method. In this way, we can indicate in which data segment we want the new record to be located.

For <u>Changing the relation types</u> we can only do it by changing the relations with the already implemented and explained methods to manage relations. These methods are *createRelation(…)*, *modifyRelation(…)* and *removeRelation(…)*. No way of changing relation types without changing relations was found.

For <u>Adding indexes and index entries</u> we found full support by using the index manager that can be obtained by calling *db.getMetadata().getIndexManager()* where *db* is an OGraphDatabase object. With this manager we can call *createIndex(…)* method to create a new one or *getIndex("name")* to get an existing one on field "name". Then, in the last case we can call *put(…)* method to add new entries to the existing index.

This is the second **Maintainability** table, where the first two points are <u>Stability</u> ones and the resting ones are <u>Testability</u> capabilities.

	Lack of DBMS specific errors	**Quality of log data**	**Tools for debugging**	**Test frameworks**
Weight (1, 2 or 3)	3	2	2	2

Presence (%)	90	70	70	70

The only OrientDB specific error that was found is the one about the shortest path operation. Then, it receives a punctuation of 90%, i.e. we include a penalty of 10% because of this error.

OrientDB uses *java.util.logging.Level* java class to specify the logging level that is required (i.e. WARNING, FINE, INFO, etc.). Then, Log data can be configured in a configuration file using Java syntax. However, there is not exactly an OrientDB specific mechanism for this task.

For the Tools for debugging, OrientDB doesn't provide specific tools for doing it. It is compatible with Gephi, which is a visualization tool that also allows for debugging. Besides, in the documentation an explanation on how to configure it together with the Integrated Development Environment (IDE) is provided [DEBUG_SERVER].

For the Testing frameworks, as already happened for other DBMS, OrientDB doesn't provide its own, but it is compatible with all testing frameworks that can be used with the Java programming language. Some of them are JUnit, Hamcrest, etc.

To end up with all the tables, we have the **Portability** one containing both Install-ability and Portability compliance features. In the following Portability table we can see the first three points as Install-ability ones and the resting ones related with the Portability compliance:

	Time for downloa ding	Time for installing	Time to get it running	Format compati bility	Fast import and export
Weight (1, 2 or 3)	2	2	2	3	2
Presence (%)	100	100	90	90	70

The Time for downloading OrientDB was adequate. No registration was required and we had options for each operating system.

The Time for installing was really short, we had both .jar file and Maven dependency options.

The Time to get the system running was considered as adequate. No relevant problem was found during this phase.

For the Format compatibility in export and import operations, we found that OrientDB can work with GraphML by using the Tinkerpop Blueprints export and import operations.

For the consideration of the Fast import and export operations we can say the same as before. These operations don't require a really long time for the size for which we tested them, but they are not extremely fast. This is why we assign a score of 70% to it.

9.2 Final Selection

First of all, we would like to clarify that we are not going to give a final, absolute answer about which is the best system. What we are aimed to do is at analyzing as much points as possible to get a conclusion or selection of the best systems according to different criteria. An example of this could be the fact of having a really good system in terms of performance but very bad for functionality.

We would like to give, just as a reference, the final scores of each system. They are computed as the sum of each individual score multiplied by its corresponding weight. Then, we finally divide by the sum of the maximum value the weights can have and also divide by 3 to obtain the number expressed as a percentage. We divide by three to have the weights expressed as a 1 base. With maximum weight we only mean that for the negative weights we consider a value of 0, as it is the best or maximum value for them. The formula would be:

$$Final\ Score_{s \in systems} = \frac{\sum_{f \in features} weight_{fs} \times score_{fs}}{3 \times \sum_{f \in features} \max(weight_{fs})}$$

And the table containing these values, where each row represents an "*s*", is shown below:

System	**Final Score**
Dex	20134 / (3*88) = **76.26%**
HyperGraphDB	5180 / (3*88) = **19.62%**
Neo4j	7230 / (3*88) = **27.39%**
OrientDB	7395 / (3*88) = **28.01%**

We can observe a really big difference between Dex, the one with the highest final score, and the other ones. However, we have to compare systems according to specific criteria.

What we do now is performing a comparison according to two points of view. The first one consists of comparing by features or criteria in order to decide which system or systems was/were seen as the best one/s for a specific point. The second viewpoint refers to comparing each DBMS on its own, without caring about other systems' results, but just which point or category is the strongest one for the system.

9.2.1 Feature viewpoint comparison

For the first comparison, we build the following tables. We remind each table corresponds to a comparison point and this allows for having a picture of the best system for each feature.

1. **Create node/relation:**

System	Score
Dex	**70**
HyperGrDB	**100**
Neo4j	**90**
OrientDB	**90**

2. **Modify node/relation:**

System	Score
Dex	**80**
HyperGrDB	**80**
Neo4j	**80**
OrientDB	**60**

3. **Remove node/relation:**

System	Score
Dex	**100**
HyperGrDB	**80**
Neo4j	**100**
OrientDB	**100**

4. Create relation type:

System	Score
Dex	**90**
HyperGrDB	**80**
Neo4j	**100**
OrientDB	**80**

5. All relation type:

System	Score
Dex	**70**
HyperGrDB	**70**
Neo4j	**100**
OrientDB	**100**

6. Traverse tree:

System	Score
Dex	**60**
HyperGrDB	**70**
Neo4j	**90**
OrientDB	**100**

7. Shortest path:

System	Score
Dex	**100**
HyperGrDB	**0**
Neo4j	**100**
OrientDB	**0**

8. All node's data:

System	Score
Dex	**80**
HyperGrDB	**100**
Neo4j	**100**
OrientDB	**100**

9. Get ancestors /descendants:

System	Score
Dex	**80**
HyperGrDB	**80**
Neo4j	**100**
OrientDB	**80**

10. **Nbr. descs. /ancestors:**

System	Score
Dex	**80**
HyperGrDB	**90**
Neo4j	**80**
OrientDB	**80**

11. **Export tree:**

System	Score
Dex	**100**
HyperGrDB	**0**
Neo4j	**100**
OrientDB	**100**

12. **Export tree's branch:**

System	Score
Dex	**100**
HyperGrDB	**0**
Neo4j	**100**
OrientDB	**100**

13. **Import data:**

System	Score
Dex	**100**
HyperGrDB	**0**
Neo4j	**100**
OrientDB	**100**

14. **Query by field (age):**

System	Score
Dex	**80**
HyperGrDB	**100**
Neo4j	**90**
OrientDB	**90**

15. **Get alive people:**

System	Score
Dex	**80**
HyperGrDB	**100**
Neo4j	**90**
OrientDB	**90**

16. **Get distinct surnames:**

System	Score
Dex	90
HyperGrDB	90
Neo4j	90
OrientDB	90

17. **Get all birth cities:**

System	Score
Dex	90
HyperGrDB	90
Neo4j	90
OrientDB	90

18. **Password to authenticate:**

System	Score
Dex	**0**
HyperGrDB	**0**
Neo4j	**60**
OrientDB	100

19. **Database backup:**

System	Score
Dex	100
HyperGrDB	**0**
Neo4j	**20**
OrientDB	**90**

20. **Different privacy levels:**

System	Score
Dex	100
HyperGrDB	100
Neo4j	100
OrientDB	100

21. **Operations not available:**

System	Score
Dex	0
HyperGrDB	**20**
Neo4j	0
OrientDB	**5**

22. **Low-level elements:**

System	Score
Dex	**30**
HyperGrDB	**0**
Neo4j	**0**
OrientDB	**0**

23. **DBMS specific bugs:**

System	Score
Dex	**0**
HyperGrDB	**30**
Neo4j	**20**
OrientDB	**5**

24. **Switch databases:**

System	Score
Dex	**0**
HyperGrDB	**0**
Neo4j	**0**
OrientDB	**0**

25. **Time to read & understand:**

System	Score
Dex	**80**
HyperGrDB	**50**
Neo4j	**100**
OrientDB	**80**

26. **Implementation time:**

System	Score
Dex	**70**
HyperGrDB	**60**
Neo4j	**100**
OrientDB	**70**

27. **Query languages:**

System	Score
Dex	**60**
HyperGrDB	**0**
Neo4j	**100**
OrientDB	**90**

28. Visualization tools quality:

System	Score
Dex	**0**
HyperGrDB	**70**
Neo4j	**100**
OrientDB	**80**

29. User transparency:

System	Score
Dex	**100**
HyperGrDB	**100**
Neo4j	**100**
OrientDB	**100**

30. Time behavior:

System	Score
Dex	**100**
HyperGrDB	**80**
Neo4j	**40**
OrientDB	**90**

31. Resources utilization:

System	Score
Dex	**100**
HyperGrDB	**80**
Neo4j	**50**
OrientDB	**80**

32. Storage size:

System	Score
Dex	**100**
HyperGrDB	**0**
Neo4j	**0**
OrientDB	**100**

33. Nbr. of nodes, rels. & indxs.:

System	Score
Dex	**66.67**
HyperGrDB	**90**
Neo4j	**70**
OrientDB	**95**

34. **Compaction operations:**

System	Score
Dex	**0**
HyperGrDB	**40**
Neo4j	**0**
OrientDB	**70**

35. **Data occ. little space:**

System	Score
Dex	**0**
HyperGrDB	**0**
Neo4j	**0**
OrientDB	**10**

36. **Change relation types:**

System	Score
Dex	**90**
HyperGrDB	**80**
Neo4j	**100**
OrientDB	**80**

37. **Add new indexes:**

System	Score
Dex	**100**
HyperGrDB	**100**
Neo4j	**100**
OrientDB	**100**

38. **Lack of DBMS spec. errors:**

System	Score
Dex	**100**
HyperGrDB	**70**
Neo4j	**100**
OrientDB	**90**

39. **Log data quality:**

System	Score
Dex	**100**
HyperGrDB	**50**
Neo4j	**90**
OrientDB	**70**

40. **Tools for debugging:**

System	Score
Dex	70
HyperGrDB	70
Neo4j	70
OrientDB	70

41. **Test frameworks:**

System	Score
Dex	**70**
HyperGrDB	**70**
Neo4j	80
OrientDB	**70**

42. **Time for downloading:**

System	Score
Dex	**70**
HyperGrDB	100
Neo4j	100
OrientDB	100

43. **Time for installing:**

System	Score
Dex	**80**
HyperGrDB	100
Neo4j	100
OrientDB	100

44. **Time to get it running:**

System	Score
Dex	**90**
HyperGrDB	**90**
Neo4j	100
OrientDB	**90**

45. **Format compatibility:**

System	Score
Dex	100
HyperGrDB	**0**
Neo4j	**90**
OrientDB	**90**

46. **Fast import and export:**

System	Score
Dex	70
HyperGrDB	**0**
Neo4j	70

OrientDB	70

With these tables we can clearly see that for dimension 1. (**create node and relation**), system HyperGraphDB deserves being the best one because of the use of domain objects to be stored in the database. Dex, instead, is seen as the more limited for this dimension because of the, already cited, use of low-level elements.

For dimension 2. (**modify node and relation**) we cannot see any system especially better than the others. The only remarkable thing is the fact that OrientDB is the most limited one, due to the need of removing the old node or relation and creating a new one.

For dimension 3. (**remove node and relation**) we can see most of the systems support it correctly. The only limited system in this sense is HyperGraphDB due to the bug that we saw before.

For **creating relation types** (dimension 4.) we can see Neo4j is the best system, as it uses the already cited Enums. There is no system completely bad for this dimension; we can only observe HyperGraphDB and OrientDB are the most limited ones, as they require for changing a relation for changing a relation type.

For **getting all relation types** (5. dimension), Neo4j and OrientDB were seen as the best ones as they allow for building a query before iterating and, then, we don't need to filter inside the loop. An implementation like this one couldn't be obtained for the other systems.

For the **traverse tree dimension** (6.) we can see that the best support is provided by OrientDB and the worst by Dex. For OrientDB we can do this by using more or less only one instruction, and for Dex it is limited, as we said, in terms of performance and easiness.

For the **shortest path dimension** (7.), we can identify two systems (Dex and Neo4j) for which it was completely supported by using a specific API method. And we can also see that for the other systems, this operation was not supported correctly.

For the **all node's data dimension** (8.) we cannot see any remarkable system. We can identify Dex as the most limited one, because of the use of low-level elements (Value object).

Neo4j was the best system for the **get ancestors and descendants** dimension (9.). The implementation obtained with this system allows us to specify a maximum level value and it has no limitations.

For the 10. dimension, i.e. the **number of descendants and ancestors**, we found HyperGraphDB was the best one, as it allows for creating a traversal from the node and then iterating over it. Neo4j also allows for the same, but we have to iterate over an Iterable<Node>, so we have to get a value that we are not finally using.

For the **export tree** or 11. dimension we can see it is fully supported for most of the systems. The only system that doesn't support it is HyperGraphDB, since it is not compatible with Tinkerpop Blueprints. The same can be said for the **export tree's branch** or 12. dimension and the **import data** or 13. dimension.

The system that best suits for the **query for age** dimension (14.) and the **query for alive people** (15.) one is HypergraphDB. This is due to the fact of supporting domain objects as part of the persistence layer, which eases this operation.

For both **get distinct surnames** (16.) and **get all birth cities** (17.) dimensions, we can observe an almost complete support for all systems, with the only problem of not allowing for field projections.

The best system for **password authentication** (18. dimension) was OrientDB. Neo4j more or less supports it, but is a little bit limited; and the resting ones don't support it at all.

For the **database backup** (19. dimension) we saw the best system was Dex, as it provides both backup and restore operations or methods. For the resting ones, they either not support it (HyperGraphDB) or they give support but in a rather artificial way (e.g. copying the whole database folder).

For all DBMS, we found the same support for **different privacy levels** (20. dimension). For all of them we didn't find any problem for being able to guarantee this.

The best systems in terms of **operations support** (21. dimension) are Neo4j and Dex, with all the required operations supported. The worst in this sense was HyperGraphDB with 20% of the required operations unsupported (concretely, the import and exports and the shortest path).

For the **use of low-level elements** (22. dimension) we found that Dex was the only one violating this point, especially with the use of Value objects and also other low-level elements that made the development a little bit more difficult.

For the **DBMS specific bugs** dimension (23.) we found that Dex is the only system 100% free of errors. The system with highest presence of this kind of errors was HyperGraphDB, especially the already cited problem with the remove node operation.

The **ability to switch between databases** (dimension 24.) was seen as completely supported by all systems. We couldn't find any problem for which this wouldn't be possible.

Neo4j was seen as the system for which the **time for reading and understanding** (dimension 25.) was the shortest one, for the reasons already cited (basically quantity and variety of documentation). The system that was seen as the most limited in this sense is HyperGraphDB; for it, few documentation was found and almost nothing to be downloaded as a pdf. Exactly the same happens with the **implementation time** (dimension 26.).

For the presence of **different query languages** (27.), the best system was Neo4j, as it provides the API, CYPHER and GREMLIN languages and also support for the Tinkerpop Blueprints driver. The most limited system in this sense was HyperGraphDB, only providing the native API.

For the **visualization tools** dimension (28.) we found Neo4j as the best one, with all the capabilities of the web server, accessible through a web browser. Dex, instead, was the system for which we couldn't find support in this dimension.

For the **user transparency** dimension (29.) we found it is fully supported for all systems. I.e., we didn't find any problem for guaranteeing that the final user is unaware of the DBMS used.

In the **time behavior** dimension (30.) we detected Dex as the fastest system and Neo4j as the slowest one, by executing the same list of operations. Here we can observe the results are fair, as these are precisely the systems for which all operations were supported too. The same result can be observed in the **resources utilization** dimension (31.).

Dex and OrientDB were the only systems for which we found support for getting information about the current **storage size** (32. dimension). For the resting systems this operation was seen as unsupported or inexistent.

The best support for getting the **number of nodes, number of relations and number of indexes** (dimension 33.) in the system was found using OrientDB, as it provided support for all of them. The worst case in this sense was seen in Dex, since it didn't support the indexes part.

The best information about **compaction operations** (34. dimension) was found for OrientDB. For the other ones, no information or very few was found.

Almost no support was found for all DBMS, in the 35. Dimension, i.e. the **guarantee of having the data occupying as little space as possible**.

For **changing the relation types** (36. dimension) we found the best support using Neo4j, i.e. with a Java Enum containing all possible relation types. For Dex, the API provided a method for doing this. Finally, the most limited in this sense were the resting ones (i.e. HyperGraphDB and OrientDB) because of not allowing relation type edition without changing relations.

Add new indexes operation (37. dimension) was seen as fully supported by all systems in an adequate way.

For the **DBMS specific errors** (38. dimension) we saw, as we said, Neo4j and Dex were the ones free of errors. Then, the one with the highest quantity of errors was HyperGraphDB, as already cited.

The best system in terms of **log data** (39. dimension) was Dex, as it supports it in an automatic way, just starting the system, and it is configurable. HyperGraphDB is the DBMS for which we found the minimum quantity of information about this point.

For the **debugging tools** (dimension 40.) we saw the same situation for all DBMS: some are supported, but no DBMS specific tools, only the ones in general compatible with Java programming language. More or less the same can be said for the **testing frameworks** (dimension 41.); we only assign a little bit more punctuation to Neo4j since, for it, this is officially documented, by a JUnit and Hamcrest tutorial.

The **time for downloading** (dimension 42.) and the **time for installing** (dimension 43.) was adequate for all systems. The only system with a small penalty in these dimensions is Dex, because a registration for downloading was needed, and also because we couldn’t use Maven for installing it, only the .jar file.

All DBMS provided an adequate **time to get them running** (44. dimension). This was especially good in the case of Neo4j and we remark this is due to the documentation, examples, etc. provided for it.

The **format compatibility** dimension (45.) was fully supported by Dex, with the possibility of exporting to CSV and GraphML formats. The worst case in this dimension was seen when using HyperGraphDB, as it seems like not supporting export operations at all.

Finally, for the last dimension (i.e. 46. or **fast import and export**) we saw all systems provide an adequate support, except for HyperGraphDB, for which we couldn't find an implementation of the import operation.

9.2.2 DBMS viewpoint comparison

Now we are going to show the tables split by DBMS, i.e. the comparison among all features of each system without mixing or comparing the results with other systems. We note this table is just like the ones seen before in each DBMS section, but now we want to put all together in only one table to get the better results for each system.

We can see in violet the operations that are the best ones defined in a positive way; and in yellow the best ones, among the ones defined in a negative way (with maximum value as 0).

Dex table:

Feature or operation	Score
1. Create node/relation	70
2. Modify node/relation	80
3. Remove node/relation	**100**
4. Create relation type	90
5. All relation types	70
6. Traverse tree	60

7. Shortest path	100
8. All node's data	80
9. Get ancestors/descendants	80
10. Number of descendants/ancestors	80
11. Export tree	100
12. Export tree's branch	100
13. Import data	100
14. Query by field (age)	80
15. Get alive people	80
16. Get distinct surnames	90
17. Get all birth cities	90
18. Password authentication	0
19. Backup of the database	100
20. Different privacy levels	100
21. Operations still not available (negatively defined)	0
22. Use of low-level elements (negatively defined)	30
23. Bugs caused by the DBMS (negatively defined)	0
24. Switch between databases (negatively defined)	0
25. Time to read and understand	80
26. Adequate implementation time	70
27. Different query languages	60
28. Visualization tools quality	0

29. User transparency	100
30. Time behavior	100
31. Resources utilization	100
32. Storage size	100
33. Number of nodes, relations and indexes	67
34. Compaction operations	0
35. Data occupying as little space	0
36. Change relation types	90
37. Add new indexes	100
38. Lack of DBMS specific errors	100
39. Quality of log data	100
40. Tools for debugging	70
41. Testing frameworks	70
42. Time for downloading	70
43. Time for installing	80
44. Time to get it running	90
45. Format compatibility	100
46. Fast import and export	70

This means that the operations or features that are best supported for Dex are: "Format compatibility", "Quality of log data", "Lack of DBMS specific errors", "Add new indexes" and so on.

HyperGraphDB table:

Feature or operation	Score
1. Create node/relation	100
2. Modify node/relation	80
3. Remove node/relation	80
4. Create relation type	80
5. All relation types	70
6. Traverse tree	70
7. Shortest path	0
8. All node's data	100
9. Get ancestors/descendants	80
10. Number of descendants/ancestors	90
11. Export tree	0
12. Export tree's branch	0
13. Import data	0
14. Query by field (age)	100
15. Get alive people	100
16. Get distinct surnames	90
17. Get all birth cities	90
18. Password authentication	0
19. Backup of the database	0
20. Different privacy levels	100

21. Operations still not available (negatively defined)	20
22. Use of low-level elements (negatively defined)	0
23. Bugs caused by the DBMS (negatively defined)	30
24. Switch between databases (negatively defined)	0
25. Time to read and understand	50
26. Adequate implementation time	60
27. Different query languages	0
28. Visualization tools quality	70
29. User transparency	**100**
30. Time behavior	80
31. Resources utilization	80
32. Storage size	0
33. Number of nodes, relations and indexes	90
34. Compaction operations	40
35. Data occupying as little space	0
36. Change relation types	80
37. Add new indexes	**100**
38. Lack of DBMS specific errors	70
39. Quality of log data	50
40. Tools for debugging	70
41. Testing frameworks	70
42. Time for downloading	**100**

43. Time for installing	100
44. Time to get it running	90
45. Format compatibility	0
46. Fast import and export	0

In HyperGraphDB table we can observe it is a system suitable for e.g. getting "All node's data" and also for "Creating new nodes and relations".

Neo4j table:

Feature or operation	Score
1. Create node/relation	90
2. Modify node/relation	80
3. Remove node/relation	100
4. Create relation type	100
5. All relation types	100
6. Traverse tree	90
7. Shortest path	100
8. All node's data	100
9. Get ancestors/descendants	100
10. Number of descendants/ancestors	80
11. Export tree	100
12. Export tree's branch	100
13. Import data	100

14. Query by field (age)	90
15. Get alive people	90
16. Get distinct surnames	90
17. Get all birth cities	90
18. Password authentication	60
19. Backup of the database	20
20. Different privacy levels	100
21. Operations still not available (negatively defined)	0
22. Use of low-level elements (negatively defined)	0
23. Bugs caused by the DBMS (negatively defined)	20
24. Switch between databases (negatively defined)	0
25. Time to read and understand	100
26. Adequate implementation time	100
27. Different query languages	100
28. Visualization tools quality	100
29. User transparency	100
30. Time behavior	40
31. Resources utilization	50
32. Storage size	0
33. Number of nodes, relations and indexes	70
34. Compaction operations	0
35. Data occupying as little space	0

36. Change relation types	100
37. Add new indexes	100
38. Lack of DBMS specific errors	100
39. Quality of log data	90
40. Tools for debugging	70
41. Testing frameworks	80
42. Time for downloading	100
43. Time for installing	100
44. Time to get it running	100
45. Format compatibility	90
46. Fast import and export	70

For Neoj4 we can observe a good support for e.g. "Different kinds of query languages supported" and "Visualization tools", among others.

OrientDB table:

Feature or operation	Score
1. Create node/relation	90
2. Modify node/relation	60
3. Remove node/relation	100
4. Create relation type	80
5. All relation types	100
6. Traverse tree	100

7. Shortest path	0
8. All node’s data	100
9. Get ancestors/descendants	80
10. Number of descendants/ancestors	80
11. Export tree	100
12. Export tree’s branch	100
13. Import data	100
14. Query by field (age)	90
15. Get alive people	90
16. Get distinct surnames	90
17. Get all birth cities	90
18. Password authentication	100
19. Backup of the database	90
20. Different privacy levels	100
21. Operations still not available (negatively defined)	5
22. Use of low-level elements (negatively defined)	0
23. Bugs caused by the DBMS (negatively defined)	5
24. Switch between databases (negatively defined)	0
25. Time to read and understand	80
26. Adequate implementation time	70
27. Different query languages	90
28. Visualization tools quality	80

29. User transparency	**100**
30. Time behavior	90
31. Resources utilization	80
32. Storage size	**100**
33. Number of nodes, relations and indexes	95
34. Compaction operations	70
35. Data occupying as little space	10
36. Change relation types	80
37. Add new indexes	**100**
38. Lack of DBMS specific errors	90
39. Quality of log data	70
40. Tools for debugging	70
41. Testing frameworks	70
42. Time for downloading	**100**
43. Time for installing	**100**
44. Time to get it running	90
45. Format compatibility	90
46. Fast import and export	70

OrientDB’s table shows it is a good system for having “Password authentication” and also for “Traversing the tree”, among others.

10. Discussion and conclusions

After comparing all DBMS in many different ways we can get some conclusions that we state in this part.

The first conclusion is that even though Dex was the DBMS with the biggest global or average punctuation, this doesn't mean it is the best one of them. Dex, for example is a system having not the best punctuation for "Create node/relation" nor "Modify node/relation" that are important operations with a weight of 3 (the maximum one).

This is an important conclusion, since it allows us to state that it is not possible to get a unique, final answer about the best system. This analysis allowed us to get better results than without a context, as previous studies did. However, the effectiveness and efficiency of the system would depend on the concrete operation or feature we are analyzing.

We can draw many conclusions for each DBMS and each operation by analyzing the results. Some of the most remarkable ones are what follows.

Dex is a really good system for performance, i.e. we can execute operations in a really fast way and with the minimum resources or memory consumption. It is also very good in terms of what we understand for maturity, i.e. Dex is one of the two systems, together with Neo4j, for which we found an implementation for all required operations

The main point in which Dex was seen as not so good is the API, there were many methods using a slightly unclear naming and the, already mentioned, use of low-level elements, such as Value objects.

For HyperGraphDB, we found it as a good system for storing information directly as domain objects. This was seen as very flexible and useful for many operations. Concretely, queries for a specific field become easier and more efficient, since we don't need to transform database objects into domain ones, we can directly obtain domain objects as a result of the query.

One of the main downsides of HyperGraphDB is, however, the fact of not supporting right now many operations, so we found here a rather important problem of maturity. Another relevant problem, related to the previous one is the fact of not supporting data import nor export operations. This represents a big problem, since we want this system to be compatible with GEDCOM format and for that we need these operations.

Neo4j is a system that provides a really clear API, a lot of documentation, examples, a pdf manual, etc. It also seems to have a rather good level of maturity for being, together with Dex, one of the two systems being compatible with all the methods or operations required. It was seen as a system with which we can start working in a quite fast way, because of all documentation provided, the possibility of using a visualizer, a console browser and many other tools provided by a web server. Compared with Dex, it was also seen as kind of "higher-level" than it. Neo4j's API was seen cleaner, without low-level elements and being able to manage relation types as Java Enum values.

However, the biggest problem that was found for Neo4j is in performance. As it was said before, Neo4j was seen as many times slower than the other DBMS.

OrientDB is the DBMS in which we achieved the best implementation of the traverse tree operations and one of the best, together with Neo4j, for getting all relation types operation. It is also a more or less good system, compared with the others, in terms of support for maintenance operations, such as database compaction operations.

However, we saw a small problem of maturity for OrientDB because of not supporting the shortest path operation, considering that, according to the official google groups, it is expected to be supported.

Finally, according to this previous reflection, we would consider both Dex and Neo4j the systems that best supported our requirements and, thus, could more probably work better than the other two. We exclude from this group HyperGraphDB and OrientDB for all the operations not supported by them. These operations are considered as necessary for a genealogy system; we would need to get the shortest path to determine the relationship between two people if any and this is not supported by OrientDB nor HyperGraphDB. The last one also misses support for both import and export that are needed, as we explained before, for guaranteeing GEDCOM compatibility.

Moreover, we have to remind that finding an alternative format for storing family trees with graph NoSQL DBMS (a part from GEDCOM) was also one of the objectives of the thesis. Then, although it was mentioned many times, we have to say that GraphML was the format found for that and which is considered as a standard for storing graphs in many graph NoSQL DBMS.

We also need to make a short reference to the initial planning to conclude if it was adequate or not. Our conclusion is that we were able to follow it correctly for most of the thesis parts. The only exception, in which we could consider a small deviation, is the DBMS comparison part. This part ended up being longer than expected and, thus, it took some days more than what we initially planned.

Finally, we can state that one of our objectives was accomplished, i.e. the contribution to the knowledge of graph NoSQL DBMS. Also, as expected, we were able to get conclusions with a fixed domain. We have to note an interesting fact which is the coincidence in our conclusions with one of the previous works (concretely, [DPHSGAB]), in which also Dex and Neo4j were seen as the best ones.

11. Bibliography and other references

[APSTGC] Neo Technology company. A pleasant stroll through general concepts, and Neo4j particulars. [Video] http://www.neo4j.org/learn#graphdbIntro (minute 16:30).

[DPHSGAB] Domínguez-Sal, D.; Urbón-Bayes, P.; Giménez-Vañó, A.; Gómez-Villamor, S.; Martínez-Bazán, N.: Larriba-Pey , J. L. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. DAMA-UPC, Universitat Politècnica de Catalunya. [H.T. Shen et al. (Eds.): WAIM 2010 Workshops, LNCS 6185, pp. 37–48], 2010.

[ACCGDM] Angles, Renzo. A Comparison of Current Graph Database Models. University of Talca, Chile. icdew, pp.171-177, 2012 IEEE 28th International Conference on Data Engineering Workshops, 2012

[SGDM] Angles, Renzo; Gutiérrez, Claudio. Survey of Graph Database Models. ACM Comput. Surv. 40, 1, Article 1, 39 pages. 2008

[QMSSE] Bandor, Michael S. Quantitative Methods for Software Selection and Evaluation. Carnegie Mellon University. 2006

[UQMSPS] Franch, Xavier; Carvallo, Juan Pablo. Using Quality Models in Software Package Selection. Universitat Politècnica de Catalunya. 2003

[CST] Column-oriented Database Systems. Harizopoulos, Stavros (HP Labs), Abadi, Daniel (Yale), Boncz, Peter (CWI). VLDB 2009 Tutorial. http://cs-www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf. 2009

[NEO4J] Learn, Develop, Participate - Neo4j The World's Leading Graph Database. http://www.neo4j.org/

[INFGRAPHDB] Product Objectivity. http://objectivity.com/products/infinitegraph/technical-specifications

[FLOCKDB] Twitter Engineering Introducing FlockDB. http://engineering.twitter.com/2010/05/introducing-flockdb.html

[PREGEL] Pregel. http://dl.acm.org/citation.cfm?id=1582716.1582723

[PHOEBUS] xslogic phoebus · GitHub. https://github.com/xslogic/phoebus

[JPREGEL] JPregel @ GitHub. http://kowshik.github.com/JPregel/index.html

[ARANGODB] ArangoDB - The universal free and open source nosql database. http://www.arangodb.org/

[HYPGRADB] HypergraphDB - A Graph Database. http://www.hypergraphdb.org/index

[INFOGRID] InfoGrid Web Graph Database. http://infogrid.org/trac/

[DEX] Sparsity-technologies DEX high-performance graph database. http://www.sparsity-technologies.com/dex

[BIGDATA] bigdata®. http://www.systap.com/bigdata.htm

[ORIENT] OrientDB Graph-Document NoSQL dbms. http://www.orientdb.org/

[ORIENTSLIDES] OrientDB-the-graph-database-2.0 http://2012.nosql-matters.org/cgn/wp-content/uploads/2012/06/OrientDB-the-graph-database-2.0.pdf

[TITAN] Titan. http://thinkaurelius.github.com/titan/

[VERTDB] stevedekorte vertexdb · GitHub. https://github.com/stevedekorte/vertexdb

[TWITTER] Twitter. https://twitter.com/

[STACKOVER] Stack Overflow. http://stackoverflow.com/

[GOOGLE] Google. http://www.google.com/

[CODD] A Relational Model of Data for Large Shared Data Banks. Codd, E. F. IBM Research Laboratory, San Jose, California. Communications of the ACM. Volume 13. Number 6. June, 1970. http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

[GEDCOM_SYNC] Why GEDCOM Files are not in Sync with all Genealogy Programs. ourFamilyology, Inc. http://www.family-genealogy.com/downloads/GEDCOM_Sync_Issues.pdf

[DBRANK] DB-Engines Ranking - popularity ranking of relational DBMS. http://db-engines.com/en/ranking/relational+dbms

[ORACLEDB] Oracle System Properties. http://db-engines.com/en/system/Oracle

[ORACLETIME] Oracle Anniversary timeline. http://www.oracle.com/us/corporate/profit/p27anniv-timeline-151918.pdf

[MYSQL] MySQL System Properties http://db-engines.com/en/system/MySQL

[MYSQLHIST] MySQL MySQL 5.1 Reference Manual 1.3.3 History of MySQL. http://dev.mysql.com/doc/refman/5.1/en/history.html

[NOSQL] NoSQL Databases Christof Strauch. Hochschule der Medien, Stuttgart. http://www.christof-strauch.de/nosqldbs.pdf

[TOPBIG] Top Big Data skills MongoDB and Hadoop 10gen. http://www.10gen.com/post/40021118238/mongodb-and-hadoop-are-top-big-data-skills

[COMPDOCKEY] Comparing Document Databases to Key-Value Stores • myNoSQL. http://nosql.mypopescu.com/post/659390374/comparing-document-databases-to-key-value-stores

[GOODFOR] pstaender mongraph · GitHub. https://github.com/pstaender/mongraph#whats-it-good-for

[HPACQ] HP News - HP Completes Acquisition of Vertica Systems, Inc.
http://www8.hp.com/us/en/hp-news/press-release.html?id=907883&pageTitle=HP%20Completes%20Acquisition%20of%20Vertica%20Systems,%20Inc.#.UYw3wrXIbw8

[GVSSQL] Graphs vs. SQL. Interview with Michael Blaha   ODBMS Industry Watch.
http://www.odbms.org/blog/2013/04/graphs-vs-sql-interview-with-michael-blaha/

[GENERIC] Generic Types (The Java™ Tutorials   Learning the Java Language   Generics (Updated)). http://docs.oracle.com/javase/tutorial/java/generics/types.html

[YWORKS] yEd - Graph Editor. http://www.yworks.com/en/products_yed_about.html

[MVN] Maven - Welcome to Apache Maven. http://maven.apache.org/

[SVN] subversion.tigris.org. http://subversion.tigris.org/

[SUBCLIPSE] subclipse.tigris.org. http://subclipse.tigris.org/

[ECLIPSE] Eclipse - The Eclipse Foundation open source community website.
http://www.eclipse.org/

[M2E] m2eclipse. http://eclipse.org/m2e/

[ORACLENOSQL] Oracle NoSQL Database Technical Overview.
http://www.oracle.com/technetwork/products/nosqldb/overview/index.html

[JVIRTUALVM] jvisualvm - Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool. http://docs.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html

[HYPER_SLIDES] HyperGraphDB Data Management for Complex Systems. (Slides).
http://www.hypergraphdb.org/docs/HyperGraphDB-Presentation.pdf

[SERVER_INSTALL] 21.1. Server Installation. http://docs.neo4j.org/chunked/stable/server-installation.html

[AUTHENTICATION_EXT] neo4j-contrib authentication-extension · GitHub. https://github.com/neo4j-contrib/authentication-extension

[SECURITY_SERVER] 31.1. Securing access to the Neo4j Server. http://docs.neo4j.org/chunked/milestone/security-server.html

[GREMLIN] 22.18. Gremlin Plugin. http://docs.neo4j.org/chunked/stable/gremlin-plugin.html

[CYPHER] 10.1. What is Cypher. http://docs.neo4j.org/chunked/stable/cypher-introduction.html

[GEPHI] Neo4j   Gephi, open source graph visualization software. http://gephi.org/tag/neo4j/

[REMOTE_DEBUG] 21.3. Setup for remote debugging. http://docs.neo4j.org/chunked/stable/server-debugging.html

[BASIC_UNIT_TEST] 4.4. Basic unit testing. http://docs.neo4j.org/chunked/stable/tutorials-java-unit-testing.html

[ORIENT_BACK] What's the best way to backup an orientdb database - Google Groups. https://groups.google.com/forum/?fromgroups#!topic/orient-database/IsLCmHZk50s

[ORIENT_AUTO_BACK] Automatic Backup · nuvolabase orientdb Wiki · GitHub. https://github.com/nuvolabase/orientdb/wiki/Automatic-Backup

[ORIENT_STUDIO] OrientDB_Studio - orient - OrientDB Studio web application - NoSQL document database light, portable and fast. Supports ACID Tx, Indexes, asynch queries, SQL layer, clustering, etc - Google Project Hosting. https://code.google.com/p/orient/wiki/OrientDB_Studio

[ORIENTDB_COMPACT] OrientDB and replication, scalability and fault-tolerance comments please! - Google Groups. https://groups.google.com/forum/#!msg/orient-database/ldWerjEgH18/KKbJSXLU_ckJ

[DEBUG_SERVER] DBServer - orient - OrientDB Server - NoSQL document database light, portable and fast. Supports ACID Tx, Indexes, asynch queries, SQL layer, clustering, etc - Google Project Hosting. https://code.google.com/p/orient/wiki/DBServer#Debug_the_server

[INFOQ] Graph Databases, NOSQL and Neo4j. http://www.infoq.com/articles/graph-nosql-neo4j