Vrije Universiteit Brussel

FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRONICS AND INFORMATICS

Investigating Graph-based Storage Backends for Hypermedia Link Services

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Applied Computer Science

Jonas Vereycken

Promotor: Prof. Dr. Beat Signer
Advisor: Reinout Roels

Academic year 2015-2016

Abstract

As the linking of units of information is the key role of hypermedia systems, the storage layer for such systems should be adapted accordingly. Relational databases will not do the trick because of their expensive joins. Graph databases seem to be a better fit for handling this interconnected data and this solution is further investigated in this thesis. To do so, hypermedia models should be understood clearly. This is done based on the RSL metamodel, the iServer linking service and the EdFest application. Later on, graph databases and their features are discussed. Based on a comparison of commercially available graph databases, Neo4j and OrientDB are the chosen databases to import the EdFest data into. Different approaches for designing graph data models are discussed to do so. For both Neo4j and OrientDB, three data models are proposed based on these approaches. A first model resembles the most to the RSL metamodel concepts, while the two others are both simplifications that should bring more performance by giving up some functionality. The Edfest data was then loaded into the databases according to these data models. Afterwards, the different data models and the different databases are benchmarked to each other by means of an empirical study. The results show some performance differences between Neo4j and OrientDB. While Neo4j is more performant for queries that involve a lot of graph traversals, OrientDB is a lot faster when a larger amount of results should be returned.

Declaration of Originality

I hereby declare that this thesis was entirely my own work and that any additional sources of information have been duly cited. I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this thesis has not been submitted for a higher degree to any other University or Institution.

Acknowledgements

This thesis will be submitted to obtain the degree of Master of Science in Applied Sciences and Engineering: Applied Computer Science. First of all, I want to thank my parents to support me to study, both mentally and financially. The motivation I get thanks to their interest in my studies and in my life in general cannot be underestimated. I am sure they are proud that I will graduate in this Master program.

Furthermore, I want to thank Reinout Roels, who gave me the chance to write a thesis on this subject. His great help during our weekly meetings was very welcome. He helped to shape this thesis and was always there when I got stuck.

I also want to thank everyone that helped me during my studies, especially all the professors that took the time to teach me interesting new things. In particular, I want to thank Prof. Dr. Beat Signer, promotor of this thesis, to arouse my interest in this topic around databases.

To all of you, thank you.

Contents

1 Introduction

2 Hypermedia Link Services

2.1 Hypermedia . . . 5
2.2 The RSL Hypermedia Metamodel . . . 6
2.3 iServer and EdFest . . . 9

3 Graph Databases

3.1 Graph Databases and the NoSQL Movement . . . 11
3.2 Features of Graph Databases . . . 13
3.3 Use Cases . . . 16

4 Comparison of Existing Graph Databases

4.1 Overview . . . 19
4.2 The Different Database Systems . . . 21
4.2.1 ArangoDB . . . 21
4.2.2 InfiniteGraph . . . 21
4.2.3 MarkLogic . . . 21
4.2.4 Neo4j . . . 21
4.2.5 OrientDB . . . 22
4.2.6 Sparksee . . . 22
4.2.7 Stardog . . . 22
4.2.8 Titan . . . 22
4.3 Shortlist . . . 23

5 Mapping the RSL Model to Graph Databases

5.1 Possible Mappings . . . 25
5.1.1 Typing and Sub-typing . . . 25
5.1.2 Links . . . 27
5.1.3 Blobs (Binary Large Objects) . . . 28
5.1.4 Entity Properties . . . 28
5.2 Chosen Mappings . . . 29

6 Importing Data in Neo4j and OrientDB
6.1 The Data 31
6.2 The Data in a Graph Model 32
6.3 Importing the Data 35
6.3.1 Neo4j 36
6.3.2 OrientDB 36

7 Benchmarking Neo4j and OrientDB
7.1 Read Performance 38
7.1.1 Read Whole Database 38
7.1.2 Read Entity with Specific Name 39
7.1.3 Read Entity Who's Name Starts with Specific Character 40
7.1.4 Read All Links with their Sources and Targets 41
7.1.5 Conclusion Read Performance 43
7.2 Write Performance 44
7.2.1 Creating a New Entity 44
7.2.2 Creating a New Link 45
7.2.3 Creating a New Instance 47
7.2.4 Conclusion Write Performance 51
7.3 Traversal Performance 52
7.3.1 Shortest Path 52
7.3.2 All Paths 54
7.3.3 Conclusion Traversal Performance 55
7.4 Conclusion of the Benchmark 55

8 Conclusion and Future Work
8.1 Conclusion 57
8.2 Limitations 59
8.3 Future Work 60

A Appendix A
References 69

List of Figures

2.1 The core link metamodel 7
2.2 User management 8
2.3 Layers 8
2.4 Navigational and structural links 9

3.1 Graph example 14
3.2 Comparison of query execution of relational database and graph database 15

4.1 Overview of graph databases 20

5.1 Modelling inheritance in graph models using labels (Neo4j) or classes (OrientDB) 26
5.2 Modelling inheritance in graph models using other nodes . . . 27
5.3 Modelling links as edges 27
5.4 Modelling links as nodes 28
5.5 Modelling properties 29

6.1 Graph model ‘with explicit properties’ 33
6.2 Graph model ‘with explicit links’ 34
6.3 Graph model ‘without explicit links’ 35

7.1 Result of query in section 7.1.1 39
7.2 Result of query in section 7.1.2 40
7.3 Result of query in section 7.1.3 41
7.4 Result of query in section 7.1.4 43
7.5 Result of query in section 7.2.1 45
7.6 Result of query in section 7.2.2 46
7.7 Result of query in section 7.2.3 51
7.8 Result of query in section 7.3.1 53
7.9 Result of query in section 7.3.2 55

1
Introduction

In this thesis the potential and performance of graph-based back end storages are investigated for use in so-called hypermedia link services. In the context of hypermedia the linking or associating of information plays a central role and is often handled by a dedicated link service that is responsible for storing the links, metadata and sometimes the actual information itself. Clearly, as the amount of information and links grow, performance becomes an important factor. However, as with most information systems the performance of the storage layer is a deciding factor for the performance of the entire system. This implies that the underlying storage model may significantly impact the system's performance.

It is also clear that the underlying storage layer can influence the performance of some data operations more than others. However, we state that the operations required by link services are particularly sensitive to design choices made at the storage level. For instance, one might want to find all the items that link to a specific piece of information or one might need to traverse a chain of linked information. If the information and links would be stored in accordance to the relational model, large amounts of join operations would have to be performed to combine the data, which are known to be computationally expensive (Liu & Chirathamjaree, 1996). Therefore a storage layer based on the relational model would be a very bad choice for link services. As linked data takes on the shape of a graph, with information

as nodes and links as edges, it also makes sense to treat the information as a graph at the storage level. This means that it would also make sense to use storage approaches that are more suitable for graph-like structures.

As stated before this thesis will investigate alternative storage approaches focussing on graph databases in particular. While there are many link models and corresponding services, we will base ourselves on the principles described by the RSL hypermedia metamodel (Signer & Norrie, 2007) during our search for a better storage solution. The contribution of this thesis is therefore threefold. First of all, we investigated alternative storage approaches that do not rely on the relational model while keeping in mind the negative and positive sides of the recent NoSQL movement (McCreary & Kelly, 2014). A comparison of commonly used implementations is also made. Secondly, to be able to compare different database systems, database modelling in the context of link services needs to be done. This might be influenced by the way the database system works, so some of the design choices are discussed for the chosen database systems. Thirdly, we compared the performance of two database systems and some different design choices via an empirical study. This was achieved by making a benchmark for the different approaches by using a common data set and by measuring the performance of specific queries on the data.

In Chapter 2 we will start by explaining the ideas and concepts behind link services and then move on to detailing a specific hypermedia linking model, the RSL metamodel. This will help us to illustrate the connectedness of hypermedia, which allows us to motivate the use of graph databases for back end storage later on. Also EdFest, one of the real world application based on the RSL metamodel is mentioned here. The data of this application will later be used in this thesis as input for our benchmarks. To clearly understand what these graph databases exactly are, an explanation of graph databases is given in Chapter 3. Not only the features of graph databases are discussed in this chapter, but also the role of graph databases in the NoSQL movement is discussed. Furthermore, some typical use cases for graph databases are elaborated.

In Chapter 4, an overview of commercially available graph databases is given. Out of this long list, two database systems have been chosen based on some general criteria. These two databases will then be further investigated in the remainder of this thesis. To test these databases, there should first be a mapping between the existing data model and the data models for graph databases. Several design approaches are discussed in Chapter 5. In Chapter 6, more explanation about the used data is given and we will explain how the data is imported in the different databases. Also the different data

models are explained based on the different designing approaches presented in Chapter 4. Finally, in Chapter 7 the performance benchmarks for the two chosen databases and the different data models are detailed. The goal here is to not only compare the two commercially available databases, but also compare the consequences of the design decisions made in earlier chapters. This thesis is wrapped up in Chapter 8 with a discussion and conclusions, together with some suggestions for further research.

2

Hypermedia Link Services

In this chapter we will start by introducing hypermedia and the ideas behind hypermedia models. We will then discuss one particular hypermedia meta-model called the RSL model. We have chosen the RSL model as a reference for modelling a hypermedia system at the storage level, and it will be used as a starting point for work discussed in later chapters.

2.1 Hypermedia

In 1965, Ted Nelson was the first one to use the terms hypertext and hypermedia. While he defined hypertext as "non-sequential writing-text that branches and allows choices to the reader", hypermedia is a term to a similar concept but with non-textual data, such as images, videos or sound (Nelson, 1965). This data should thus not be sequentially followed, but the user of the hypermedia system is able to navigate through the content in a way that he wants. The navigation through the content is done via so-called hyperlinks.

The world wide web as it is today, is probably the most known example of a hypermedia system. However it is not fully navigable, users can find their own way through the Internet by clicking on words, part of texts, videos or images, which will link the user to other sources of information such as another webpage, another video, image, text and so on. It is of course unthinkable that a user should go through the internet sequentially.

A typical PowerPoint presentation however, is usually presented and followed in a sequential manner. The presence of movies, images or sound fragments in a PowerPoint presentation does not make the presentation an hypermedia system, since the user has no other choice than following the sequential path of the presentation.

To organise the content in hypermedia systems, there is a need of hypermedia models. These models define which kind of information can exist in the system (text, images, video, sound, ...) and also defines the links between these pieces of information. Notice that different types of links between the data can exist. Navigational links for example will guide you from one piece of information to another one. One can however also define links between two pieces of information that declare that the content the pieces concerned is related. By doing this, much smarter hypermedia systems can be built. On top of this, the model could also define user management for example.

Over the years, different hypermedia models have been developed, each with its own advantages and disadvantages. The Dexter Hypertext Reference Model for example, was intended to provide a basis for hypermedia systems. This makes that these systems can be compared to each other. Another goal of Dexter was to develop standards for interoperability and interchangeability (Halasz & Schwarz, 1994). The Amsterdam Hypermedia Model improved the Dexter model by adding the dimension of time and context (Hardman, Bulterman, & van Rossum, 1994). Another example of a well-known hypermedia model is the object-oriented hypermedia design model, developed in an object-oriented mindset (Schwabe & Rossi, 1995).

The proliferation of these hypermedia models resulted in the creation of a metamodel for hypermedia models. A metamodel provides the building blocks with which a model can address case specific issues. With these building blocks, more complex hypermedia models can be created. In the next section, such a hypermedia metamodel will be discussed: the RSL-metamodel, developed by Signer and Norrie (Signer & Norrie, 2007).

2.2 The RSL Hypermedia Metamodel

In the paper 'As We May Link: A General Metamodel for Hypermedia Systems' (Signer & Norrie, 2007), Signer and Norrie present a metamodel for hypermedia systems. This was done because few hypermedia systems were using the established principles of metamodeling from the database community. The platform described in the paper aims to support various categories of hypermedia systems. This metamodel is called the resource-selector-link model, which will be shortly called RSL model in the remainder of this thesis

(Signer & Norrie, 2007).

The RSL model consists of a core metamodel and some elaborations, which will be shortly described below. Note that the RSL metamodel was defined using the semantic, object-oriented data model OM (Norrie, 1993). A detailed explanation can be found in the paper itself (Signer & Norrie, 2007).

The first part is the core link metamodel, which describes the different entities: links, selectors and resources. A resource is an information unit that functions as an abstract concept for all the types of media that can exist in a model, which can be video, image, sound and so on. Because a user does not want to address a whole resource all the time, selectors are introduced so different parts of the resource can be distinguished. Different parts of an image can serve as a selector of the same resource, being the picture itself. A selector always refers to one single resource. A third kind of entity is a link, which has at least one target and one source, both being entities. A link can thus link between a selector and a resource, between two resources, between two selectors or can even have another link as source or target. Note that a concept of a link can serve more than only navigational links. More details on this will be given in the link metamodel below.

Entities have properties, which will be stored as an entity's property attribute. In this way, optimal flexibility for the designer of a hypermedia model based on the RSL metamodel is guaranteed. Note that entities also have context resolvers, which will return a boolean value for the accessibility of an entity based on contextual information.

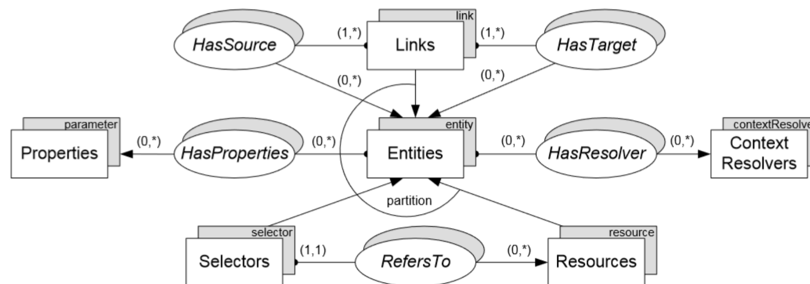

Figure 2.1: The core link metamodel

For users to be part of the platform, the user model is developed. In this metamodel (Figure 2.2), the relations between different types of users and the entities of the core metamodel are described. Users can have access to entities or can create entities. Users can refer to an individual or a group of users.

Figure 2.2: User management

As can be seen in the core metamodel, it is possible that a so-called nested link problem arises. This is the case when different selectors that define parts of the same resource overlap. One selector can specify a paragraph for example, while another one specifies a word within this chapter. The RSL metamodel resolves this issue by introducing the concept of layers, which is depicted in Figure 2.3.

Figure 2.3: Layers

A further elaboration of the link metamodel is done by distinguishing between structural links and navigational links (Figure 2.4). As earlier explained navigational links are used to guide a user from one entity to another, while structural links define a structure over entities, so that entities with similar content can be linked together for example.

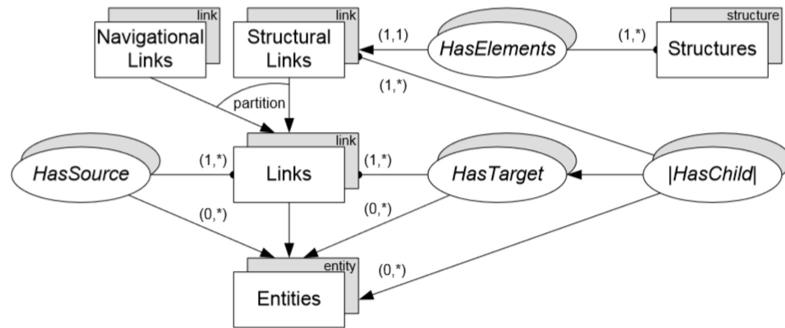

Figure 2.4: Navigational and structural links

2.3 iServer and EdFest

An implementation of the RSL metamodel is iServer, an RSL link service. It enables management functionality between different multimedia resources. The iServer platform has for example a plug-in for interactive paper (Signer, 2005).

This technology enables the development of applications such as EdFest. This application was made for Edinburh Fringe, a cultural festival in Edinburgh, Scotland. It is the world's largest arts festival and takes place every August. This three-week festival has more than 1700 shows at approximately 250 locations (Edinburgh Festival Fringe Society, 2016). To guide all the tourists around the city, various forms of printed information was distributed. This context was ideal to test the interactive paper plug-in of iServer. To test this, the tourist were also handed a Nokia Digital Pen, headphones and a GPS module.

When a tourist would take the map and click on a specific venue with the pen, an audio fragment started to give all the necessary information about this venue, such as the shows that will take place at this venue. Next to venues, different parts on the map were defined to give other audio results. In terms of the RSL metamodel, the map was an implementation of the concept of a resource, while the different venues on the map are the selectors. When clicked on a certain selector, a navigational link would evoke the corresponding audio fragment, which is an entity itself. Another example are the audiofragments for directions to a specific venue. Thanks to the GPS module, the location of the tourists could be determined, making it possible to guide the tourists to the venue they had clicked on. The directions to this venue are of course dependent of the tourists current location. Also other functionality was available, but these two examples suffice to grasp the idea

of the EdFest application.

As mentioned before, iServer was the technology that made this application possible. iServer is based on extending an object-oriented database management system. However, it is not proven that an object-oriented database is the most suited database for applications like this.

One can imagine that the data for this application is strongly connected. It is basically a network of entities linked to each other. This raises the question which database handles this kind of the data best. With the rise of the NoSQL movement, other kinds of databases than the relational ones gain interest. Of course, a graph database pops up first when it comes to handling strongly connected data. Before investigating the use of graph databases as back end storage solution for this application and for link services in general, the following chapter will first introduce the features of graph databases. Later on in this thesis, the strongly connected data from the EdFest application is imported in two commercially operational databases to investigate their use for this context. However, before doing so, an overview of existing graph database vendors is provided, a comparison is made and the different ways of modelling a graph database is investigated.

3
Graph Databases

In this chapter, an introduction to graph databases is given. Not only the features of this kind of databases are discussed, also its role in the NoSQL movement is explained. At the end of the chapter, some typical use cases for graph databases are explained.

3.1 Graph Databases and the NoSQL Movement

With the rise of NoSQL databases, the popularity of databases other than relational databases has grown. However, there were some non-relational databases around for quite a while, such as object oriented databases. These databases were kind of exotic nevertheless. The term 'NoSQL' does not mean in anyway 'against SQL'. It is broad term that wants to raise awareness that there are other storage solutions next to relational databases, which mostly use SQL, and that relational databases should not always be the default solution to store data. Some storage problems are better tackled with other databases than the relational ones. Besides this, the term 'NoSQL' was nothing more than a fancy marketing thing that would be used for a conference about new storage technologies, organised by Johan Oskarsson in 2009. Erik Evans from Rackspace came up with this term, also because it

would fit nicely as a hashtag on Twitter (Vasiliev, 2013). NoSQL is thus an umbrella term used to name all databases that are not using SQL. However, in this NoSQL container, 4 kinds of databases are typically distinguished (Fowler, 2012).

A first category are the *key-value databases*. These databases simply work with pairs of keys and values. In these values, all kinds of (unstructured) data can be stored (McCreary & Kelly, 2014). The most known key-value stores are Redis, Memcached and Riak (DB-Engines, 2016).

A second category are the so called *column family databases*. This kind of database is an extension of a simple key-value store in that sense that the key in a column family database is more complex and can have multiple elements. A simple example could be spreadsheet, where a specific cell with a value has a row-index and a column-index. In a column family database, this could be extended to a lot more dimensions (McCreary & Kelly, 2014). The most well-known column family store is Google's BigTable (Rahien, 2010).

Document stores make up the third category. This is another extension of the simple key-value stores, but this time it is the value that will have more possibilities to work with. The value of a document store is mostly a JSON document or an XML document. The value is thus more structured than in key-value stores. The benefit of working with JSON or XML documents is that these documents are searchable and that it is not necessary to return the whole value/document each time. Also parts of the documents can be returned, based on the query (McCreary & Kelly, 2014). MongoDB is by far the most well-known vendor of document databases (DB-Engines, 2016).

These three categories, that were discussed shortly above, are all aggregate-oriented databases. There is more de-normalisation than in relational databases. The values (or documents) are aggregates. These kind of aggregate-oriented databases are mainly designed for handling big data, for being partitioned and to be performant in terms of look-up speed (Fowler, 2012). The fourth category that is typically distinguished are *graph databases*. This kind of database is not aggregate-oriented, but is designed to handle strongly related data. Otherwise than the name would expect, relational databases are not that good in handling related data, because of the expensive joins that have to be carried out when querying these databases (Robinson, Webber, & Eifrem, 2010).

For the purpose of this thesis, graph databases seem to be the best choice to handle the strongly related data from hypermedia systems. Therefore, graph databases will be discussed in more detail in the next parts.

3.2 Features of Graph Databases

Graph databases are based on the graph theory from mathematics, founded by Leonard Euler. Graph databases are developed with the intention to store strongly related data. These kind of databases claim that their way of storing data is way more performant than relational databases for dealing with relationships, in contrast to what the name would suggest (Webber, 2013).

Graph databases explicitly show relationships (edges) between several nodes (vertices). Different sources may use a different terminology, but a node is the same as a vertex, and so are an edge and a relationship. A node represents an instance of an object, that can be connected to other nodes via edges. Both nodes and edges can have properties. Edges can have a direction, and more specifically, non-directed, self-directed and two-sided directions, for reciprocal relationships, are also possible (Robinson et al., 2010). Figure 3.1 gives a very simple example of a graph of a chess club with two members. There are nodes for members, where Alice and Bob are stored with their age. Furthermore, there is also a node for the group they are member of, the Chess group. In the relationships that represent a membership, the start date of the membership is stored as a property. Notice that there are also outgoing relationships for the Chess Group to its members. In this graph, also the relationship between the members is shown.

In relational databases relations are not stored explicitly. The relationships only exist in two keys that are preserved in the two subjects of the relationship. The database itself thus does not know about any relationship (Robinson et al., 2010). In order to find potential relationships a join operation needs to be performed which will attempt to combine rows in the tables based on specific conditions (depending on the type of join). In this case matching the two keys would be used to find information that is associated with each other. However, to find all the matching keys for a specific row, it has to be compared with all rows from the other table which is of course a very expensive operation.

The explicit storage of relationships brings some great advantages. When a new relation is created, there is no need to change anything in the related nodes itself, because of the strict separation between node and edge (Robinson et al., 2010). Furthermore, graph databases are way faster for queries that focus on relationships. This is due to the explicitly defined relations and to the index-free adjacencies, which means that there is no lookup of indexes needed because every node knows with which other nodes it is connected via its edges (Merenyi, 2013).

Therefore, graph databases are really good at doing graph traversals:

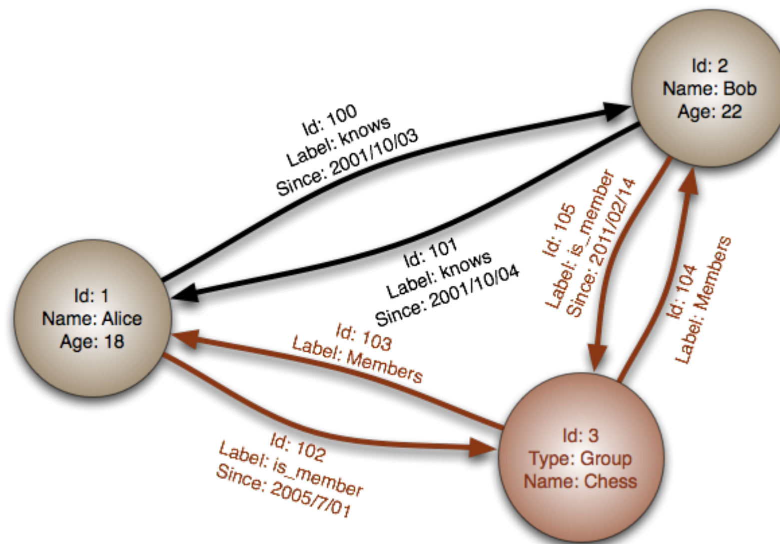

Figure 3.1: Graph example

Source: (GraphDatabase, 2016)

walking over the graph from node to node. This is confirmed by research from Partner and Vukotic (2012). In their book 'Neo4j in action' (Partner & Vukotic, 2012) they have set up an experiment with a fictional social network of 1 million users. On average, every user has 50 friends. This case was implemented in a relational database (MySQL[1]) and in a graph database (Neo4j[2]). The preliminary test was to acquire all the friends of friends of all the users in the database. Afterwards, all the friends of friends of friends (depth 3) were asked for. This was repeated in the same way for a depth of 4 and a depth of 5 friend relationships. The results are shown in Table 3.2.

As one can see, the difference in speed for the friends of friends (depth 2) is not that big. However, when digging deeper in the social network, the MySQL implementation lags behind. Already when friends with a depth of 3 is required, the Neo4j implementation is about 180 times as fast. With a depth of 4, it takes the relational database over 25 minutes (1543 seconds) to execute the query, while the execution was not finished within 1 hour for a depth of 5 friendship relation. The graph database on the contrary remains quite fast in executing these complex queries. This huge difference can be explained by the way both databases store these strong interrelated data.

[1] `https://www.mysql.com/`
[2] `http://neo4j.com/`

Depth	Execution Time – seconds MySQL	Execution Time – seconds Neo4j
2	0.016	0.010
3	30.267	0.168
4	1,543.505	1.359
5	Not Finished in 1 Hour	2.132

Figure 3.2: Comparison of query execution of relational database and graph database

Source: Partner, J., Vukotic, A. (2012). Neo4j in action

While the relational database needs a lot of expensive joins to do the job, the graph database can easily traverse a graph with explicit relationships and index-free adjacencies (Partner & Vukotic, 2012).

One could argue that object databases are also a good fit for connected data, thanks to the use of pointers to refer from one object to another. This makes that expensive joins are out of the question. However, this index-free adjacency is not the only requirement to handle strongly connected data well. First of all, the pointers in object databases are lightweight, meaning that they cannot hold properties or have a type. Furthermore, object databases are not developed with the idea of graph traversing, making the functionality to do so less extensive. Graph database on the contrary come with built-in functionality such as, for example, different shortest path algorithms. A third argument against object databases in this use case is that the database is very application-dependent. This means that every application will preferably need its own object database, making the compatibility between data in these databases a lot harder to handle.

Despite the advantages graph databases bring in a lot of different use cases, there are also some drawbacks about this kind of databases. Because graph databases are designed to store strongly interconnected data, it is not straightforward to have a good solution to partition the database, which means that the database is split over different physical locations.However this is not impossible in theory, it would raise a lot of performance issues. One could state that, in terms of the CAP theorem[3] graph databases are most often designed in favor of consistency and availability, while they are

[3]The CAP theorem, put forward by Eric Brewer (2000) and later proven by Seth Gilbert and Nancy Lynch, states that a computer system can only have two of the following three properties at the same time: Consistency, Availability and Partitioning tolerance (Brewer, 2000; Gilbert & Lynch, 2002)

not partitioning tolerant (Fowler & Sadalage, 2013).

To summarise, one could state that graph databases are interesting data storage solutions for different use cases. Even though vendors have various implementations with various variations on these concepts, the advantages of graph databases are mainly because of the explicit relationships, the index-free adjacency, the lack of need for joins, the traversal performance, the natural normalisation because they contain specific instances instead of classes or archetypes and the natural and direct representation of the domain (Grzegorczyk, 2014).

In the next part, some interesting use cases for graph databases are explored.

3.3 Use Cases

As graph databases are good at finding relations and finding patterns, commercial applications of these kind of databases are widely found in recommendation engines. Thanks to graph databases, some suggestions or recommendations can be shown based on previous purchases, related products or friends that bought similar products. This is the way to go for individual advertisement. This is not only done for products, but also for videos on YouTube or music on Spotify. Graph databases are an ideal fit for this kind of applications because they support inductive and suggestive algorithms, in contrast to more deductive and more precise calculations used by applications such as payroll management and tax calculation. Graph databases are designed to traverse graphs and to discover new relationships (Robinson et al., 2010). The possibility to generate recommendations goes even further when it's combined with geospatial data. Suggestions can be then more refined, based on the location of a person and interesting things around their location (Fowler & Sadalage, 2013).

Also in logistics there are numerous use cases thinkable. For example, a shortest route search engine for trains or other transportation means. Stations will be nodes, and the rails between stations would be represented by edges, where the travel time between different stations is persisted as a property of the edge. The same reasoning can of course be applied to other transportation means, where the cost can be represented not only by travel time, but also by fuel consumption or other variables (Fowler & Sadalage, 2013).

Another use case, which might be less obvious, can be found in anti-fraud applications. Graph databases are good at finding paths and patterns, which is impossible for humans when confronted with a huge number of nodes and

relationships. If bank accounts and bank transactions are inserted in a graph database as nodes and relationships between nodes, the graph database can be queried for patterns. In this way, so called rings can be discovered in the network. By further investigating these suspicious rings, money laundering might be detected. In real life, these applications and algorithms are of course more advanced than described here, but such kind of applications do exist (Guzenda & Quinn, 2013).

The ability to recognise patterns has many more use cases, such as making an in-depth market-analysis. The interweaving of different companies might become clear and unfair competition could be detected in this way. Graph databases have also already proven their benefits in health care applications. By combining symptoms, causes, treatments and the genetic background of a person, diseases can be dealt with much more efficiently (Guzenda & Quinn, 2013).

After having an introduction about graph databases in this chapter, the next chapter will be more focussed on products and vendors of graph databases that are on the market right now.

4 Comparison of Existing Graph Databases

The concept of graphs and graph databases, as explained in the previous chapter, has led to different graph database products by different vendors. To choose graph databases for evaluation, a quick market overview of available graph database products is given. Later on in this chapter, two databases will be selected and will be explored in more detail.

4.1 Overview

In Table 4.1 a table is shown with the main properties of different graph databases and RDF stores. Of course, not every single graph database and RDF store is considered, but only those with a minimum of momentum in the database world. This momentum is expressed by the number in the column 'Community'. The number in there gives an indication of the popularity of the database system. The scores are retrieved from DB-Engines[1] on November 5th 2015 and are calculated as described below. The value of these numbers, which will be called the community score, does not have a meaning as such, but are only used to compare database systems to each

[1] http://db-engines.com/

other. The other properties in the table should be clear by themselves.

Parameters for calculation (DB-Engines, 2016):

1. Number of mentions of the system on websites, measured as number of results on queries in search engines as Google and Bing.

2. General interest in the system, measured as frequency of searches in Google Trends.

3. Frequency of technical discussion about the system, measured as number of interested users on Stack Overflow and DBA Stack Exchange.

4. Number of job offers in which the system is mentioned, measured as the number offered op jobs on job search engines as Indeed and Simply Hired.

5. Number of profiles in professional networks in which the system is mentioned on LinkedIn.

6. Relevance in social networks, measured in the number of Tweets in which the system is mentioned.

   For comparison: Oracle's relational DBMS scored 1462 points in May 2016 while the most famous NoSQL document store, MongoDB, totals a score of 320 in this month (DB-Engines, 2016).

Database	Model	Comm scores (May 2016)	Free Edition	Visualization tools	Query languages	Partitioning methods	Property values
ArangoDB	Multi-model	1,56	yes	yes	AQL, Javascript, Gremlin	Sharding	JSON data types
InfiniteGraph	Graph	0,23	Only trial				
MarkLogic	Multi-model	9,07	Only trial	/	Java API, Node.js Client, ODBC, SPARQL	Sharding	JSON, XML, RDF, Geospatial, Text, and Binaries
Neo4j	Graph	32,61	yes	yes	Cypher, native Java, Gremlin	/	Primitives or an array of one
OrientDB	Multi-model	6,13	yes	yes	SQL dialect, Gremlin	Sharding	JSON data types, BLOBs
Sparksee	Graph	0,16	yes	Only by exporting the graph	Java, .NET, C++, Python and Objective-C through API, Gremlin	/	Primitives
Stardog	Triplestore	0,54	yes	/	SPARQL	/	RDF triples
Titan	Graph	4,8	yes	Third party tools as Keylines	Gremlin	Pluggable backends	Primitives

Figure 4.1: Overview of graph databases

4.2 The Different Database Systems

4.2.1 ArangoDB

ArangoDB is a multi-model database. It is both a graph database and a key-value store, as well as a document store, which means that it also supports JSON data types. With their version of the database (2.7.0) released in October in 2015, this database is very up to date. This open source database is available under Apache version 2 license. ArangoDB supports the most common programming languages, but prefers AQL as querying language, the ArangoDB Querying Language. The community score for the database is however quite low (1,56). (ArangoDB, 2015)

4.2.2 InfiniteGraph

InfiniteGraph is a distributed graph database. This database is not further examined due to the low community score (0,23) and the absence of an open source license. (Objectivity, 2015)

4.2.3 MarkLogic

The MarkLogic database system is a multi-model database, implying it can be categorised as a document store, an RDF-store and a native XML DBMS. With a community score of 9,07, MarkLogic is the second most popular database that is considered here. It supports all the most common programming languages and offers a wide range of query languages. Unfortunately, the database is not open source, there is only a trial version available for free. (MarkLogic, 2015)

4.2.4 Neo4j

Neo4j is a pure graph database and is well maintained with new versions coming out regularly. It is by far the most known graph database, with a community score of 32,61. There is an open source version available (GPL version 3). It supports the most common programming languages. The Neo4j database does not support partitioning nor is it possible to store other data types than primitives or arrays of primitives. This is due to the choice for speed and efficiency when traversing graphs. Although there are other query languages supported, Neo4j pushes the user towards its own developed query language called Cypher. (Neo4j, 2015)

4.2.5 OrientDB

OrientDB is a multi-model database, being a graph database and a document database at once. The latest update of the open source database was in October 2015 and the popularity of the database is growing, as shown by the 6,13 community score. Furthermore, it supports all the main programming languages. As a querying language, it claims to use a SQL dialect called OSQL (OrientDB SQL), and this claim does not seem to be protested yet. (OrientDB, 2015)

4.2.6 Sparksee

Sparksee is a product from Sparsity Technologies, a spin-off of the University of Catalunya to commercialise their graph database. Before its 5th release, in February 2014, Sparksee was called DEX. Sparksee is the first graph database available on mobile operating systems such as Android and iOS. Sparksee is a native graph database. It has however no own querying language, but provides an API for the most popular programming languages to query the graphs. With a community score of 0,16, Sparksee cannot be seen as a big player in the graph database market. (Sparksee, 2016)

4.2.7 Stardog

Stardog is essentially an RDF store, but it provides an implementation with a property graph. It supports all the main programming languages and uses an established query language for RDF stores, namely SPARQL. However, with a community score of 0,54, it is not (yet) a known name in the database world. (Stardog, 2015)

4.2.8 Titan

Titan is a distributed graph. The partitioning of the graph, however, depends on the pluggable backend, which can be Apache HBase[2], Apache Cassandra[3] or Oracle BerkeleyDB[4]. In fact, Titan can be considered more as a graph layer over the pluggable backend. Titan supports Java and Python, among some others, as programming languages. Titan is designed to handle thou-

[2] https://hbase.apache.org/

[3] http://cassandra.apache.org/

[4] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/

sands concurrent users that are executing complex graph traversals. (Think Aurelius, 2015)

4.3 Shortlist

Out of this list, there are 4 databases that are further considered, namely: ArangoDB, Neo4j, OrientDB and Sparksee. These four databases all have a free license, claim to be a real graph database instead of a triplestore or RDF store and they all have a minimum of momentum in the graph database world. However, this list with four databases will be slimmed down to only two databases due to the limited scope in time of this thesis. The two chosen databases will be further tested and real data will be inserted in these two databases for a comparative benchmark later in this thesis.

To choose two databases out of the four mentioned above, it could be interesting to compare a native graph database, such as Neo4j and DEX, with a multi-model database, such as ArangoDB and OrientDB. While one would expect a native graph database would be faster for querying graphs, it is claimed that multi-model databases can compete (and beat) native graph databases (Weinberger, 2015). In this respect, a first choice will be made between Neo4j and Sparksee, a second between ArangoDB and OrientDB.

Sparksee and Neo4j are both native graph databases, which means they are purposely built to deal with graphs. A benchmark by shows that both Neo4j and Sparksee are leading the pack as it comes to performance of graph databases (Dominguez-Sal et al., 2010). Another benchmark states that Neo4j however is faster than Sparksee (Jouili & Vansteenberghe, 2013). Another advantage of Neo4j over Sparksee is the querying language of Neo4, called Cypher. Neo4j aims to make Cypher the default query language for graph databases, as SQL is for relational databases. Cypher is purposely built to query graphs. In collaboration with Oracle and Spark, Neo4j set up the openCypher project to realise this goal (Eifrem, 2015). On top of this, Neo4j's community is way larger than Sparksee's. Also Gartner acknowledges this, by putting Neo4j as only native graph database in their Operational Database Magic Quadrant. This makes that Neo4j is considered as a challenger in the database market (Feinberg, Adrian, Heudecker, Ronthal, & Palanca, 2015).

The second choice that has to be made is between ArangoDB and OrientDB. Exisiting benchmarks are not unanimous about which of both is the fastest. ArangoDB's own benchmark states that ArangoDB is by far the fastest, even faster than Neo4j (Weinberger, 2015). OrientDB contests these result and claims OrientDB was wrongly implemented for this bench-

mark (Garulli, 2015). OrientDB points to other benchmarks to prove their database is faster, even 10 times as fast as Neo4j (OrientDB, 2013). Based on these benchmarks, no winner can be chosen. However, there are some other criteria to base a decision on, which are mainly in favor of OrientDB. OrientDB is the second most popular graph database around (Feinberg et al., 2015). Furthermore, OrientDB also claims to have the most suited query language for graph databases. Orient Technologies has developped an SQL dialect, called OSQL, especially for querying graph databases. Instead of creating "yet another query language", they extended the standard SQL language with some support for graph databases (OrientTechnologies, 2016). ArangoDB on the contrary gets less attention with their AQL query language. Despite claiming they are the fastest database around, the community does not seem to believe this, regarding their community score of 1,56. For the remainder of this thesis, OrientDB will be used as the second platform for performing some tests. However, the choice for OrientDB over ArangoDB cannot be made on benchmark figures and is debatable.

Having made the choice for Neo4j and OrientDB, graph models have to be made to use these graph databases. In the next chapter, there will be a discussion about different approaches to model graphs and more specifically, the RSL metamodel. Later on in this thesis, the RSL-based data from EdFest will be imported in Neo4j and OrientDB using the graph models proposed in the next chapter.

5 Mapping the RSL Model to Graph Databases

The data from the RSL-based EdFest model is of course not immediately ready to import in a graph database. First, the concepts that define an RSL model should be mapped into concepts from graph databases. Therefore, an extensive research on graph modelling was necessary. The most important concepts of graph models needed to support the EdFest application are explained here below.

5.1 Possible Mappings

5.1.1 Typing and Sub-typing

In the metamodel presented in the paper from Signer & Norrie (2007), one can see that sub-typing is an important component of the metamodel. For instance, when a developer is creating a hypermedia model for his application and wants to add images as a resource, his "Image" entity would be a sub-type of the "Resource" entity defined in the core link metamodel. This concept maps well to most graph databases. Notwithstanding Neo4j (labels) and OrientDB (classes) are using different names for it, their solution to implement types is quite similar. For the nodes can be given a label (or a class

in OrientDB), so that it is known what this node represents. Multiple labels (classes) can be given to a node, so that a hierarchy of labels can exist. In this way, typing and inheritance can be naturally achieved in graph databases.

An example of this approach is given in Figure 5.1 where a selector, "Selector1", refers to a resource, "Resource1". The node "Selector1" is of the type Selector, which on its turn is of the type Entity. The same holds for "Resource1" being a Resource. The 'being a type of' relationship is indicated using labels in Neo4j or classes in OrientDB. Notice that instead of using the types as they are used here, also jargon of the data concerned can be used. Moreover, it would also be possible to bring the data closer to the metamodel, by allowing the inheritance of types from concepts of the metamodel. Consider for example a picture named "Picture1". In this picture, a certain area is defined based on some coordinates, serving as a selector. The node that represents this selector could be labeled with the labels PictureArea, Selector and Entity. The picture itself would have the labels Picture, Resource and Entity. In this way, the link between the model and the RSL metamodel becomes more clear.

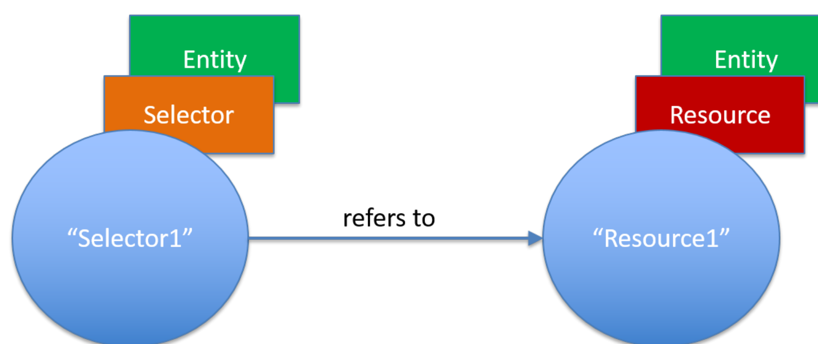

Figure 5.1: Modelling inheritance in graph models using labels (Neo4j) or classes (OrientDB)

Another, more artificial, solution could exist in a way that a node is created for every concept in the database. All the nodes that are a type of this concept should then link to this node. This solution brings some overhead, but has as an advantage that it is possible to store information about the concept itself, which is not possible if the concept is modelled as a label (class). This approach is shown in Figure 5.2. Because this solution will give too much overhead and will not create substantial advantages, it will not be implemented and tested, however it is theoretically feasible.

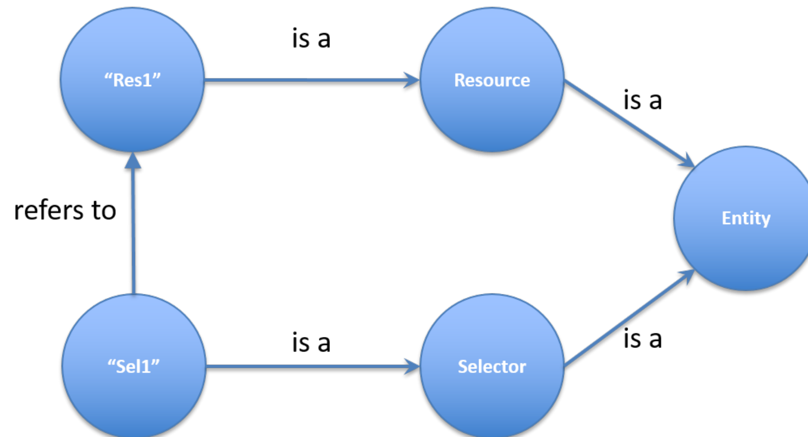

Figure 5.2: Modelling inheritance in graph models using other nodes

5.1.2 Links

The RSL paper also explicitly defines links in their metamodel. This is exactly what graph databases also aim to do: making links (relationships) first class citizens. It is perfectly possible to store features or properties on relationships for example. This approach is shown in Figure 5.3.

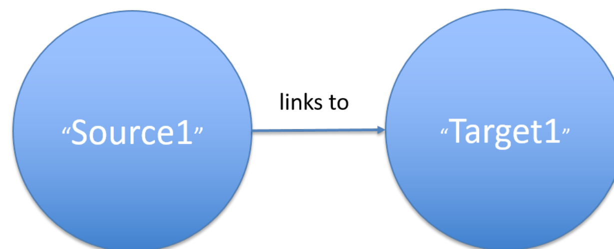

Figure 5.3: Modelling links as edges

However, there are also some discrepancies between the links from the metamodel and the relationships in graph models. The most important one is that, in the metamodel, links can link to other links. In most graph databases on the other hand, this is not possible. To make it possible for links to link to other links, a link should be modelled as an intermediate node between two other nodes, as shown in Figure 5.4.

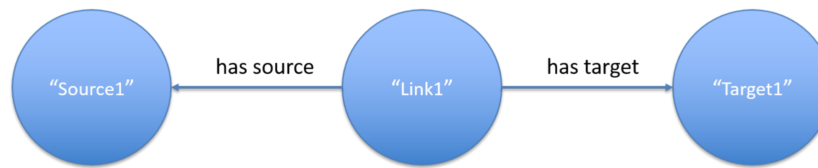

Figure 5.4: Modelling links as nodes

To check whether the extra functionality of the second approach slows down the querying speed, both approaches will be considered and tested.

5.1.3 Blobs (Binary Large Objects)

When talking about hypermedia systems, a storage solution for blobs such as images, videos or audio fragments, should be discussed. If blobs are stored in a database, this database should have a big storage capacity due to possibly numerous and large files. In theory it is not impossible to store blobs directly into graph databases. However, this is not the best idea to do, since graph databases are not designed to be partitioned over multiple servers. The database might then reach its full capacity soon, without having the option to install another database next to it to solve this problem. When confronted with big data, this is not a future proof solution. Therefore, this solution will not be implemented and tested in graph databases.

A better way to handle this, is to only store references to blobs. The values of these blobs can then be stored in databases that are designed to store large amounts of data and that are designed to be partitioned, such as the key-value stores that were shortly discussed above. The implementation of these key-value stores is out of the scope of this thesis though. Thus, the reference to a blob will be stored in the graph database, while the retrieval of the blob in the key-value store will not.

5.1.4 Entity Properties

As can be seen in the RSL model the properties of entities are explicitly modelled to be as general as possible. This approach brings a lot of advantages, as discussed in the paper. If this would be implemented in a graph database, there should be a node for every property and a relationship between this node and the node that describes the entity. This approach brings some overhead and will likely cause some slower query performance. This approach is depicted on the left side of Figure 5.5.

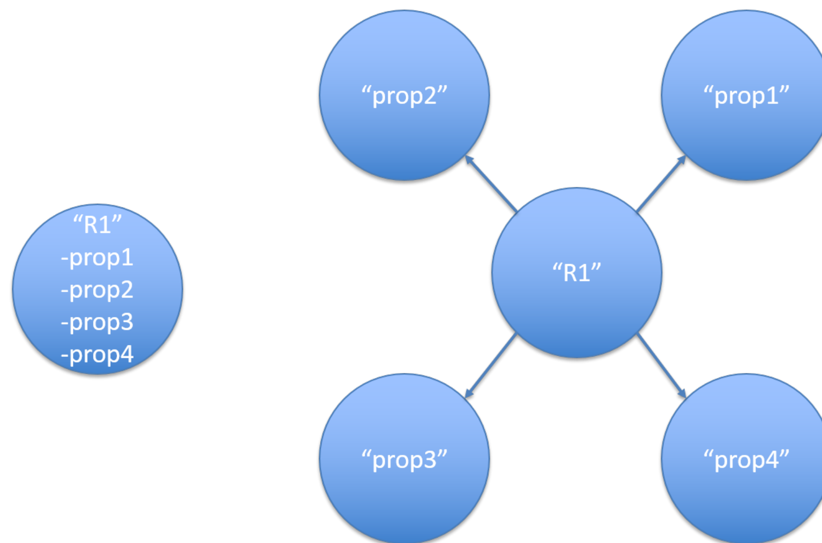

Figure 5.5: Modelling properties

On the right side of Figure 5.5 another approach is shown to store properties. Here, properties are stored just inside a node as an attribute of this node instead of making nodes for each property. This last approach differs from the metamodel in terms of functionality, but might bring some advantages in terms of performance.

There are advantages and disadvantages for both approaches, so both should be compared to each other.

5.2 Chosen Mappings

As discussed above, there are a lot of different theoretical options to design a graph model for a data model based on the RSL metamodel. As motivated above, the approach to store a blob directly into a node will not be considered. Concerning the options to implement typing and sub-typing, it is sure that the use of labels or classes is the most elegant way to solve this. The other option will not be implemented. When looking to modelling approaches for links, a link can be modelled as a node or as an edge. Both solutions should be further investigated to balance out functionality versus performance. The same holds for the two approaches regarding properties.

All this eventually leads to three different graph models that will be considered. The first model resembles the most the original RSL model: properties are modelled as separate nodes instead of attributes of an entity node and links are modelled as intermediary nodes between a selector and

a resource. In the remainder of this thesis, this model will be referred to as the model 'with explicit properties'. Please not that this also includes that links are explicitly modelled as nodes.

Because the functionality advantages of the explicit modelling of properties and links can be argued, two alternative models are proposed for testing. The first one, which will be called 'with explicit links' from now on, is a model where the properties are modelled as attributes of an entity node, while links are still explicitly modelled as nodes. A third approach, 'without explicit links', models links as edges, giving up the functionality of linking to links. Whether these two alternative models will bring the hoped performance improvements will be investigated in Chapter 7. Before doing so, Chapter 6 will discuss how the data from EdFest is imported in the graph models that were proposed in this chapter. Notice that the data is imported in the three models in Neo4j and also in OrientDB, resulting in six different database implementations:

1. With explicit properties in Neo4j[1]
2. With explicit links in Neo4j
3. Without explicit links in Neo4j
4. With explicit properties in OrientDB[2]
5. With explicit links in OrientDB
6. Without explicit links in OrientDB

[1]Note that the model with explicit properties also has explicit links
[2]Note that the model with explicit properties also has explicit links

6 Importing Data in Neo4j and OrientDB

In Chapter 2, the context of the data was described as being data from EdFest, an application for a cultural festival in Edinburgh. In Chapter 5, different mapping solutions were proposed to model this data in graph models. In this chapter, there will be a more detailed look to the data from EdFest and it will be shown how this data will fit in the given graph data models. In the last part of this chapter, there will be a brief explanation of how the data is imported in Neo4j and OrientDB, the chosen graph databases from Chapter 4.

6.1 The Data

The data mainly consists of sources that have links to targets. While a link just has a name, a source has multiple attributes such as the width, height and x and y coordinates of the source on a specific page. This page is also defined in the model and has a name and a pagenumber as attributes. The page on its turn, is linked to a document, which has a name, an id, width, height and content as attributes. A source also has a layer, with as attribute just the layer's name.

These sources are linked to targets. There are three different kind of

targets, which all have their own kind of attributes. Remark that this is definitely not a problem to store in graph databases as these databases can exist without schemas. The three different targets are called OMSWETargets (with attributes: name, description, and content), Active Component Targets (with attributes: name, identifier, uri, timeout and data) and Map Button Targets (with attributes: name, identifier, uri, timeout, data, captureShape and fixUri). All these elements are created by a so called creator, with a name, a description, a login and a password.

It can be clearly seen that this data is based on the RSL metamodel. From the core link metamodel (Figure 2.1) one can learn that a source is a selector that is linked, via a link, to a target resource. Each of these three entities has properties and is created by a user, which is in this case an individual and not a group of users (see Figure 2.2). Also the concept of layers is expressed in the given data.

6.2 The Data in a Graph Model

As discussed in Chapter 5, there are three different models that will be used to import the data to a graph database. The first one resembles the most to the RSL metamodel and is shown below.

The most important thing is that all the properties are modelled as different nodes (all the small grey nodes). The nodes with the name of another node are the green nodes. The entity nodes themselves are empty, but are connected to their properties. In this figure, the yellow node is the creator, the blue node is the link that links the purple source to the red target. The source has a layer (grey node under the source) and a page (grey node above the source). The page also has a document, which is the pink node on top of the graph. Note also that a link is modelled here as a node, which means it can link to other links as well. As said earlier, there will be referred to this graph as the model 'with explicit properties', since properties are explicitly modelled as nodes here. Notice that also links are modelled as nodes in this model.

A first alternative is shown in Figure 6.2. In this model, the attributes of an entity are no longer modelled as different nodes, but are now the properties of an entity and are thus stored inside each entity. The links are still modelled explicitly as nodes, for the advantages this entails. As mentioned in Chapter 5, this model is called the model 'with explicit links'.

The last and most simple model that will be considered is shown in Figure 6.3. The links are now represented by edges. This brings as a disadvantage that a link can no longer link to other links. Some functionality is lost

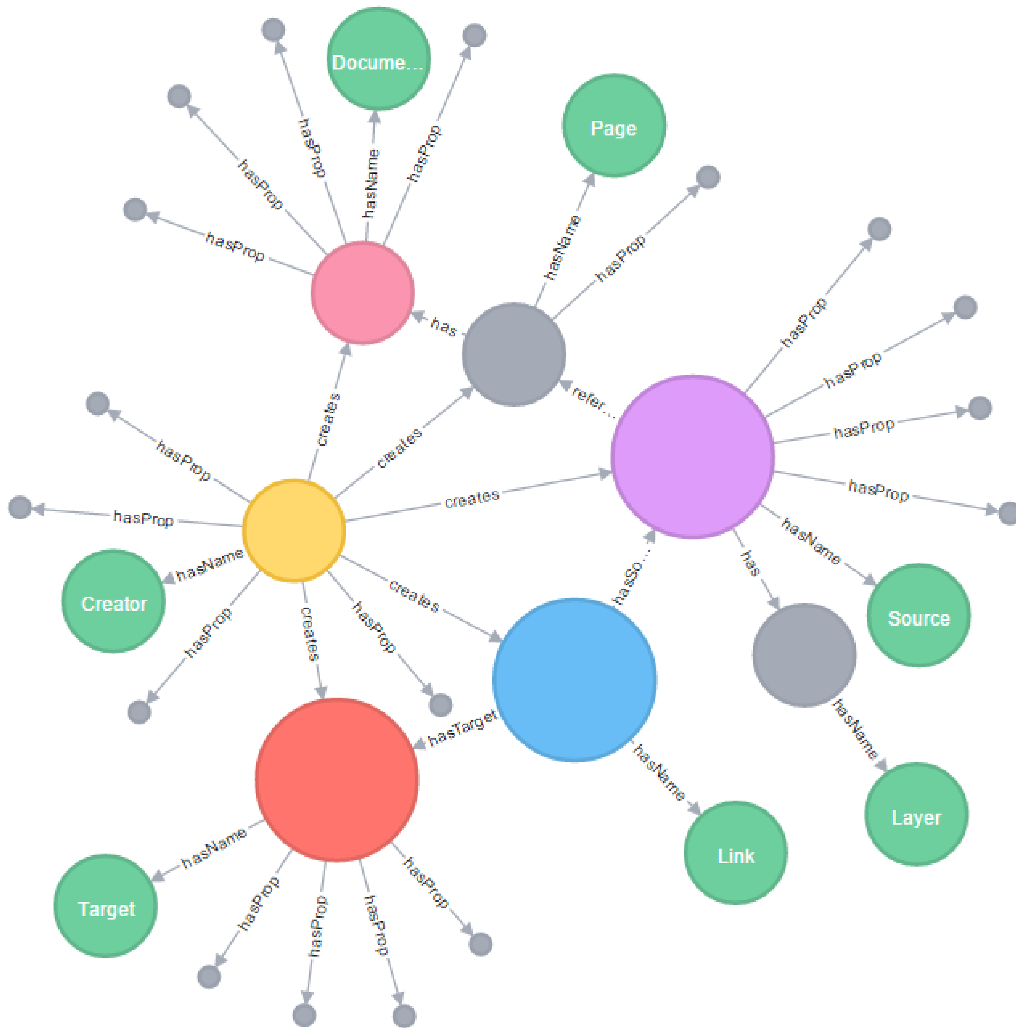

Figure 6.1: Graph model 'with explicit properties'

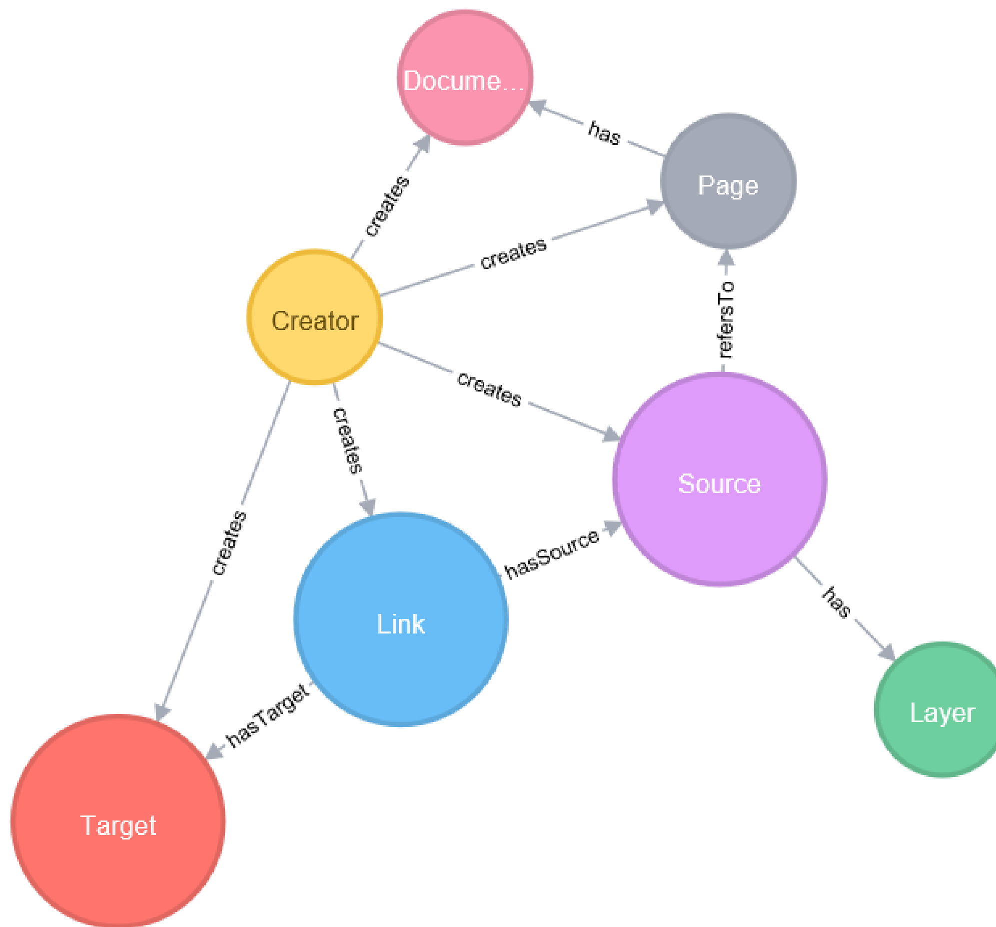

Figure 6.2: Graph model 'with explicit links'

here, but this might improve the performance. The attributes of a link can be stored in the edges as properties. Note that it is not possible to have an edge between Creator and the link as an edge. Therefore, the creator (or a reference to it) of a link should be stored as a property in the given edge. To refer to this model, the naming 'without explicit links' is used.

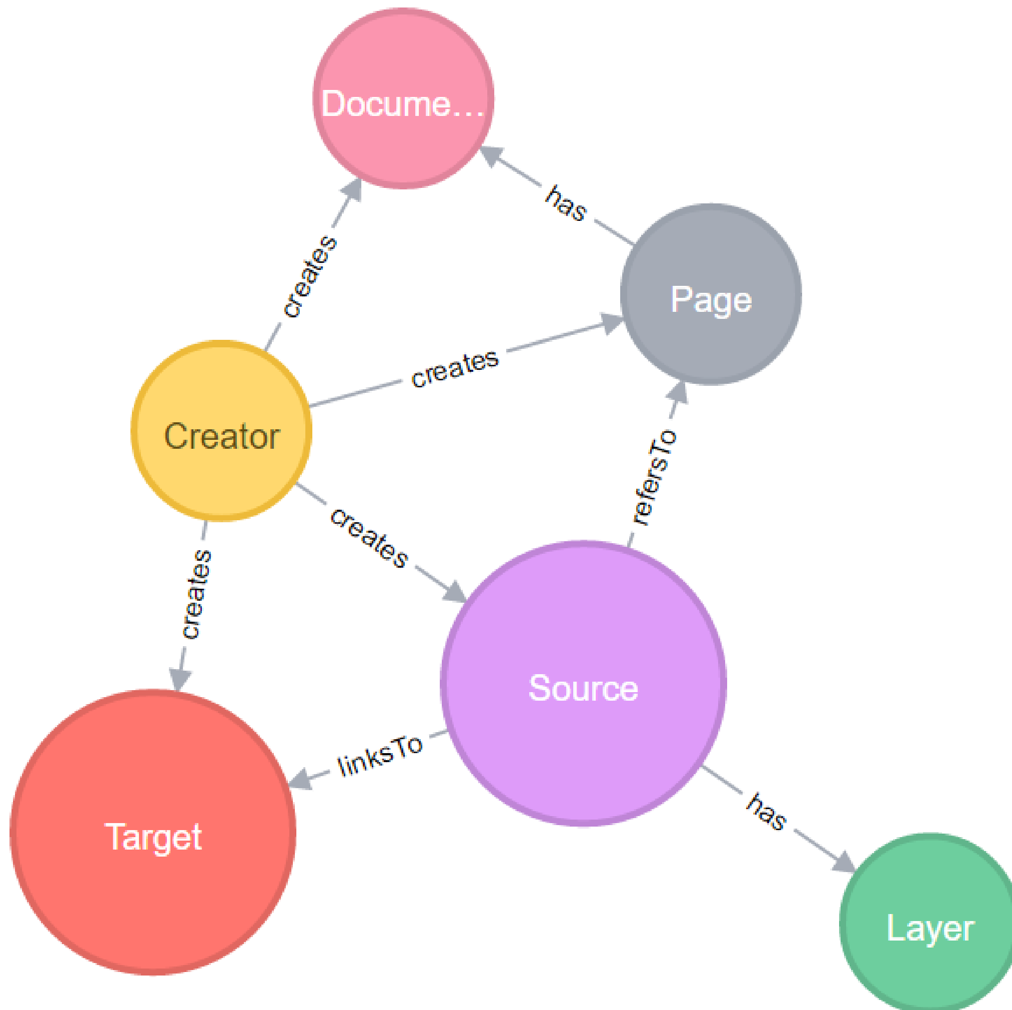

Figure 6.3: Graph model 'without explicit links'

6.3 Importing the Data

Once the design decisions are made and the data is mapped to these graph models, the next step that has to be taken is the import of the data in the given graph databases according to the different models. A brief explanation on how to import the data, which was originally given as an XML-file, is given below.

6.3.1 Neo4j

It is not possible to import the XML-file directly into Neo4j. Therefore, the XML-file was converted to a csv-file. It was then possible to use the 'Load CSV' command in Cypher, the query language from Neo4j. This command will parse each line of the csv-file and every value of each line can be used to import it in the Neo4j database. An example is given below in Cypher:

```
LOAD CSV WITH HEADERS FROM
'file:///C:/Festivaldata.csv' AS line
CREATE (link:Link {name: line.name})
```

The csv file is loaded from the given location and is read line by line. It is then possible to create a link for example. The name property will have the value 'line.name', which is the value of the name variable on this line in the csv-file. In this way, all the other values are also imported in Neo4j. The full code to do so, for the three different models, is given in appendix A.

6.3.2 OrientDB

Once the data is imported to Neo4j, it is possible to export the database as a graphml-file[1], a special file format for graphs. This is done using the Neo4j shell tools[2]. The graphml-file can then be imported in an OrientDB database. The steps to do so are explained in the OrientDB manual[3]. In this way the exact same models are ready to be used as a Neo4j database and an OrientDB database. The six database instances are ready to be compared one to another in Chapter 7.

[1]All the graphml-files can be found in attachment to this thesis
[2]https://github.com/jexp/neo4j-shell-tools
[3]http://orientdb.com/docs/2.1/Import-from-Neo4j-into-OrientDB.html

7 Benchmarking Neo4j and OrientDB

To test the design choices detailed in Chapter 6, different types of queries will be run against the different models and different databases. In total there are six database instances: two brands of databases (Neo4j and OrientDB), and three models per database. The first model, 'with explicit properties', has both explicit properties and explicit links. The more simple model, 'with explicit links', does still have explicit links but the properties are modelled as attributes within nodes. The last and most simple model, 'without explicit links', is modelled in such a way that links are modelled as edges and that properties are still modelled as attributes within nodes.

The queries that will be run against the different database instances can be divided in three types: read-queries, write-queries and traverse-queries. The traverse-queries are typical for graph databases, since this is the only type of database you can really traverse. For every kind of query, the results will be shown in this chapter.

All the tests were performed on an HP ENVY laptop with an Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz (8 CPUs) processor and 12GB of RAM memory. The used Neo4j database is Neo4j 2.3.3 community edition, while the OrientDB version is OrientDB 2.2.0 community edition. Every query was performed 100 times. In all the graphs below, the average query

times are shown. The full output of all the queries can be found as a separate attachment of this document.

7.1 Read Performance

7.1.1 Read Whole Database

In a first query, the database was asked to return all the nodes and relationships with their related properties. Of course, the different graph models have a different amount of nodes and relationships, which will most likely influence the query speed. The queries in Cypher and OSQL are shown below.

In Cypher:

```
Match n
Return n
\label{firstqueryneo}
```

In OSQL:

```
Select * from V
```

Figure 7.1 gives the average query times for each database instance to return the whole database. It is in line with expectations that the database instance with explicit properties takes longer to return everything, since it has to return more. In can be seen in the outputs that the databases ‘with explicit properties’ return 9732 nodes, while the databases ‘with explicit links’ have to return 2850 nodes. The simplest model, ‘without explicit properties’, only has to return 1907 nodes. These findings hold for both Neo4j and OrientDB. However, the difference in query speed between OrientDB and Neo4j is noticable: while OrientDB returns everything in between 14 and 60 milliseconds, depending on the graph model, Neo4j takes around 40 times as long. In the simplest model, ‘without explicit links’, it takes Neo4j already over half a second, while the graph model ‘with explicit properties’, takes almost 2,5 seconds on average.

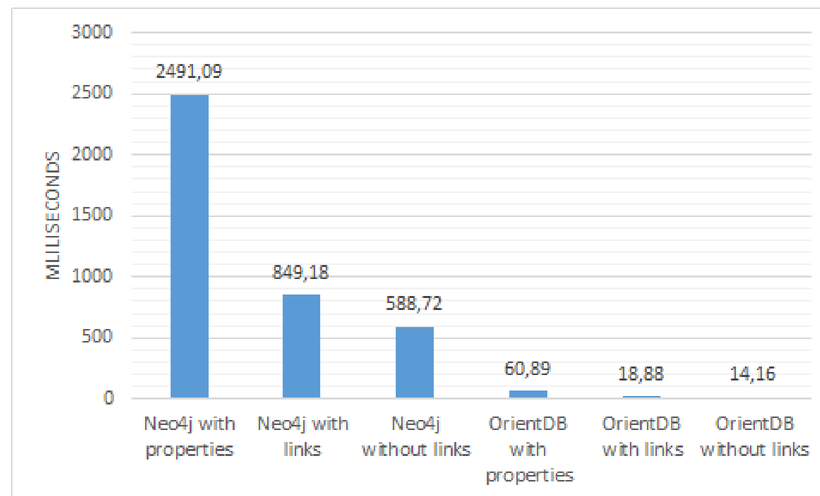

Figure 7.1: Result of query in section 7.1.1

7.1.2 Read Entity with Specific Name

This query aims to look up a Source with a specific name: 'Map Help Shape'. In the graph models 'with explicit properties', this name is not stored as a property in a Source node, which will be likely to slow down the query speed because a traversal to a node containing the name will have to be made. The different queries that are used to retrieve the Source with name 'Map Help Shape' are given below for the different databases.

In Cypher 'with explicit links' and 'without explicit links':

```
Match (s:Source)
Where s.name="Map Help Shape"
Return s
```

In Cypher 'with explicit properties':

```
Match (s:Source)--(n:Name{name:"Map Help Shape"})
Return s,n
```

In OSQL 'with explicit links' and 'without explicit links':

```
Select * from Source where name="Map Help Shape"
```

In OSQL 'with explicit properties':

```
select from Source where both('hasName')[@class='Name'].name="Map Help Shape
  ";
```

Figure 7.2 shows the average times in milliseconds to look up the Source with name 'Map Help Shape'. As one could expect, the databases 'with

explicit properties' perform slower than the databases that have stored the properties of an entity within the entity node. Especially the 'with explicit properties' model in OrientDB takes a long time to return the requested Source. Furthermore, one can see that on average, Neo4j performs this query faster than OrientDB, also for the databases without explicit properties.

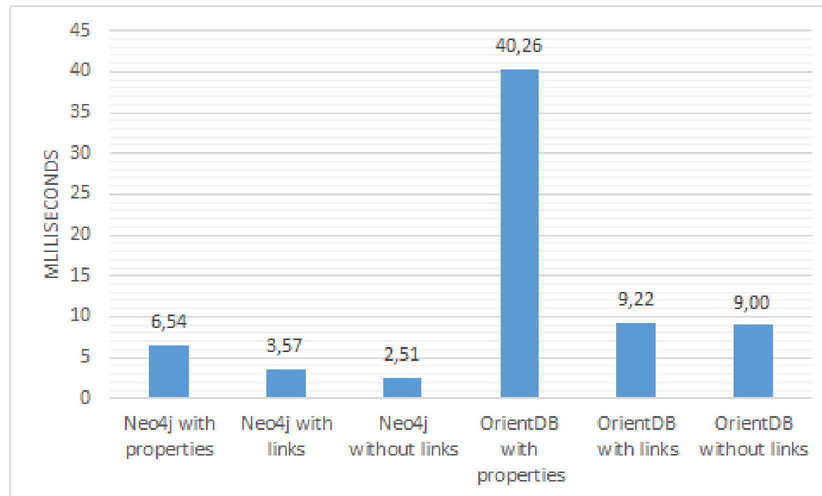

Figure 7.2: Result of query in section 7.1.2

7.1.3 Read Entity Who's Name Starts with Specific Character

The query that is tested here resembles the query in the previous part. However, instead of just returning one Source with a specific name, the database should now return all the Sources which name starts with the letter H. There are 15 Sources that have a name starting with 'H'. All the databases return all these 15 nodes, but the time it takes to perform the query differs of course. The different queries are stated below.

In Cypher 'with explicit links' and 'without explicit links':

```
Match (s:Source)
Where s.name starts with ('H')
Return s;
```

In Cypher 'with explicit properties':

```
Match (s:Source)--(na:Name)
Where na.name starts with ('H')
Return s;
```

In OSQL 'with explicit links' and 'without explicit links':

```
Select * from Source where name like 'H%'
```

In OSQL with explicit properties:

```
Select from Source where both('hasName')[@class='Name'].name like "H%"
```

As the query resembles the previous query, where a Source with a specific name was requested, one could expect similar results. The average query times are shown in Figure 7.3. In both Neo4j and OrientDB the more complex model, 'with explicit properties', takes longer to process the query. However, the slow performance of OrientDB database 'with explicit properties' is again remarkable.

In contrast to the query where a Source with a specific name was asked, Neo4j does not outperform OrientDB when it has to return all the Sources who's name starts with an 'H'.

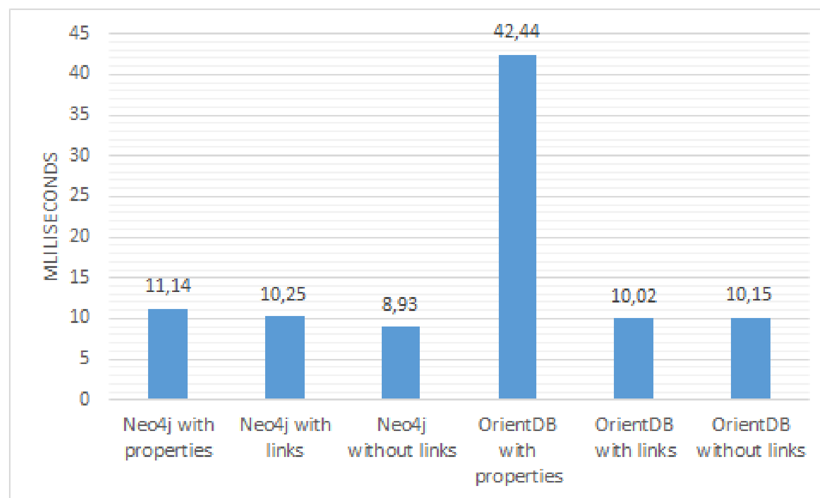

Figure 7.3: Result of query in section 7.1.3

7.1.4 Read All Links with their Sources and Targets

This query intends to have all the links with their Sources and Targets returned. If there are no explicit links, the database should just return the Source and the Target and the interconnecting edge. If there a link is explicitly modelled as a node, the database should return three nodes (Source, Link, Target) and two edges (between Source and Link and between Link and Target). This means that the models 'with explicit properties' and 'with explicit links' should return 943 instances with each having 3 nodes (a source, a link and a target), which totals 2829 nodes that have to be returned. 'With-

out explicit links' should only return 1886 nodes. The different queries for the different models in Cypher and OSQL are given below.

In Cypher 'with explicit links' and 'with explicit properties':

```
Match (s:Source)--(l:Link)--(t:Target)
Return s,l,t
```

In Cypher 'without explicit links':

```
Match (s:Source)--(t:Target)
Return s,t
```

In OSQL 'with explicit links' and 'with explicit properties':

```
Select from (traverse both('hasTarget'),both('hasSource') from Link);
```

In OSQL 'without explicit links':

```
Select from (traverse both('linksTo') from Source);
```

The average query time is indeed lower in both Neo4j and OrientDB for the graph model 'without explicit links' in comparison to the model 'with explicit links'. This can be explained by the fact that the latter has to return more nodes than the first one. A result which might be unexpected, is the good speed performance of the Neo4j database 'with explicit properties'. Although this database should also return 2829 nodes instead of the 1886 of the database 'without explicit links', the database returns the demanded nodes faster. This can be explained by having a deeper look into what the database exactly returns. The nodes in the model 'with explicit properties' do not contain properties, since these are modelled as different nodes. The returned nodes for this query are thus lightweight, without any properties, which makes returning them easier and faster.

Another remarkable result is that OrientDB outperforms Neo4j for this test. Just like in section 7.1.1, Neo4j is a lot slower when it has to return a lot of information.

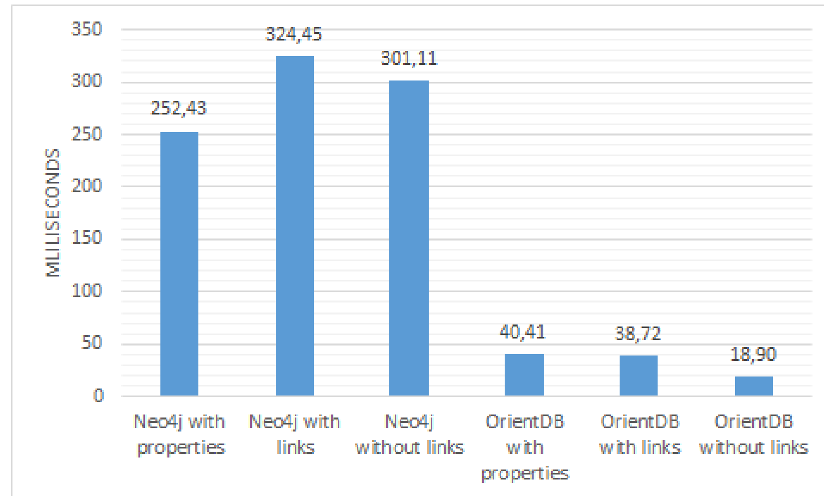

Figure 7.4: Result of query in section 7.1.4

7.1.5 Conclusion Read Performance

Concerning the read performance, three important conclusions can be drawn. Firstly, the graph models 'with explicit properties' are slower than the graph models 'with explicit links', which are on their turn slower than the models 'without explicit links'. This comes of course as no surprise. This slower query performance should be weighed against the loss of functionality in every different case.

A second conclusion is, as shown in Figures 7.1.1 and 7.1.4, is that Neo4j is terribly slower than OrientDB when it has to return a lot of records. Both for returning all the nodes and relationships (Figure 7.1) and for returning all the links with their sources and links (Figure 7.4) Neo4j performs multiple times slower.

Another lesson learned about the read performance of both databases and the different graph models, is the slow query read performance of OrientDB when it has to traverse the graph to execute the query. In Figure 7.2 and Figure 7.3 the slow query performance is striking. To execute these queries, the edge hasName between the entity node Source and the property node Name has to be traversed every time. Apparently, OrientDB needs a lot of time to do so in comparison to Neo4j, which handles this way faster.

7.2 Write Performance

7.2.1 Creating a New Entity

This query will make a new Source with 5 properties, among them the name of the Source. For simplicity reasons, all the values of the properties are named ‘test’. The queries to do so are listed below. Notice that 6 nodes and 5 relationships should be created in each query for the databases ‘with explicit properties’, since every property is modelled as a different node.

In Cypher ‘with explicit properties’:

```
Create (s:Source)
Create (p1:Name{name:"test"})
Create (p2:Prop{prop:"test"})
Create (p3:Prop{prop:"test"})
Create (p4:Prop{prop:"test"})
Create (p5:Prop{prop:"test"})
Create (s)-[:hasName]->(p1)
Create (s)-[:hasProperty]->(p2)
Create (s)-[:hasProperty]->(p3)
Create (s)-[:hasProperty]->(p4)
Create (s)-[:hasProperty]->(p5)
Return s,p1,p2,p3,p4,p5
```

In Cypher ‘with explicit links’ and ‘without explicit links’:

```
Create (s:Source{name:"test",prop2:"test",prop3:"test",prop4:"test"})
return s
```

In OSQL ‘with explicit properties’:

```
Let s = Create vertex Source
Let n = Create vertex Name set name='newSource'
Let p1 = Create vertex prop1 set prop1='test'
Let p2 = Create vertex prop2 set prop2='test'
Let p3 = Create vertex prop3 set prop3='test'
Let p4 = Create vertex prop4 set prop4='test'
Let p5 = Create vertex prop5 set prop5='test'
Create edge hasName from $s to $n
Create edge hasProp from $s to $p1
Create edge hasProp from $s to $p2
Create edge hasProp from $s to $p3
Create edge hasProp from $s to $p4
Create edge hasProp from $s to $p5
```

In OSQL ‘with explicit links’ and ‘without explicit links’:

```
Create vertex Source set name='newSource', prop2='test', prop3='test', prop4
    ='test',prop5='test'
```

Figure 7.5 shows the average speed in milliseconds of the creation of a new Source with its properties. It does not come as a surprise that the databases that model the properties as nodes are slower than those databases that model the properties of a Source as attributes within the Source node. Remarkable is the fact that it takes way longer for the OrientDB database

'with explicit properties' in comparison to the OrientDB databases without these explicit links.

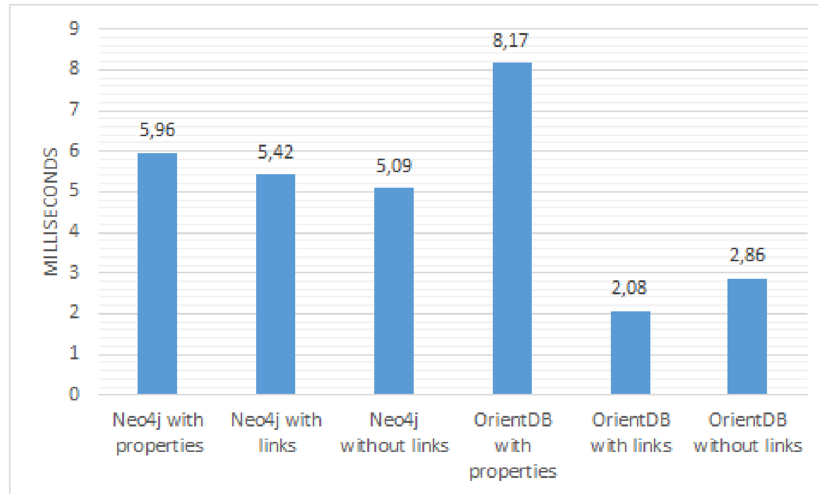

Figure 7.5: Result of query in section 7.2.1

7.2.2 Creating a New Link

This query will create a new link between a Source with name "Help Page Shape 2" and a Target with the name "Help Page Link 1". In a first part of the query, both the Target and the Source should be found, before creating a new link with a name property of the link. Depending on the model, this link is a node or a relationship. The name property will be modelled as another node or as a property of the link node, also depending on the graph model concerned.

In Cypher 'with explicit properties':

```
Match (n:Name{name:"Help Page Shape 2"})-[:hasName]-(s:Source)
Match (na:Name{name:"Help Page Link 1"})-[:hasName]-(t:Target)
Create (s)<-[:hasSource]-(l:Link)-[:hasTarget]->(t)
Create (name:Name{name:'test'})
Create (l)-[:hasName]->(name)
Return s,l,t,name
```

In Cypher 'with explicit links':

```
Match (s:Source{name:"Help Page Shape 2"}), (t:Target{name:"Help Page Link 1
    "})
Create (s)<-[:hasSource]-(l:Link{name:"test"})-[:hasTarget]->(t)
Return s,l,t
```

In Cypher 'without explicit links':

```
Match (s:Source{name:"Help_Page_Shape_2"}), (t:Target{name:"Help_Page_Link_1
    "})
Create (s)-[:linksTo{name:"test"}]->(t)
Return s,t
```

In OSQL 'with explicit properties':

```
Let l = Create vertex Link set name='newLink'
Let n = Create vertex Name set name='newLink'
Let s = Select expand(both()) from Name where name='Help_Page_Shape_2';
Let t = Select expand(both()) from Name where name='Help_Page_Link_1';
Create edge hasName from $l to $n
Create edge hasSource from $l to $s
Create edge hasTarget from $l to $t
```

In OSQL with explicit links:

```
Let l = Create vertex Link set name='newLink'
Let s = Select from Source where name='Help_Page_Shape_2'
Let t = Select from Target where name='Help_Page_Link_1'
Create edge hasSource from $l to $s
Create edge hasTarget from $l to $t
```

In OSQL without explicit links:

```
Let s = Select from Source where name='Help_Page_Shape_2'
Let t = Select from Target where name='Help_Page_Link_1'
Create edge linksTo from $s to $t
```

It comes as no surprise that the more complex models take more time to create a Link between a given Source and Target than the more simple models. In both Neo4j and OrientDB this trend is clearly visible. Neo4j however outperforms OrientDB in all three graph models. The average time it took to create the Link is given in Figure 7.6.

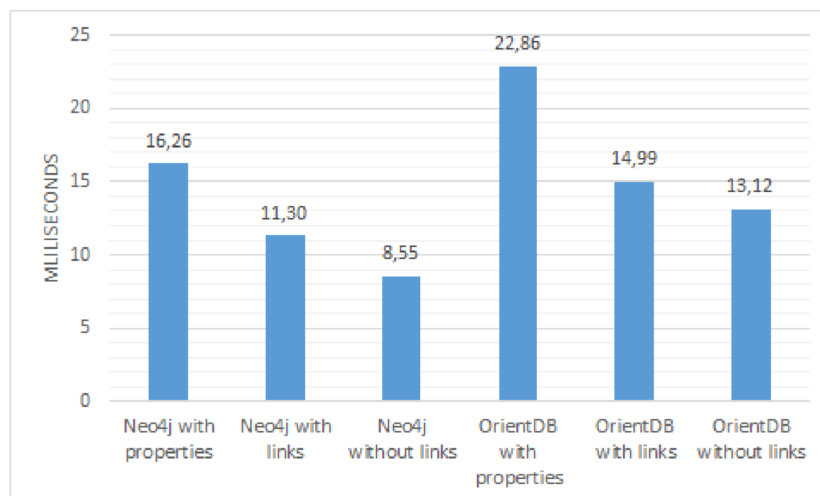

Figure 7.6: Result of query in section 7.2.2

7.2.3 Creating a New Instance

With this query, the goal is to create a whole new instance in the database, being a Source that links to a Target, with the Source having a layer, being on a page that refers to a document and a creator that creates all of this. The different possibilities to model this are given in Section 6.2. The queries to insert this in Cypher and OSQL are given below.

In Cypher with explicit properties:

```
create (l:Link)
create (n1:Name{name:"test"})
create (l)-[:hasName]->(n1)

create (c:Creator)
create (n2:Name{name:"test"})
create (p1:Prop{prop:"test"})
create (p2:Prop{prop:"test"})
create (p3:Prop{prop:"test"})
create (p4:Prop{prop:"test"})
create (c)-[:hasName]->(n2)
create (c)-[:hasProp]->(p1)
create (c)-[:hasProp]->(p2)
create (c)-[:hasProp]->(p3)
create (c)-[:hasProp]->(p4)

create (s:Source)
create (n3:Name{name:"test"})
create (p5:Prop{prop:"test"})
create (p6:Prop{prop:"test"})
create (p7:Prop{prop:"test"})
create (p8:Prop{prop:"test"})
create (s)-[:hasName]->(n3)
create (s)-[:hasProp]->(p5)
create (s)-[:hasProp]->(p6)
create (s)-[:hasProp]->(p7)
create (s)-[:hasProp]->(p8)

create (la:Layer)
create (n4:Name{name:"test"})
create (la)-[:hasName]->(n4)

create (p:Page)
create (n5:Name{name:"test"})
create (p9:Prop{prop:"test"})
create (p)-[:hasName]->(n5)
create (p)-[:hasProp]->(p9)

create (d:Document)
create (n6:Name{name:"test"})
create (p10:Prop{prop:"test"})
create (p11:Prop{prop:"test"})
create (p12:Prop{prop:"test"})
create (p13:Prop{prop:"test"})
create (d)-[:hasName]->(n6)
create (d)-[:hasProp]->(p10)
create (d)-[:hasProp]->(p11)
create (d)-[:hasProp]->(p12)
create (d)-[:hasProp]->(p13)
```

```
create (t:Target)
create (n7:Name{name:"test"})
create (p14:Prop{prop:"test"})
create (p15:Prop{prop:"test"})
create (p16:Prop{prop:"test"})
create (p17:Prop{prop:"test"})
create (t)-[:hasName]->(n7)
create (t)-[:hasProp]->(p14)
create (t)-[:hasProp]->(p15)
create (t)-[:hasProp]->(p16)
create (t)-[:hasProp]->(p17)

create (l)-[:hasTarget]->(t)
create (l)-[:hasSource]->(s)
create (c)-[:creates]->(l)
create (c)-[:creates]->(s)
create (c)-[:creates]->(p)
create (c)-[:creates]->(d)
create (c)-[:creates]->(t)
create (s)-[:has]->(la)
create (s)-[:refersTo]->(p)
create (p)-[:has]->(d)

return l,c,s,la,p,d,t,n1,n2,n3,n4,n5,n6,n7,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,
    p11,p12,p13,p14,p15,p16,p17
```

In Cypher with explicit links:

```
create (l:Link{name:"test"})
create (c:Creator{name:"test", prop2:"test", prop3:"test", prop4:"test"})
create (s:Source{name:"test",prop2:"test",prop3:"test",prop4:"test"})
create (la:Layer{name:"test"})
create (p:Page{name:"test",prop2:"test"})
create (d:Document{name:"test",prop2:"test",prop3:"test",prop4:"test",prop5:
    "test"})
create (t:Target{name:"test",prop2:"test",prop3:"test",prop4:"test",prop5:"
    test"})
create (l)-[:hasTarget]->(t)
create (l)-[:hasSource]->(s)
create (c)-[:creates]->(l)
create (c)-[:creates]->(s)
create (c)-[:creates]->(p)
create (c)-[:creates]->(d)
create (c)-[:creates]->(t)
create (s)-[:has]->(la)
create (s)-[:refersTo]->(p)
create (p)-[:has]->(d)
return l,c,s,la,p,d,t
```

In Cypher without explicit links:

```
create (c:Creator{name:"test", prop2:"test", prop3:"test", prop4:"test"})
create (s:Source{name:"test",prop2:"test",prop3:"test",prop4:"test"})
create (la:Layer{name:"test"})
create (p:Page{name:"test",prop2:"test"})
create (d:Document{name:"test",prop2:"test",prop3:"test",prop4:"test",prop5:
    "test"})
create (t:Target{name:"test",prop2:"test",prop3:"test",prop4:"test",prop5:"
    test"})
create (s)-[:linksTo{name:"test", prop2:"test", prop3:"test", prop4:"test",
    prop5:"test"}]->(t)
```

```
create (c)-[:creates]->(l)
create (c)-[:creates]->(s)
create (c)-[:creates]->(p)
create (c)-[:creates]->(d)
create (c)-[:creates]->(t)
create (s)-[:has]->(la)
create (s)-[:refersTo]->(p)
create (p)-[:has]->(d)
return c,s,la,p,d,t
```

In OSQL with explicit properties:

```
let l = create vertex Link set name='newLink'
let n1 = create vertex Name set name='newLink'
create edge hasName from $l to $n1;

let s = create vertex Source
let n2 = create vertex Name set name='newSource'
let p1 = create vertex prop1 set name='test'
let p2 = create vertex prop2 set name='test'
let p3 = create vertex prop3 set name='test'
let p4 = create vertex prop4 set name='test'
create edge hasName from $s to $n2
create edge hasProp from $s to $p1
create edge hasProp from $s to $p2
create edge hasProp from $s to $p3
create edge hasProp from $s to $p4


let t = create vertex Target
let n3 = create vertex Name set name='newTarget'
let p5 = create vertex prop1 set name='test'
let p6 = create vertex prop2 set name='test'
let p7 = create vertex prop3 set name='test'
let p8 = create vertex prop4 set name='test'
let p9 = create vertex prop5 set name='test'
create edge hasName from $t to $n3
create edge hasProp from $t to $p5
create edge hasProp from $t to $p6
create edge hasProp from $t to $p7
create edge hasProp from $t to $p8
create edge hasProp from $t to $p9

let c = create vertex Creator
let n4 = create vertex Name set name='newCreator'
let p10 = create vertex prop1 set name='test'
let p11 = create vertex prop2 set name='test'
let p12 = create vertex prop3 set name='test'
let p13 = create vertex prop4 set name='test'
let p14 = create vertex prop5 set name='test'
create edge hasName from $c to $n4
create edge hasProp from $c to $p10
create edge hasProp from $c to $p11
create edge hasProp from $c to $p12
create edge hasProp from $c to $p13
create edge hasProp from $c to $p14

let la = create vertex Layer
let n5 = create vertex Name set name='newLayer'
create edge hasName from $la to $n5

let p = create vertex Page set name='newPage', prop2='test'
```

```
let n6 = create vertex Name set name='newPage'
let p15 = create vertex prop2 set name='test'
create edge hasName from $p to $n6
create edge hasProp from $p to $p15

let d = create vertex Document
let n7 = create vertex Name set name='newDocument'
let p16 = create vertex prop1 set name='test'
let p17 = create vertex prop2 set name='test'
let p18 = create vertex prop3 set name='test'
let p19 = create vertex prop4 set name='test'
let p20 = create vertex prop5 set name='test'
create edge hasName from $d to $n7
create edge hasProp from $d to $p16
create edge hasProp from $d to $p17
create edge hasProp from $d to $p18
create edge hasProp from $d to $p19
create edge hasProp from $d to $p20

create edge creates from $c to $l
create edge creates from $c to $s
create edge creates from $c to $t
create edge creates from $c to $p
create edge creates from $c to $d
create edge hasTarget from $l to $t
create edge hasSource from $l to $s
create edge has from $s to $la
create edge refersTo from $s to $p
create edge has from $p to $d
```

In OSQL with explicit links:

```
let l = create vertex Link set name='newLink'
let s = create vertex Source set name='newSource', prop2='test', prop3='test
    ', prop4='test'
let t = create vertex Target set name='newTarget', prop2='test', prop3='test
    ', prop4='test',prop5='test'
let c = create vertex Creator set name='newCreator', prop2='test', prop3='
    test', prop4='test'
let la = create vertex Layer set name='newLayer'
let p = create vertex Page set name='newPage', prop2='test'
let d = create vertex Document set name='newCreator', prop2='test', prop3='
    test', prop4='test',prop5='test'
create edge creates from $c to $l
create edge creates from $c to $s
create edge creates from $c to $t
create edge creates from $c to $p
create edge creates from $c to $d
create edge hasTarget from $l to $t
create edge hasSource from $l to $s
create edge has from $s to $la
create edge refersTo from $s to $p
create edge has from $p to $d
```

In OSQL without explicit links:

```
let s = create vertex Source set name='newSource', prop2='test', prop3='test
    ', prop4='test'
let t = create vertex Target set name='newTarget', prop2='test', prop3='test
    ', prop4='test',prop5='test'
let c = create vertex Creator set name='newCreator', prop2='test', prop3='
    test', prop4='test'
```

```
let la = create vertex Layer set name='newLayer'
let p = create vertex Page set name='newPage', prop2='test'
let d = create vertex Document set name='newCreator', prop2='test', prop3='
    test', prop4='test',prop5='test'
create edge creates from $c to $s
create edge creates from $c to $t
create edge creates from $c to $p
create edge creates from $c to $d
create edge linksTo from $s to $t
create edge has from $s to $la
create edge refersTo from $s to $p
create edge has from $p to $d
```

The results of these queries are depicted in Figure 7.7 and show the average query time in milliseconds. The trend of slower queries in the databases ‘with explicit properties’ is again visible. The models ‘without explicit links’, where there are less nodes in the database, proof themselves again to be faster than the other models. While Neo4j and OrientDB are almost evenly fast for the models without explicit properties, it is remarkable that Neo4j is a lot faster when tested for the query with explicit properties.

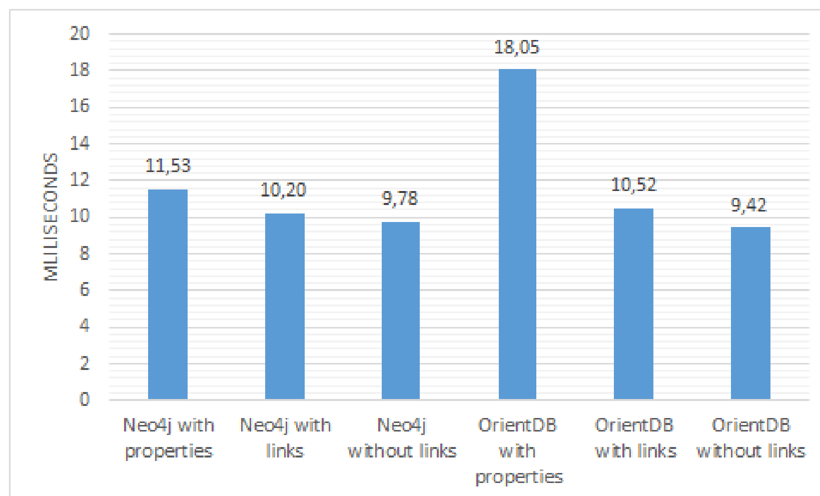

Figure 7.7: Result of query in section 7.2.3

7.2.4 Conclusion Write Performance

The first conclusion that can be drawn about the write performance, is the same as with the read performance: the more nodes and edges there are, the slower the queries are executed. However this is not surprising at all, one has to consider the speed of returning the queries regarding to the extra functionality more intermediate nodes and edges bring.

In contrast to the slower performance of Neo4j compared to OrientDB

regarding the read performance when lots of records have to be returned, there is no clear winner between both databases when it comes to the write performance.

The remarkable fact that OrientDB ‘with explicit properties’ performs way slower in contrast to all the other tested databases is also present for the write performance. In all three tests above, this database performs the worst by far. This strengthens the hypothesis that OrientDB is not the best fit for a lot of graph traversals.

7.3 Traversal Performance

7.3.1 Shortest Path

Graph databases claim to be very good at traversing graphs. This also includes algorithms as shortest paths. The queries that are formulated below return the shortest path between an entity with the name “Help Page Shape 1” and an entity with the name “Info Shape p-o1180”[1]. It should be noticed that the shortest path between these entities can only follow relationships via links (hasSource or hasTarget relationships). In this way it is checked how a Source links to a Target, which on its turn can be a Source to link to another Target and so on. If the two entities are for example created by the same Creator, the shortest path will be via this Creator. However, these latter shortest paths are not included. Only traversals via links are allowed in the queries below.

In Cypher with explicit properties:

```
MATCH (s:Source)-[:hasName]-(:Name{name:"Help Page Shape 1"})
MATCH (ss:Source)-[:hasName]-(:Name{name:"Performance Info Shape p-o1180"})
MATCH p=shortestPath((s)-[:hasSource|:hasTarget*]-(ss))
RETURN p;
```

In Cypher with explicit links:

```
MATCH p= shortestPath((s:Source{name:"Help Page Shape 1"})-[rels:hasTarget |
    :hasSource*]-(t:Target{name:"Performance Info Shape p-o1180"}))
RETURN p
```

In Cypher without explicit links:

```
Match p=shortestPath((s:Source{name:"Help Page Shape 1"})-[:linksTo*]-(t:
    Target{name:"Performance Info Shape p-o1180"}))
Return p
```

[1]Different paths were artificially added to the data to check whether the queries actually returns the shortest path between these entities. The shortest path traverses 7 nodes if links are modelled as nodes, otherwise the shortest path contains 4 nodes. To know all the paths between two nodes, see Section 7.3.2

In OSQL with explicit properties: The 'links' class in the queries below is a superclass for all the edges that are allowed to be traversed, namely hasSource and hasTarget.

```
select expand(shortestpath ((select expand(both('hasName')) from Name where
    name="Help_Page_Shape_1"),(select expand(both('hasName')) from Name
    where name="Performance_Info_Shape_p-o1180"),null,'links'));
```

In OSQL with explicit links:

```
select expand(shortestpath ((select from Source where name='Help_Page_Shape_
    1'),(select from Source where name='Performance_Info_Shape_p-o1180'),
    null,'links'));
```

In OSQL without explicit links:

```
select expand(shortestpath ((select from Source where name='Help_Page_Shape_
    1'),(select from Source where name='Performance_Info_Shape_p-o1180'),
    null,'linksTo'));
```

As expected, the shortest path query returns the shortest path faster if there are less relationships to traverse. The more complex model 'with explicit properties' is slower than the model 'with explicit links', which on its turn is slower than the model 'without explicit links'. It is more remarkable that Neo4j performs these queries way faster than OrientDB.

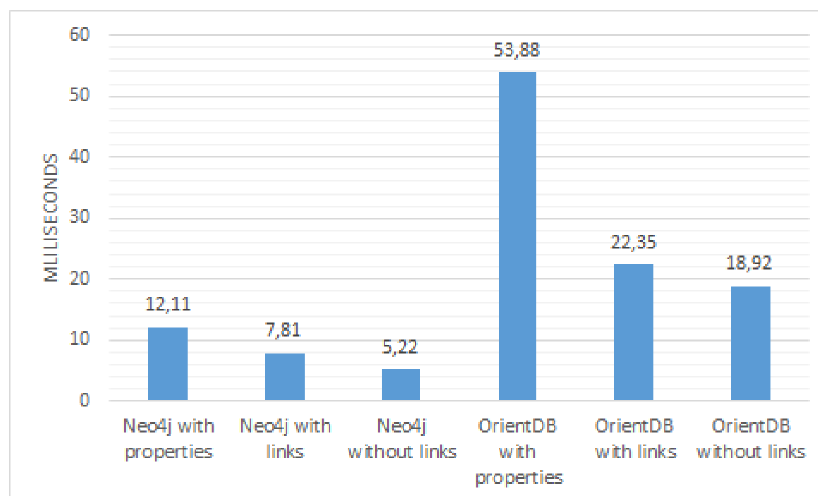

Figure 7.8: Result of query in section 7.3.1

7.3.2 All Paths

Instead of only returning the shortest path, this query serves to return all the paths between the given entities[2]. Again, only the paths via links are allowed. All the queries correctly return 12 different paths. The Cypher queries to do so are shown below. Unfortunately, it is not possible to perform this query in OrientDB in an elegant way in this version.

In Cypher with explicit properties:

```
MATCH (s:Source)-[:hasName]-(:Name{name:"Help_Page_Shape_1"})
MATCH (ss:Source)-[:hasName]-(:Name{name:"Performance_Info_Shape_p-o1180"})
MATCH p=(s)-[:hasSource|:hasTarget*]-(ss)
RETURN p;
```

In Cypher with explicit links:

```
Match p=(s{name:"Help_Page_Shape_1"})-[rels:hasTarget | :hasSource*]-(t{name
    :"Performance_Info_Shape_p-o1180"})
Return p;
```

In Cypher without explicit links:

```
Match p=(s:Source{name:"Help_Page_Shape_1"})-[:linksTo*]-(t:Target{name:"
    Performance_Info_Shape_p-o1180"})
Return p;
```

The execution speed is slower than the execution of the shortest path query. It is again clear in Figure {7.9 that the more complex the graph model is, the longer it takes to return all the paths. The differences are getting bigger when the queries need more graph traversals. As mentioned above already, it is not possible to query the database for all the paths between two given entities in an elegant way in OrientDB. However, it is remarkable that Neo4j is even faster in looking for all different paths (Figure7.9) than OrientDB is for only returning the shortest path (Figure 7.8).

[2]These paths were artificially created in the data to check whether the query returns the correct paths

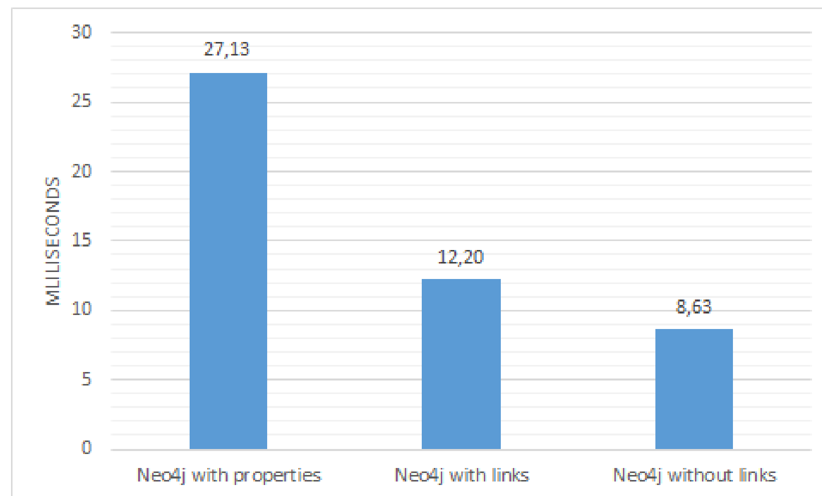

Figure 7.9: Result of query in section 7.3.2

7.3.3 Conclusion Traversal Performance

The statement that OrientDB performs worse than Neo4j when a traversal of the graph is needed, becomes even more clear in this section. To find the shortest path, a lot of traversals are of course necessary. As can be seen in Figure 7.8, Neo4j scores a lot better than OrientDB. Neo4j is on average about 3 times faster to execute the query than OrientDB is. An even more traversal intense job is to return all the possible paths between two nodes. Neo4j handles this quite fine (Figure 7.9). OrientDB does not support a function or another elegant way to get all these paths. Not only the performance is worse in OrientDB, also the possibilities and functionalities for traversing graphs is less supported.

Furthermore, the models with more nodes and edges are, as one can expect, less performant than the simpler models for queries that require intensive graph traversing.

7.4 Conclusion of the Benchmark

A first conclusion that can be drawn out of all these tests is the fact that the graph models that are more complex, with more nodes and edges, are performing slower than databases without these extra nodes. These extra nodes are the result of decisions to model properties and links as nodes or not. In the most extended model, the model 'with explicit properties', both properties and links are modelled as separate nodes. This slows down the

query performance in comparison to the model 'with explicit links', where properties are modelled as attributes of entity nodes, while links are still modelled as nodes. On its turn, this model returns results slower when queried in comparison to the model 'without explicit links', where the links are modelled as edges between two entity nodes. Simplifying the databases to boost query speed has also a downside: the functionality of the model also drops. When links are modelled as edges for examples, it is no longer possible to have links to other links, since a relationship cannot point to an edge. Also the information that can be stored in an edge is less than in a node.

Other interesting results are due to the different nature of Neo4j and OrientDB. While Neo4j is a native graph database, OrientDB is a multi-model database, which combines the graph model with a document datastore. In the results, this becomes clear in two ways. Firstly, when a lot of results should be returned for a query, Neo4j is a lot slower to do so than OrientDB. This can be clearly seen in Figure 7.1 and Figure 7.4, where it takes OrientDB only a fraction of the time Neo4j needs to return a large number of nodes. As already said, this is due to the nature of both databases. The native graph database of Neo4j focusses on graph querying and less on returning batches of results. The focus on graph querying also has some positive results of course. This becomes clear when a lot of graph traversals has to be done for a query, which is the second case where the different nature between both databases becomes clear. In all the relevant queries that are performed in Section 7.1 and Section 7.2, the OrientDB database with the model 'with explicit properties' performs way slower than all the other database instances. This raises the conjecture that OrientDB does not handle graph traversals very well. This conjecture is confirmed in Section 7.3, where more intensive graph traversal queries were ran against the different databases. The time differences between Neo4j and OrientDB to calculate the shortest path between two nodes are huge (Figure7.8). Not only the query speed for graph traversals is better in Neo4j than in OrientDB, also the functionality around this is better. In OrientDB, it is for example not possible to retrieve all the paths between two nodes. Neo4j however, returns all the different paths very fast thanks to a built-in function. This is again due to the different nature of both databases: while Neo4j is a native graph database and thus only focusses on graphs, OrientDB is a multi-model database that tries to combine different models such as a graph model and a document model.

8 Conclusion and Future Work

8.1 Conclusion

When data is strongly connected, graph databases are a good back end storage solution, according to literature. Before evaluating these graph databases, a clear understanding of the different features of graph databases is necessary. An introductory explanation about graph databases, along with some use cases and the role of graph databases in the NoSQL movement, is given in Chapter 3. Afterwards, in Chapter 4, different commercially operational graph databases are introduced. Out of this list, two database vendors are chosen based on benchmarks in the literature and based on their promising query languages. The databases that were chosen are Neo4j and OrientDB. Neo4j is a native graph database, while OrientDB is a multi-model database, combining graph features with a document store. Both Neo4j, with Cypher, and OrientDB, with OSQL, aim to develop the standard query language for graph databases. For now, it is not yet sure who will win this. However, Neo4j is now the leading graph database(DB-Engines, 2016) and it plans to stay there.

Before being able to compare Neo4j and OrientDB with each other, a data model has to be designed that fits graph databases. It is possible to do so using different approaches. As discussed in Chapter 5, the solution to work with labels in Neo4j and classes in OrientDB is the most elegant way to

deal with typing and sub-typing. Also the solution to store BLOBs in graph databases is quite straightforward: since graph databases are by nature not meant to be partitioned, it won't be a good idea to store all kinds of large blob files directly into the database. Therefore, keeping a reference to the according file and storing the file itself in another kind of database (key-value for example), seems to be the best solution.

Regarding the modelling of links and properties, the choices are less obvious. Links can be modelled as edges between two nodes or as an explicit intermediary node. The latter approach has the advantage that links can link to other links, as proposed in the RSL metamodel. Properties can also be modelled according to the RSL metamodel, thus as separate nodes. Another approach is to model properties as attributes of entity nodes. Both solutions should be further investigated, since the loss of functionality of one approach should be weighed against a possible increase of querying speed.

Regarding the considerations about different modelling options in Chapter 5, three data models are proposed in Chapter 6. The first one, 'with explicit properties', is a model closely to the RSL metamodel. Both properties and links are modelled as separate nodes. In a first attempt to simplify the model, the properties were no longer modelled as nodes, but as attributes of entity nodes. This model is called the 'with explicit links' model, since links are still modelled as nodes. In a last model, 'without explicit links', also the links are no longer modelled as nodes, but are now designed as edges between entity nodes.

The data of the EdFest application is imported according to the three different models in two different databases, Neo4j and OrientDB. This makes 6 different databases that are compared against each other in Chapter 7. If the three proposed graph models are compared to each other, it holds for both their implementation in Neo4j and OrientDB that the model 'with explicit properties' is by far the slowest for every kind of query. The model 'with explicit links' is also slower than the model 'without explicit links'. However, it should be considered for every application whether the gain in speed is preferable over the loss of functionality. After all, some generality from the RSL metamodel is lost by simplifying the data model to improve the query performance.

When comparing Neo4j to OrientDB, two conclusions can be drawn. Firstly, Neo4j outperforms OrientDB when it comes to traversing graphs. When there are more nodes to be traversed, as in the model 'with explicit properties', OrientDB always lags behind regarding the query execution. Also in the queries to find the shortest path between two nodes for example, Neo4j performs way faster than OrientDB does. This is due to the nature of

both databases: Neo4j is a native graph database, built from deep down to handle graphs, while OrientDB is a multi-model database, which combines a document store with graph functionality. This different kind of nature also results in a second conclusion: OrientDB outperforms Neo4j when the database has to return a lot of results. When the database was queried to, for example, return all the nodes of a specific type, Neo4j took way longer to perform this than OrientDB does.

However, it is not easy to say which database and which data model is the best. It really depends on the grounds you base the verdict, such as query speed, functionality, query language and so on. The database that is the best fit for an application, depends on what the applications aims to do. Based on this, the right kind of database, the right data model and the right vendor should be chosen. This thesis broadens the scope of databases that are taken into consideration to graph databases. Furthermore, it presents some possible datamodels. Finally, it compares the different datamodels in two different graph databases from a different vendor, namely OrientDB and Neo4j. Based on this information, an application developer should be able to decide whether a graph database could be used for the application or not.

8.2 Limitations

This thesis only discusses the option of graph databases for the back end storage for link services. It is not proven that graph databases are also the best fit for these kind of applications. Other kind of databases, such as object databases, key-value stores, document databases and relational databases should be considered and should be compared to each other on different aspects.

In this thesis, tests between two databases, Neo4j and OrientDB, are discussed. These database were chosen on general criteria such as the support of the community and the popularity of the query language. It is not proven that the query performance of these databases is therefore better than the query performance of other, lesser known databases. It is not proven that Neo4j or OrientDB is the best graph database around.

The graph databases that were further investigated in this thesis are Neo4j community edition 2.3.3 and OrientDB community edition 2.2.0, because these were the two most recent free versions of both databases at the time of writing this thesis. However, there are updates and new versions of these databases regularly, with new functionality and new features. Results may thus vary for other versions in the future.

8.3 Future Work

As said in the section about the limitations of this thesis, only two graph databases are considered in depth in this thesis. The results of these databases should be compared to results of other databases. These other databases can both be other graph databases, as well as other kind of databases, such as the object database in which the original data about EdFest was stored.

Furthermore, more queries can be compared to each other if necessary. Therefore, the application developer should have a clear view on the desired functionality of the application, so the exact queries for the database can be tested.

Appendix A

Cypher code to import data in graph model 'with explicit properties'

```
//CREATE links , sources and creators of links and sources

CREATE CONSTRAINT ON (l:Link) ASSERT l.name IS UNIQUE;
CREATE CONSTRAINT ON (c:Creator) ASSERT c.name IS UNIQUE;
CREATE CONSTRAINT ON (s:Source) ASSERT s.name IS UNIQUE;
CREATE CONSTRAINT ON (la: Layer) ASSERT la.name IS UNIQUE;
CREATE CONSTRAINT ON (p: Page) ASSERT p.name IS UNIQUE;
CREATE CONSTRAINT ON (d: Document) ASSERT d.name IS UNIQUE;
CREATE CONSTRAINT ON (t: Target) ASSERT t.id IS UNIQUE;
CREATE CONSTRAINT ON (name: Name) ASSERT name.name IS UNIQUE;
CREATE CONSTRAINT ON (des: Description) ASSERT des.description IS UNIQUE;
CREATE CONSTRAINT ON (login: Login) ASSERT login.login IS UNIQUE;
CREATE CONSTRAINT ON (pswd: Password) ASSERT pswd.password IS UNIQUE;

________________________________________________________________________


USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldata.csv' AS line

//link
MERGE (link:Link{name:line.name})
MERGE (name:Name{name:line.name})
MERGE (link)-[:hasName]-(name)

//creators
MERGE (linkcreator: Creator{name:line.name2})
MERGE (name2:Name{name:line.name2})
MERGE (description: Description{description:line.description})
MERGE (login:Login{login:line.login})
MERGE (password:Password{password:line.password})
```

```
MERGE (linkcreator)-[:hasName]->(name2)
MERGE (linkcreator)-[:hasDescription]->(description)
MERGE (linkcreator)-[:hasLogin]->(login)
MERGE (linkcreator)-[:hasPassword]->(password)


MERGE (sourcecreator: Creator{name:line.name4})
MERGE (name4:Name{name:line.name4})
MERGE (description5:Description{description:line.description5})
MERGE (login6:Login{login:line.login6})
MERGE (password7:Password{password:line.password7})
MERGE (sourcereator)-[:hasName]->(name4)
MERGE (sourcecreator)-[:hasDescription]->(description5)
MERGE (sourcecreator)-[:hasLogin]->(login6)
MERGE (sourcecreator)-[:hasPassword]->(password7)


MERGE (pagecreator: Creator{name:line.name10})
MERGE (name10:Name{name:line.name10})
MERGE (description11:Description{description:line.description11})
MERGE (login12:Login{login:line.login12})
MERGE (password13:Password{password:line.password13})
MERGE (pagecreator)-[:hasDescription]->(description11)
MERGE (pagecreator)-[:hasLogin]->(login12)
MERGE (pagecreator)-[:hasPassword]->(password13)

MERGE (documentcreator: Creator{name:line.name15})
MERGE (name15:Name{name:line.name15})
MERGE (description16:Description{description:line.description16})
MERGE (login17:Login{login:line.login17})
MERGE (password18:Password{password:line.password18})
MERGE (documentcreator)-[:hasDescription]->(description16)
MERGE (documentcreator)-[:hasLogin]->(login17)
MERGE (documentcreator)-[:hasPassword]->(password18)


//source
MERGE (source:Source{name:line.name3})
MERGE (name3:Name{name:line.name3})
MERGE (upperleftx:UpperLeftX{upperLeftX:line.x})
MERGE (upperlefty:UpperLeftY{upperLeftY:line.y})
MERGE (width:Width{width:line.width})
MERGE (height:Height{height:line.height})
MERGE (source)-[:hasName]->(name3)
MERGE (source)-[:hasUpperLeftX]->(upperleftx)
MERGE (source)-[:hasUpperLeftY]->(upperlefty)
MERGE (source)-[:hasWidth]->(width)
MERGE (source)-[:hasHeight]->(height)



//layer
MERGE (layer: Layer{name:line.name8})
MERGE (name8:Name{name:line.name8})
MERGE (layer)-[:hasName]->(name8)

//page
MERGE (page: Page{name:line.name9})
MERGE (name9:Name{name:line.name9})
MERGE (number:Number{number:line.number})
MERGE (page)-[:hasName]->(name9)
MERGE (page)-[:hasNumber]->(number)
```

```
//document
MERGE (document: Document{name: line.name14})
MERGE (name14:Name{name: line.name14})
MERGE (id2:Id{id:line.id2})
MERGE (width19:Width{width:line.width19})
MERGE (height20:Height{height:line.height20})
MERGE (content:Content{content:line.content})
MERGE (document)-[:hasName]->(name14)
MERGE (document)-[:hasId]->(id2)
MERGE (document)-[:hasWidth]->(width19)
MERGE (document)-[:hasHeight]->(height20)
MERGE (document)-[:hasContent]->(content)

//relationships

//MERGEs
MERGE (linkcreator)-[:creates]->(link)
MERGE (sourcecreator)-[:creates]->(source)
MERGE (pagecreator)-[:creates]->(page)
MERGE (documentcreator)-[:creates]->(document)


//link has source
MERGE (link)-[:hasSource]->(source)

//source has layer
MERGE (source)-[:has]->(layer)

//source refers to page
MERGE (source)-[:refersTo]->(page)

//page has a document
MERGE (page)-[:has]->(document)


OMSWE Targets
------------------------------------------------------------------------
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldataomswetargets.csv' AS line

//targets
MATCH (link:Link)
where link.name=line.name

MERGE (omswetargetcreator: Creator{name:line.name24})
MERGE (name24:Name{name:line.name24})
MERGE (description25:Description{description:line.description25})
MERGE (login26:Login{login:line.login26})
MERGE (password27:Password{password:line.password27})
MERGE (omswetargetcreator)-[:hasName]->(name24)
MERGE (omswetargetcreator)-[:hasDescription]->(description25)
MERGE (omswetargetcreator)-[:hasLogin]->(login26)
MERGE (omswetargetcreator)-[:hasPassword]->(password27)

CREATE (omswetarget:OmsweTarget :Target{name:line.name23})
MERGE (name23:Name{name:line.name23})
MERGE (description28:Description{description:line.description28})
MERGE (content29:Content{content:line.content29})
MERGE (id3:Id{line:line.id})
MERGE (omswetarget)-[:hasName]->(name23)
```

```
MERGE (omswetarget)-[:hasDescription]->(description28)
MERGE (omswetarget)-[:hasContent]->(content29)
MERGE (omswetarget)-[:hasId]->(id3)

MERGE (link)-[:hasTarget]->(omswetarget)
MERGE (omswetargetcreator)-[:creates]->(omswetarget)

 Active Component Targets
--------------------------------------------------------------------------
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
 'file:///Users/verey_000/Desktop/Data/Festivaldataactivecomponenttargets.csv
     ' AS line

MATCH (link:Link)
 where link.name=line.name

MERGE (actargetcreator: Creator{name:line.name31})
MERGE (name31:Name{name:line.name31})
MERGE (description32:Description{description:line.description32})
MERGE (login33:Login{login:line.login33})
MERGE (password34:Password{password:line.password34})
MERGE (actargetcreator)-[:hasName]->(name31)
MERGE (actargetcreator)-[:hasDescription]->(description32)
MERGE (actargetcreator)-[:hasLogin]->(login33)
MERGE (actargetcreator)-[:hasPassword]->(password34)

CREATE (actarget:AcTarget :Target{name:line.name30})
MERGE (name30:Name{name:line.name30})
MERGE (identifier:Identifier{identifier:line.identifier})
MERGE (uri:Uri{uri:line.uri})
MERGE (timeout:Timeout{timeout:line.timeout})
MERGE (data:Data{data:line.data})
MERGE (id4:Id{id:line.id})
MERGE (actarget)-[:hasName]->(name30)
MERGE (actarget)-[:hasIdentifier]->(identifier)
MERGE (actarget)-[:hasUri]->(uri)
MERGE (actarget)-[:hasTimeout]->(timeout)
MERGE (actarget)-[:hasData]->(data)
MERGE (actarget)-[:hasId]->(id4)


MERGE (link)-[:hasTarget]->(actarget)
MERGE (actargetcreator)-[:creates]->(actarget)

 MapButton Targets
--------------------------------------------------------------------------
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
 'file:///Users/verey_000/Desktop/Data/Festivaldatamapbuttontargets.csv' AS
     line

MATCH (link:Link)
 where link.name=line.name

MERGE (mptargetcreator: Creator{name:line.name36})
MERGE (name36:Name{name:line.name36})
MERGE (description37:Description{description:line.description37})
MERGE (login38:Login{login:line.login38})
MERGE (password39:Password{password:line.password39})
MERGE (mptargetcreator)-[:hasName]->(name36)
MERGE (mptargetcreator)-[:hasDescription]->(description37)
```

```
MERGE (mptargetcreator)-[:hasLogin]->(login38)
MERGE (mptargetcreator)-[:hasPassword]->(password39)

CREATE (mptarget:MpTarget :Target {name:line.name35})
MERGE (name35:Name{name:line.name35})
MERGE (id5:Id{id:line.id})
MERGE (identifier40:Identifier{identifier:line.identifier40})
MERGE (uri41:Uri{uri:line.uri41})
MERGE (timeout42:Timeout{timeout:line.timeout42})
MERGE (data43:Data{data:line.data43})
MERGE (captureShape:CaptureShape{captureShape:line.captureShape})
MERGE (fixUri:FixUri{fixUri:line.fixUri})
MERGE (mptarget)-[:hasName]->(name35)
MERGE (mptarget)-[:hasId]->(id5)
MERGE (mptarget)-[:hasIdentifier]->(identifier40)
MERGE (mptarget)-[:hasUri]->(uri41)
MERGE (mptarget)-[:hasTimeout]->(timeout42)
MERGE (mptarget)-[:hasData]->(data43)
MERGE (mptarget)-[:hasCaptureShape]->(captureShape)
MERGE (mptarget)-[:hasFixUri]->(fixUri)


MERGE (link)-[:hasTarget]->(mptarget)
MERGE (mptargetcreator)-[:creates]->(mptarget)
```

Cypher code to import data in graph model 'with explicit links'

```
//create links, sources and creators of links and sources

CREATE CONSTRAINT ON (l:Link) ASSERT l.name IS UNIQUE;
CREATE CONSTRAINT ON (c:Creator) ASSERT c.name IS UNIQUE;
CREATE CONSTRAINT ON (s:Source) ASSERT s.name IS UNIQUE;
CREATE CONSTRAINT ON (la: Layer) ASSERT la.name IS UNIQUE;
CREATE CONSTRAINT ON (p: Page) ASSERT p.name IS UNIQUE;
CREATE CONSTRAINT ON (d: Document) ASSERT d.name IS UNIQUE;
CREATE CONSTRAINT ON (t: Target) ASSERT t.id IS UNIQUE;

____________________________________________________________


USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldata.csv' AS line

//link
CREATE (link:Link {name: line.name})

//creators
MERGE (linkcreator: Creator {name: line.name2, description: line.description
    , login: line.login, password: line.password})
MERGE (sourcecreator: Creator {name: line.name4, description: line.
    description5, login: line.login6, password: line.password7})
MERGE (pagecreator: Creator {name: line.name10, description: line.
    description11, login: line.login12, password: line.password13})
MERGE (documentcreator: Creator {name: line.name15, description: line.
    description16, login: line.login17, password: line.password18})

//source
MERGE (source:Source {name: line.name3, upperleftx: line.x, upperlefty:line.
    y, width: line.width, height: line.height})

//layer
MERGE (layer: Layer {name: line.name8})
```

```
//page
MERGE (page: Page {name: line.name9, number: line.number})

//document
MERGE (document: Document {name: line.name14, id: line.id2, width: line.
    width19, heigth: line.height20, content: line.content})

//relationships

//creates
CREATE (linkcreator)-[:creates]->(link)
CREATE (sourcecreator)-[:creates]->(source)
CREATE (pagecreator)-[:creates]->(page)
CREATE (documentcreator)-[:creates]->(document)


//link has source
CREATE (link)-[:hasSource]->(source)

//source has layer
CREATE (source)-[:has]->(layer)

//source refers to page
CREATE (source)-[:refersTo]->(page)

//page has a document
CREATE (page)-[:has]->(document)


OMSWE Targets
———————————————————————————————————————
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldataomswetargets.csv' AS line

//targets
MATCH (link:Link)
where link.name=line.name

MERGE (omswetargetcreator: Creator {name: line.name24, description: line.
    description25, login: line.login26, password: line.password27})
MERGE (omswetarget:OmsweTarget :Target {name: line.name23, description: line
    .description28, content: line.content29, id:line.id})

CREATE (link)-[:hasTarget]->(omswetarget)
CREATE (omswetargetcreator)-[:creates]->(omswetarget)

Active Component Targets
———————————————————————————————————————
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldataactivecomponenttargets.csv
    ' AS line

MATCH (link:Link)
where link.name=line.name

MERGE (actargetcreator: Creator {name: line.name31, description: line.
    description32, login: line.login33, password: line.password34})
MERGE (actarget:AcTarget :Target {name: line.name30, identifier: line.
    identifier, uri: line.uri, timeout: line.timeout, data: line.data, id:
```

```
    line.id})

CREATE (link)-[:targets]->(actarget)
CREATE (actargetcreator)-[:creates]->(actarget)

MapButton Targets
_______________________________________________________________

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldatamapbuttontargets.csv' AS
    line

MATCH (link:Link)
where link.name=line.name

MERGE (mptargetcreator: Creator {name: line.name36, description: line.
    description37, login: line.login38, password: line.password39})
MERGE (mptarget:MpTarget :Target {id: line.id, name: line.name35, identifier
    : line.identifier40, uri: line.uri41, timeout: line.timeout42, data:
    line.data43, captureShape: line.captureShape, fixUri: line.fixUri})

CREATE (link)-[:targets]->(mptarget)
CREATE (mptargetcreator)-[:creates]->(mptarget)
```

Cypher code to import data in graph model 'without explicit links'

```
//create links, sources and creators of links and sources

CREATE CONSTRAINT ON (c:Creator) ASSERT c.name IS UNIQUE;
CREATE CONSTRAINT ON (s:Source) ASSERT s.name IS UNIQUE;
CREATE CONSTRAINT ON (la: Layer) ASSERT la.name IS UNIQUE;
CREATE CONSTRAINT ON (p: Page) ASSERT p.name IS UNIQUE;
CREATE CONSTRAINT ON (d: Document) ASSERT d.name IS UNIQUE;
CREATE CONSTRAINT ON (t: Target) ASSERT t.id IS UNIQUE;

______________________________________________________________________


USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldata.csv' AS line

//creators
MERGE (sourcecreator: Creator {name: line.name4, description: line.
    description5, login: line.login6, password: line.password7})
MERGE (pagecreator: Creator {name: line.name10, description: line.
    description11, login: line.login12, password: line.password13})
MERGE (documentcreator: Creator {name: line.name15, description: line.
    description16, login: line.login17, password: line.password18})

//source
MERGE (source:Source {name: line.name3, upperleftx: line.x, upperlefty:line.
    y, width: line.width, height: line.height})

//layer
MERGE (layer: Layer {name: line.name8})

//page
MERGE (page: Page {name: line.name9, number: line.number})

//document
MERGE (document: Document {name: line.name14, id: line.id2, width: line.
    width19, heigth: line.height20, content: line.content})
```

```
//relationships

//creates
CREATE (sourcecreator)-[:creates]->(source)
CREATE (pagecreator)-[:creates]->(page)
CREATE (documentcreator)-[:creates]->(document)


//source has layer
CREATE (source)-[:has]->(layer)

//source refers to page
CREATE (source)-[:refersTo]->(page)

//page has a document
CREATE (page)-[:has]->(document)


OMSWE Targets
----------------------------------------------------------------
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldataomswetargets.csv' AS line

//targets
MATCH (source:Source)
where source.name=line.name3

MERGE (omswetargetcreator: Creator {name: line.name24, description: line.
    description25, login: line.login26, password: line.password27})
MERGE (omswetarget:OmsweTarget :Target {name: line.name23, description: line
    .description28, content: line.content29, id:line.id})

CREATE (source)-[:linksTo{linkName:line.name,creatorName:line.name2,
    creatorDescription:line.description,creatorLogin:line.login,
    creatorPassword:line.password}]->(omswetarget)
CREATE (omswetargetcreator)-[:creates]->(omswetarget)

Active Component Targets
----------------------------------------------------------------
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldataactivecomponenttargets.csv
    ' AS line

MATCH (source:Source)
where source.name=line.name3

MERGE (actargetcreator: Creator {name: line.name31, description: line.
    description32, login: line.login33, password: line.password34})
MERGE (actarget:AcTarget :Target {name: line.name30, identifier: line.
    identifier, uri: line.uri, timeout: line.timeout, data: line.data, id:
    line.id})

CREATE (source)-[:linksTo{linkName:line.name,creatorName:line.name2,
    creatorDescription:line.description,creatorLogin:line.login,
    creatorPassword:line.password}]->(actarget)
CREATE (actargetcreator)-[:creates]->(actarget)

MapButton Targets
----------------------------------------------------------------
```

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
'file:///Users/verey_000/Desktop/Data/Festivaldatamapbuttontargets.csv' AS
    line

MATCH (source:Source)
where source.name=line.name3

MERGE (mptargetcreator: Creator {name: line.name36, description: line.
    description37, login: line.login38, password: line.password39})
MERGE (mptarget:MpTarget :Target {id: line.id, name: line.name35, identifier
    : line.identifier40, uri: line.uri41, timeout: line.timeout42, data:
    line.data43, captureShape: line.captureShape, fixUri: line.fixUri})

CREATE (source)-[:linksTo{linkName:line.name,creatorName:line.name2,
    creatorDescription:line.description,creatorLogin:line.login,
    creatorPassword:line.password}]->(mptarget)
CREATE (mptargetcreator)-[:creates]->(mptarget)
```

References

ArangoDB. (2015). *ArangoDB Documentation.* Retrieved from www.arangodb.com

Brewer, E. A. (2000). Towards Robust Distributed Systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing.* ACM.

DB-Engines. (2016). *DB-Engines Ranking.* Retrieved from http://www.db-engines.com/en/

Dominguez-Sal, D., Urbon-Bayes, P., Gimenez-Vano, A., Gomez-Villamor, S., Martinez-Bazan, N., & Larriba-Pe, J. L. (2010). Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. In *WAIM'10 Proceedings of the 2010 international conference on Web-age information management* (p. 37-48). Springer.

Edinburgh Festival Fringe Society. (2016). *The Edinburgh festival fringe: defying the norm since 1947.* Retrieved from https://www.edfringe.com/

Eifrem, E. (2015). *Meet openCypher: The SQL for Graphs.* Retrieved from http://neo4j.com/blog/open-cypher-sql-for-graphs/

Feinberg, D., Adrian, M., Heudecker, N., Ronthal, A. M., & Palanca, T. (2015). *Magic Quadrant for Operational Database Management Systems.* Gartner. Retrieved from https://www.gartner.com/doc/reprints?id=1-2PMFPEN&ct=151013

Fowler, M. (2012). *AggregateOrientedDatabase.* Retrieved from http://martinfowler.com/bliki/AggregateOrientedDatabase.html

Fowler, M., & Sadalage, P. J. (2013). *NoSQL Distilled: A brief guide to the emerging world of polyglot persistence.* Upper Saddle River, NJ: Addison-Wesley Pearson Education.

Garulli, L. (2015). *New benchmarks from Arango.* Retrieved from `http://orient-database.narkive.com/VJ69TfiV/orientdb-new-benchmarks-from-arango`

Gilbert, S., & Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partitioning Tolerant Web Services. *ACM SIGACT*, *33*(2), 51-59.

GraphDatabase. (2016). *from Wikipedia.* Retrieved from `https://en.wikipedia.org/wiki/Graph_database`

Grzegorczyk, K. (2014). *Graph Databases* [PowerPoint]. Retrieved from `http://www.slideshare.net/kagrze/graph-databases-35707673`

Guzenda, L., & Quinn, N. (2013). *An introduction to graph databases* [PowerPoint]. Objectivity Inc. Retrieved from `http://www.slideshare.net/infinitegraph/an-introduction-to-graph-databases`

Halasz, F., & Schwarz, M. (1994). Dexter Hypertext Reference Model. *Communications of the ACM*, *37*(2), 30-39.

Hardman, L., Bulterman, D. C. A., & van Rossum, G. (1994). The Amsterdam hypermedia model: adding time and context to the Dexter model. *Communications of the ACM*, *37*(2), 50-62.

Jouili, S., & Vansteenberghe, V. (2013). An Empirical Comparison of Graph Databases. In *Social Computing (SocialCom), 2013 International Conference on* (p. 708 - 715). IEEE.

Liu, H. C., & Chirathamjaree, C. (1996). An Efficient Join for Nested Relational Databases. In *Database Expert System Applications, 7th International Conference* (p. 289-301). Springer.

MarkLogic. (2015). *MarkLogic Documentation.* Retrieved from `www.marklogic.com`

McCreary, D., & Kelly, A. (2014). *Making sense of NoSQL: A guide for managers and the rest of us.* Shelter Island, NY: Manning Publications Co.

Merenyi, R. (2013). *What are the differences between relational and graph databases.* Retrieved from `http://www.seguetech.com/blog/2013/02/04/what-are-the-differences-between-relational-and-graph-databases`

Nelson, T. H. (1965). Complex Information Processing: a File Structure for the Complex, the Changing and the Indeterminate. In *ACM '65 Proceedings of the 1965 20th national conference* (p. 84-100). ACM.

Neo4j. (2015). *Neo4j Documentation.* Retrieved from `www.neo4j.com`

Norrie, M. C. (1993). An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *12th International Conference on the Entity-Relationship Approach, Arlington, USA.* Springer.

Objectivity. (2015). *InfiniteGraph Documentation.* Retrieved from `www.objectivity.com/products/infinitegraph/`

OrientDB. (2013). *XDGBench 3rd party benchmark results against graph databases.* Retrieved from `http://orientdb.com/xdgbench-3rd-party-benchmark-results-against-graph-databases/`

OrientDB. (2015). *OrientDB Documentation.* Retrieved from `www.orientdb.com`

OrientTechnologies. (2016). *SQL Tutorial.* Retrieved from `http://orientdb.com/docs/2.0/orientdb.wiki/Tutorial-SQL.html`

Partner, J., & Vukotic, A. (2012). *Neo4j in action: early access edition.* Shelter Island, NY: Manning Publications.

Rahien, A. (2010). *That NoSQL thing: Column (Family) Databases.* Retrieved from `https://ayende.com/blog/4500/that-no-sql-thing-column-family-databases`

Robinson, I., Webber, J., & Eifrem, E. (2010). *Graph Databases.* Sebastopol, CA: O'Reilly Media.

Schwabe, D., & Rossi, G. (1995). The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, *38*(8), 45-46.

Signer, B. (2005). *Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces* (Doctoral dissertation).

Signer, B., & Norrie, M. C. (2007, November). As We May Link: A General Metamodel for Hypermedia Systems. In *Proceedings of the 26th International Conference on Conceptual Modeling.* Springer.

Sparksee. (2016). *Sparksee Documentation.* Retrieved from `http://www.sparsity-technologies.com/`

Stardog. (2015). *Stardog Documentation.* Retrieved from `www.stardog.com`

Think Aurelius. (2015). *Titan Documentation.* Retrieved from `thinkaurelius.github.io/titan/`

Vasiliev, A. (2013). *World of the NoSQL Databases.* Retrieved from `http://leopard.in.ua/2013/11/08/nosql-world/`

Webber, J. (2013). *A little graph theory for the busy developer* [PowerPoint]. Graphconnect 2013, Chicago. Retrieved from `http://www.slideshare.net/neo4j/a-little-graph-theory-for-the-busy-developer-jim-webber-graphconnect-chicago-2013`

Weinberger, C. (2015). *Native multi-model can compete with pure document and graph databases.* Retrieved from `https://www.arangodb.com/`

2015/06/multi-model-benchmark/#Appendix