TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

IDU70LT

Oleg East 111379IAPM

COMPARATIVE ANALYSIS OF RELATIONAL AND GRAPH DATABASE BEHAVIOR ON THE CONCRETE WEB APPLICATION

Master's thesis

Supervisor: Ingmar Pappel
Master of Science
Lecturer

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

IDU70LT

Oleg East 111379IAPM

RELATSIOON- JA GRAAFANDMEBAASI VÕRDLEMINE KONKREETSE VEEBIPÕHILISE RAKENDUSE PEAL

Magistritöö

Juhendaja: Ingmar Pappel
Magistrikraad
Lektor

Tallinn 2016

Declaration

Herewith I declare that this thesis is based on my own work. All ideas, major views and data from different sources by other authors are used only with a reference to the source. The thesis has not been submitted for any degree or examination in any other university.

---------- ----------

(*date*) (*signature*)

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Comparative analysis of relational and graph database behavior on the concrete web application

Abstract

This thesis is written in English and is 68 pages long, including 6 chapters, 35 figures and 17 tables.

The first commercially available RDBMS was released in 1979 and has been dominating for storing and retrieving data for a long time until nowadays. [8] But since the amount of storing data is growing from day to day, the way of representing it is becoming more and more complex, because of dependence on a rigid schema which makes it difficult to add new relations between data. One of solutions to this problem is to use the graph database which was specially developed to solve such kind of problems, as a graph is a natural way of storing connections between objects. The Graph databases is a more modern concept of storing data and it is already used by a lot of known companies today, because the network data is simpler to represent as nodes are connected by relations and not by a large number of joined together tables.

The aim of this work is to try out the new concept of representing and storing data as graph principles on the developing web application which uses relational database management system. Software application itself represents a dictionary with words and lexemes connected by relations. The idea of it is to show how words are connected together in a dictionary. So the final results are shown as a graph with lexemes and relation strengths. The realization was made with relational database PostgreSQL. That is why an idea to try out a new concept with some graph database for this application has appeared.

In this work there will be a short introduction of the application, its data model, use cases and comparison of two different database approaches. We will try to change the codebase of our application so it would work with two databases in parallel. For this work we have chosen most popular graph database Neo4j. Ultimately, we would test our application for the response time, CPU and memory usage.

Relatsioon- ja graafandmebaasi võrdlemine konkreetse veebipõhilise rakenduse peal

Annotatsioon

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 68 leheküljel, 6 peatükki, 35 joonist, 17 tabelit.

Esimene kaubanduslikult saadav relatsioonanmebaas ilmus aastal 1979. [8] Tänasepäevani see oli domineeriv mudel andmete hoidmise ja hankimiseks. Kuid salvestatud andmete maht kasvab iga päev, selle esindamise viis muutub ka keerulisemaks. See juhtub sellepärast, et mudel on väga sõtluv skeemast, mis teeb raksemaks uute sõltuvuste lisamist. Üks võimalikutest lahendustes on kasutada graafiandmemudeli, mis oli spetsiaalselt arendatud andmebaasi mudeli lihtsustamiseks ja sarnase probleemi lahendamiseks, sest graaf on loomulik viis andmete sõltuvuste salvestamiseks. Graafiandmebaasid on kaasaegne kontseptsioon andmete salvestamiseks ja on juba kasutusel paljudes tuntutes ettevõtetes, sest võrguandmed on palju lihtsam esindada sõlmedena mis on ühendatud seostega, aga mitte suure hulga liitunud tabelitena.

Selle töö eesmärk on proovida uue kontseptsiooni andmete salvestamiseks ja esindamiseks graafiprintsiibidena juba olemasolevas arenevas rakenduses, mis kasutab tavalise relatsioonandmebaasi. Tarkvararakendus ise on sõnastikute hulk, kus on sõnad ja sõnade lekseemid ühendatud seostega. Selle idee on näidata graafina, kuidas sõnad on seotud üksteisega sõnastikus ja kui tugevad sõltuvused on erinevatel sõnadel. Rakendus on tehtud relatsioonandmebaasiga PostgreSQL ja just sellepärast tuli idee proovida uue graafimudeli andmete salvestamiseks.

Selle töö käigus tuleb lühike sissejuhatus arendatavast rakendusest, selle andmemudel, kasutuslood ja kahe erineva andmebaasi mudeli võrdlemine. Me muudame meie rakenduse koodi nii, et see töötaks kahe erineva andmebaasiga paraleelselt. Selleks me valisime ühe väga populaarse graafi andmebaasi Neo4j. Lõpuks me testime meie süsteemi reaktsiooniaja, CPU ja mälukasutust.

List of abbreviations and terms

Quantitative data	Data that can be quantified and verified, and is amenable to statistical manipulation. Quantitative data gives the definition, whereas qualitative data provides the description.
Lexeme	A meaningful linguistic unit that is an item in the vocabulary of a language.
D3.js	D3.js is a JavaScript library which helps you to represent data with powerful visualization components as dynamic and interactive graphical forms which can be run in a web browser.
RDBMS	Relational Database Management System.
PostgreSQL	The world's most advanced open source relational database.
Neo4j	The world's leading graph database.
MVC	Model-view-controller pattern.
Front-end	Interface between user and server side back-end.
Back-end	Application business logic and data management.
HTML	Hypertext Markup Language.
Data access layer	A layer of a computer program which provides simplified access to data.
Hibernate	Object-relational mapping framework for Java language.
IoC	Inversion of control pattern.
AOP	Aspect-oriented programming.

ORM	Object-relational mapping.
SQL	Structured Query Language.
Cron job	Time-based job scheduler.
JSON	An open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.
Depth-first and breadth-first	An algorithm for traversing or searching tree or graph data structures.
Lucene	Free and open-source information retrieval software library.
Directed acyclic graph (DAG)	A directed graph with no directed cycles
Eden space	Part of Java Heap where the JVM initially creates any objects

List of figures

List of tables

Table of Contents

1. Introduction

Current work is made on the basis of developing application named Qlaara. Application itself represents a web dictionary where users can see the relations between different lexemes of words shown with their relations in a D3.js Force-Directed graph. Current project version was developed approximately 4 months with using a relational database PostgreSQL. It was chosen because it offers a powerful instruments and functionality for storing and managing data with using SQL.

1.1 Problem formulation

As all the main data which is represented in Qlaara is shown as a graph with a words and relatedness strength, became an idea to try out to store the same data neither in relational database but in a graph database. Because the data model is simple and introduced by a word which has one or more lexeme meaning, which in its time has one or more relation, so it can be shown as nodes with edges and properties. Neo4j was chosen as a graph database which will be used in this work.

1.2 Goal of the thesis and expected results

The main goal of this work is to compare behavior of two different databases with the Qlaara application and measure the time of queries for retrieving data from the database and displaying it for the final user. Of course the complexity of graph database's query language is also important, because it will take some time to learn its own syntax and start writing good queries.

2. Core concepts and previous experiments

2.1 Graph database principles

A graph is a data structure composed of edges and nodes. Nodes have properties and connections as relationships which also have properties. A traversal navigates a graph and identifies paths which order nodes. Figure 1 shows the main graph components.

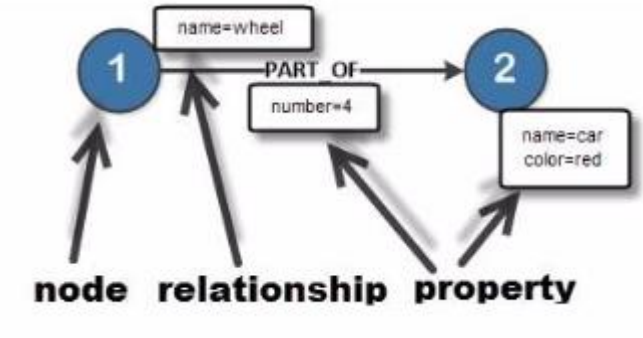

Figure 1. Graph structure

Graph database technology is an effective tool for modeling data when there is a focus on the relationships between entities. So modeling objects connected with relations means that data structure can be represented as a graph. A property graph is a common graph type supported by most systems. Property graphs are attributed, labeled and directed acyclic multi-graphs or DAG. A benefit to the multi graph is that it is the most complex implementation because every other type of graph consists of subsets of the property graph implementation. [13] It means that a property graph can effectively model all other graph types. This dynamic data model in which all nodes are connected by relations allows fast traversals along the edges between vertices. A benefit is the fact that traversals do not have to take into account nodes which are not connected, so the traversal time does not depend on the graph size.

Graph databases are really effective when working in areas where information about data interconnectivity or topology is important. In such applications the relations between data and the data itself are usually at the same level. Nowadays a lot of known companies use and develop their own implementations of the graph databases in different areas – bioinformatics, recommender systems, social networks and so on. For example, Google has BigTable,

Amazon has Dynamo, Facebook has Cassandra, which now belongs to Apache, LinkedIn has a Project Voldemort, Twitter has FlockDB and many more. Of course RDBMS can be also used for this needs, but in much more limiting and expensive way.

Getting information out of the graph needs what is known as a traversal or "walking" along the elements of the graph. One main difference between traversal and SQL query is that traversals are localized. This means that the size of the graph has no impact in traversal performance and in expensive JOIN operations. It is important to know that global indexes exist, but they are only used in finding the starting point. So it would require a linear scan of all elements without indices. Determining if a particular element has a particular property would require a linear scan of all elements at a cost of O(n) without indices, n being the number of elements in the collection. Alternatively, the cost of a lookup on an index is much lower at O(log2n). [13]

There has been no standardization of good graph database yet, so it has led to the huge amount of different implementations and frameworks for data interaction. It means that developers should learn a lot before getting started to use or understanding what is more suitable for the system. Gremlin and Cypher are two primary user languages for graph traversals. Gremlin is a domain-specific language, it is based on the top of the Groovy programming language and is closely tied to Java. It seeks to be a standard language that can be used in all major graph databases. Cypher in its way is a declarative language and inspired by SQL and is still under intense development.

2.2 Graph databases and NoSQL data models

NoSQL ("Not only SQL") movement brought many interesting solutions offering many different data models and database systems, that are suitable for different cases. Implying that in software or product design gives a lot of storage opportunities that could be applied based on the design. Using the data structure as modelled by developers has given a rise to the movement away from relational modelling towards aggregate models. An aggregate is a collection of data that we interact with as a unit. [16] That forms only one dedicated view of your data.

Most NoSQL databases store sets of disconnected aggregates and this makes it difficult to use them for connected data and graphs. One known strategy is to embed an aggregate's

identifier inside the field belonging to other aggregate – foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive. [15]

Graph databases handle fine-grained networks of information providing any perspective on your data that fits your use-cases. All key-value stores can always be represented as a graph. The same happens with document stores. The structured hierarchy of a document accommodates a lot of schema-free data that can easily be represented as a tree. Although trees are a type of graph, a tree represents only one projection of your data.

2.3 Previous experiments

Before the start we decided to gather some information of previous experiments with Neo4j and some relational database comparison. So we have found a couple of articles to be savvier.

2.3.1 Comparative analysis of relational and graph databases

The first article is written by Shalini Batra and Charu Tyagi where they tried to compare the different sides of a Neo4j and MySQL database management system. They took several evaluation parameters for comparison and made a little experiment on calculating the response time of both databases with the same tasks. So let us have a look at the comparison and experimental results.

The first parameter they took is the **level of support or maturity**. Level of support indicates a specific extent of technical assistance in the total range of assistance that is provided by an information technology product (such as a software product) to its customers. [9] So since the relational database management system exists for over the 30 years, while Neo4j version 1.1 was released in February 2010 and it is obvious that Neo4j is less stable and less mature. Relational databases have a unified language SQL and it does not differ much between implementations, whereas Neo4j's supported languages (SPARQL, Gremlin and Cypher Query) do. But the Neo4j is still growing and maturing and has not undergone the same rigorous performance testing as relational databases. [1]

Security is the second important point. It was said in the article that MySQL has extensive multi user support, however Neo4j does not have any built in mechanisms for managing security restrictions and multiple users. But since the article is 4 years old we decided to have

a look for the change log of Neo4j database and found that in version 2.2 at May 2015 a possibility of authentication was added. Also was added a full support of profiling. [10]

The last point is **flexibility**. The schema of relational database is fixed and it makes it difficult to extend other databases. Also it is very difficult to add relations between objects if you want to change database structure. For example, there is a structure like this:

1) Marko is a human and Fluffy is a dog.

2) Marko and Fluffy are good friends.

3) Human and dog are subclass of mammal.

It is very simple to represent such data in the both databases. But if we want to add a condition that Marko and Fluffy are mammals, it would be more difficult to do it with relational database compared to graph database. So with the Neo4j variant we just need to add two relations and two nodes, but with MySQL we have to change the structure of one table and add an additional one. These operations sometimes are very expensive, if you have a database with thousands of records and table relations.

A very important point was said in the article, "Neo4 has an easily mutable schema while Relational databases are less mutable. It has been theoretically said that relational model works best when there are a relatively small and static number of relationships between objects. When the data sets become larger they require expensive join operations because they search all of the data to find the data that meets the search criteria. The larger the data set is, the longer it takes to find matches. Conversely, a graph database does not scan the entire graph to find the nodes that meet the search criteria. It looks only at records that are directly connected to other records, increasing the number of nodes does not increase the retrieval time." [1] A proof experiment was carried out too.

Let us imagine that we have MySQL and Neo4j databases with such kind of data:

1)User: user_id, user_name

2)Friends: user_id, friend_id

3)Fav_movies: user_id, movie_name

4)Actors: movie_name, actor_name

And we test these databases with 3 simple queries:

S0: Find all friends of Esha.

S1: Find the favorite movies of Esha's friends.

S3: Find the lead actors of Esha's friends favorite movies.

The response time for these queries will be something like that.

Table 1. Query Results in Milliseconds.

No of objects	MySQL:S0	Neo4j:S0	MySQL:S1	Neo4j:S1	MySQL:S2	Neo4j:S2
100	19.56	8	33	12.65	111.334	19.57
500	281.38	10	333.96	17	620.56	21

And the test with the 100 and 500 of users.

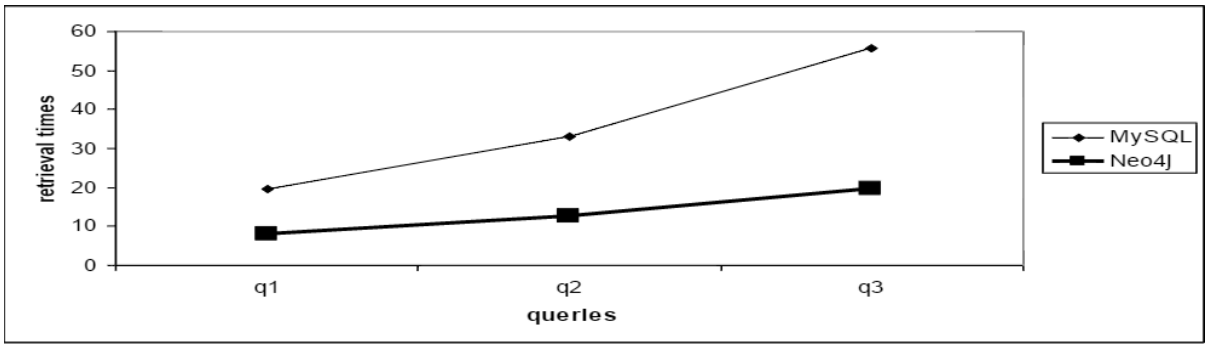

Figure 2. Retrieval time of queries by Neo4j and MySQL (100 objects).

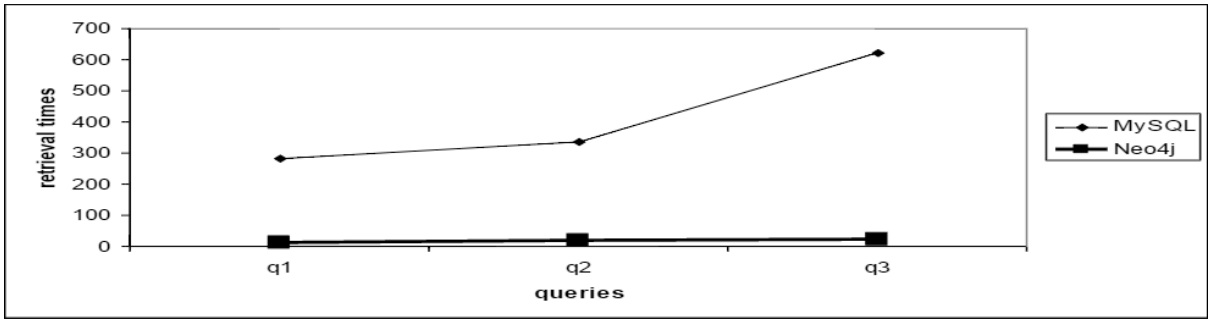

Figure 3. Retrieval time of queries by Neo4j and MySQL (500 objects).

In conclusion it was said that, despite of the advantages of relational databases in maturity and security, there is a huge advantage of graph databases in retrieval time. Also it is more

flexible in developing, without the need to restructure the schema again. Apart from that, Neo4j can be definitely used in commercial purposes.

2.3.2 A Comparison of a Graph Database and a Relational Database.

The next research article was written by a group of colleagues working in Department of Computer and Information Science in University of Mississippi. The goal is to determine whether a traditional relational database system like MySQL, or a graph database, such as Neo4j, would be more effective as the underlying technology for the development of a data provenance system. [11]

The article starts with very interesting principles which describe relational model. They are: atomicity, consistency, isolation and durability (ACID). So this set of governing principles guarantees database reliability, while NoSQL rejects them. However, NoSQL movement has its own potential flags that the data might be more suitable for a NoSQL system.

1) Having tables with lots of columns, each of which is only used by a few rows.
2) Having attribute tables.
3) Having lots of many-to-many relationships.
4) Having tree-like characteristics.
5) Requiring frequent schema changes.

So if your database model meets several of these criteria, it will be fitting to investigate NoSQL solutions to the provenance storage problem.

A total of twelve MySQL databases have been constructed for testing purposes. The databases only contain necessary structural information to represent the directed acyclic graph. Graphs were created to contain approximately 1000, 5000, 10000 and 100000 nodes. The information which was loaded consisted of random integers, random 8KB strings, and random 32KB strings.

Table 2 gives details about the databases and disk space required for each.

Table 2. Databases with sizes.

Database	Nodes count	Data type	MySQL size	Neo4j size
1000int	1000	Int	0.323M	0.428M
5000int	5000	Int	0.828M	1.7M
10000int	10000	Int	1.6M	3.2M
100000int	100000	Int	15M	31M
1000char8k	1000	8K char	18M	33M
5000char8k	5000	8K char	87M	146M
10000char8k	10000	8K char	173M	292M
100000char8k	100000	8K char	1700M	2900M
1000char32k	1000	32K char	70M	85M
5000char32k	5000	32K char	504M	406M
10000char32k	10000	32K char	778M	810M
100000char32k	100000	32K char	6200M	7900M

Both databases used full-indexing. And in general Neo4j databases was about 1.25 to 2 times the size of the corresponding relational databases. Only once the MySQL database was larger.

In total there were 6 queries generated to test these databases:

S0: Find all orphan nodes. This is for finding all nodes in the graph that are singletons, with no incoming edges and no out coming edges.

S4: Traverse the graph to the depth of 4 and count the number of reachable nodes.

S128: Traverse the graph to the depth of 128 and count the number of reachable nodes.

I1: Count the number of nodes whose payload data is equal to some value.

I2: Count the number of nodes whose payload data is less than some value.

C1: Count the number of nodes whose payload data contains some search string (length ranges from 4 to 8). Applies to character databases only.

Table 3 shows the results of the experiment:

Table 3. Query results, in milliseconds.

Database	MySQL S4	Neo4j S4	MySQL S128	Neo4j S128	MySQL S0	Neo4j S0
1000int	38.9	2.8	80.4	15.5	1.5	9.6
5000int	14.3	1.4	97.3	30.5	7.4	10.6
10000int	10.5	0.5	75.5	12.5	14.8	23.5
100000int	6.8	2.4	69.8	18.0	187.1	161.8
1000char8k	1.1	0.1	21.4	1.3	1.1	1.1
5000char8k	1.0	0.1	34.8	1.9	7.6	7.5
10000char8k	1.1	0.6	37.4	4.3	14.9	14.6
100000char8k	1.1	6.5	40.9	13.5	187.1	146.8
1000char32k	1.0	0.1	12.5	0.5	1.3	1.0
5000char32k	2.1	0.5	29.0	1.6	7.6	7.5
10000char32k	1.1	0.8	28.1	2.5	15.1	15.5
100000char32k	6.8	4.4	39.8	8.1	183.4	170.0

As it is seen in the table, traversal queries S0, S4 and S128 on Neo4j were clearly faster, sometimes even 10 times faster. The query to find orphan nodes resulted in fairly comparable results between two databases, that is why both systems are used to iterate the whole list to check each node.

Then a test with I1 and I2 queries was made and results are presented in the Table 4:

Table 4. Query results with integer values, in milliseconds.

Database	MySQL I1	Neo4j I1	MySQL I2	Neo4j I2
1000int	0.3	33.0	0.0	40.6
5000int	0.4	24.8	0.4	27.5
10000int	0.8	33.1	0.6	34.8
100000int	4.6	33.1	7.0	43.9

For the integer data relational database demonstrated the efficiency, because Neo4j uses Lucene for querying, and it is treated by default all data as a text. So these queries are not very fast, since conversion is required.

The test with the characters' data is shown in the Table 5 (d is the length of data):

Table 5. Query results with character values, in milliseconds.

Database	Rel	Neo	Rel	Neo	Rel	Neo	Rel	Neo	Rel	Neo
	d=4	d=4	d=5	d=5	d=6	d=6	d=7	d=7	d=8	d=8
1000char8k	26.6	35.3	15.0	41.6	6.4	41.6	11.1	41.6	15.6	36.3
5000char8k	135.4	41.6	131.6	41.8	112.5	36.5	126.0	33.0	91.9	41.6
10000char8k	301.6	38.4	269.0	41.5	257.8	41.5	263.1	42.6	249.9	41.5
100000char8k	3132.4	41.5	3224.1	41.5	3099.1	42.6	3077.4	41.8	2834.4	36.4
1000char32k	59.5	41.5	41.6	42.6	30.9	41.5	31.9	41.4	31.9	35.4
5000char32k	253.4	42.3	242.9	41.5	229.4	35.3	188.5	38.5	152.0	41.5
10000char32k	458.4	36.3	468.8	41.6	468.3	41.6	382.1	41.5	298.8	36.3
100000char32k	3911.3	41.4	4859.1	33.3	6234.8	37.3	4703.3	41.5	2769.6	41.5

For the last test with characters' data there were generated 4 databases with 8K characters data and four with 32K characters data. So when conducting tests on fully random data with letters, MySQL outperformed Neo4j. But with the more real-world data like words MySQL was much slower than Neo4j database.

In the second part of the article authors compared systems in maturity, ease of programming, flexibility and security.

In **maturity and level of support** the vote goes definitely for relational database. As it was written in previous article, relational databases are more mature and have a lot of support, while graph databases are younger and does not have much support.

The next comparison is the **ease of programming**. As soon as relational databases use SQL and it is very similar in different system implementations, graph databases are language-specific and have their own APIs. But the actual ease of programming is task-dependent, because, for example, graph traversals are simpler in graph databases. There are a lot of functionalities of doing it. In its turn, scanning a table for a particular attribute can be extremely easy with relational database. The last important thing is that relational databases have an ability to store graph data, while graph databases do not.

In **flexibility** the vote goes for graph database, because it has an easily mutable schema and relational system schema can be altered once the database is deployed, but doing so is a much more significant undertaking with graph database.

In **security** the relational databases are much more mature and have built-in multi-user support and functionalities for that. Neo4j in its turn have a user management at the application level. But as we found earlier it was in older versions.

In conclusion it was said that both systems performed well on the objective benchmark test. Graph database did better at the structural type queries and significantly better than the relational database in full-text character search. Speed issues related to index searching in Neo4j for numbers are related to the Lucene and its known problem. It is being developed for Lucene. But overall, for the data provenance project, it seems premature to use the graph database for a production environment.

2.4 Gathered information

If we compare these two articles we can find that both affect a very important aspect, they are the level of support or maturity, security, flexibility and ease of programming. As we got to know, there are some advantages of relational databases in security and level of support, because these databases are supported and developed for decades, since graph databases is a more modern concept.

In testing parts, the graph databases showed themselves deservedly for both experiments and authors claimed that graph databases should definitely be used in systems which structure contains a large number of relations.

3 Application design

Qlaara is a Java application, which is developing with using Spring Framework. It has simple MVC design for representing data to the end-user. Front end itself is performed with a Thymeleaf engine, which provides an elegant and well-formed way of creating templates. Some of the common solutions were made with JavaScript libraries for simple data requesting. Word relations are represented with a Force-Directed graph performed with a D3.js library. Application's main page is shown on the Figure 4.

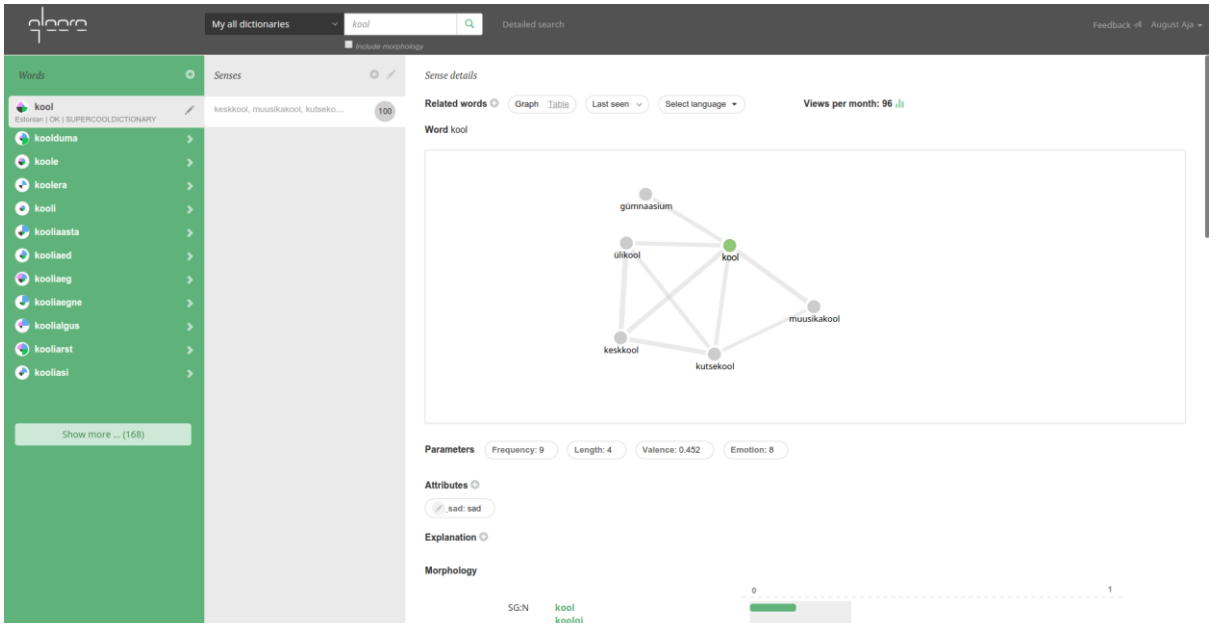

Figure 4. Qlaara main page. Graph view.

Qlaara's main search page view is very simple (Figure 4). It has four main sections, where every piece of information is represented. First of all, the navigation panel at the top, where users can search for a word in which they are interested in. Search is made by a word prefix. The left side sector shows the results of found words. The middle sector shows lexemes, which are connected to the selected word. The amount of lexemes of one word can be 1 or more and the last right sector shows the relations of the lexemes connected to the selected one. The amount is a configured option and it is set to 5. Graph view also shows the strength of the connection in range of 1 to 10 pixels. That means that 10 is 100% similarity or 1.

There are also possibilities to choose from the graph and table view, for more precise view of the connection strength. Here is the table view of main page (Figure 5).

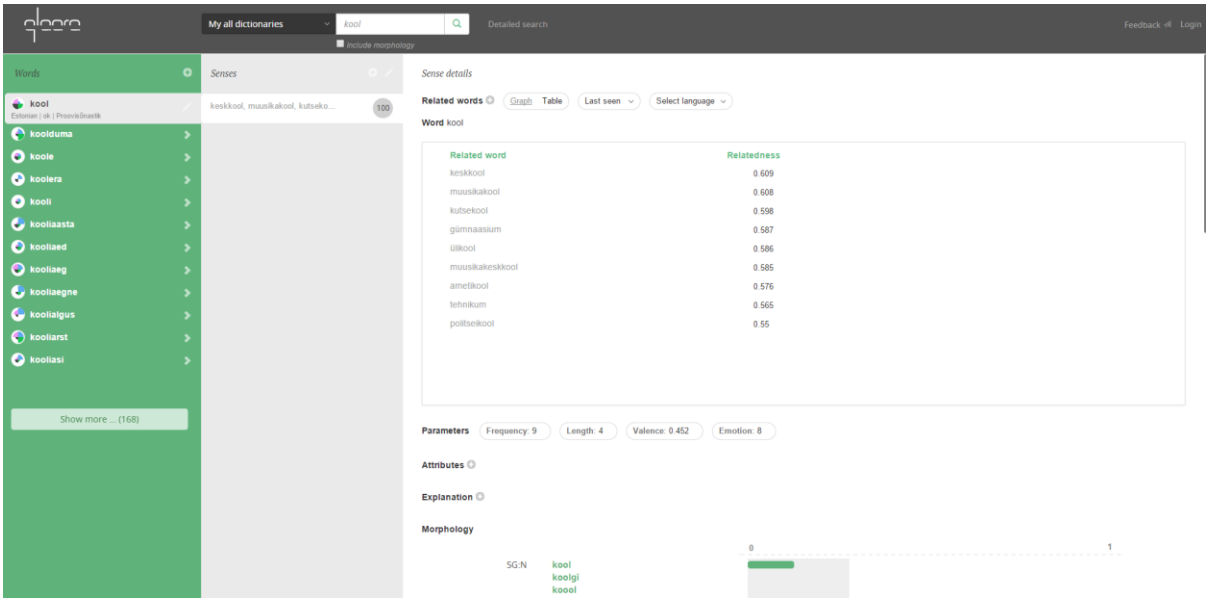

Figure 5. Qlaara main page. Table view.

In the Figure 5 you can see the table view, which shows the same relations as graph view, but it helps users to see the strength of the connection in numeric value. In addition to this there is an implemented language filter, which helps to sort words by their language in the dictionary.

3.1 Use cases

3.1.1 User interactions

Here are important cases to note. The search can be done in two ways: the simple one with a word match or word prefix and detail search by different parameters like word length, status, ratio, comments etc. Detailed search can be saved and shared with other users. After the search there should be the possibility to view found lexemes. The relations between can be viewed in a graph of table look and can be filtered by language. Every single lexeme has attributes, examples, explanation, morphology forms and comments. In addition, there should be possibility to add, edit, delete or vote for the liked lexeme. Lexemes and their parameters have different permissions to view, vote, and edit for the users.

The main system interactions between the system and the end user are shown in the Figure 6.

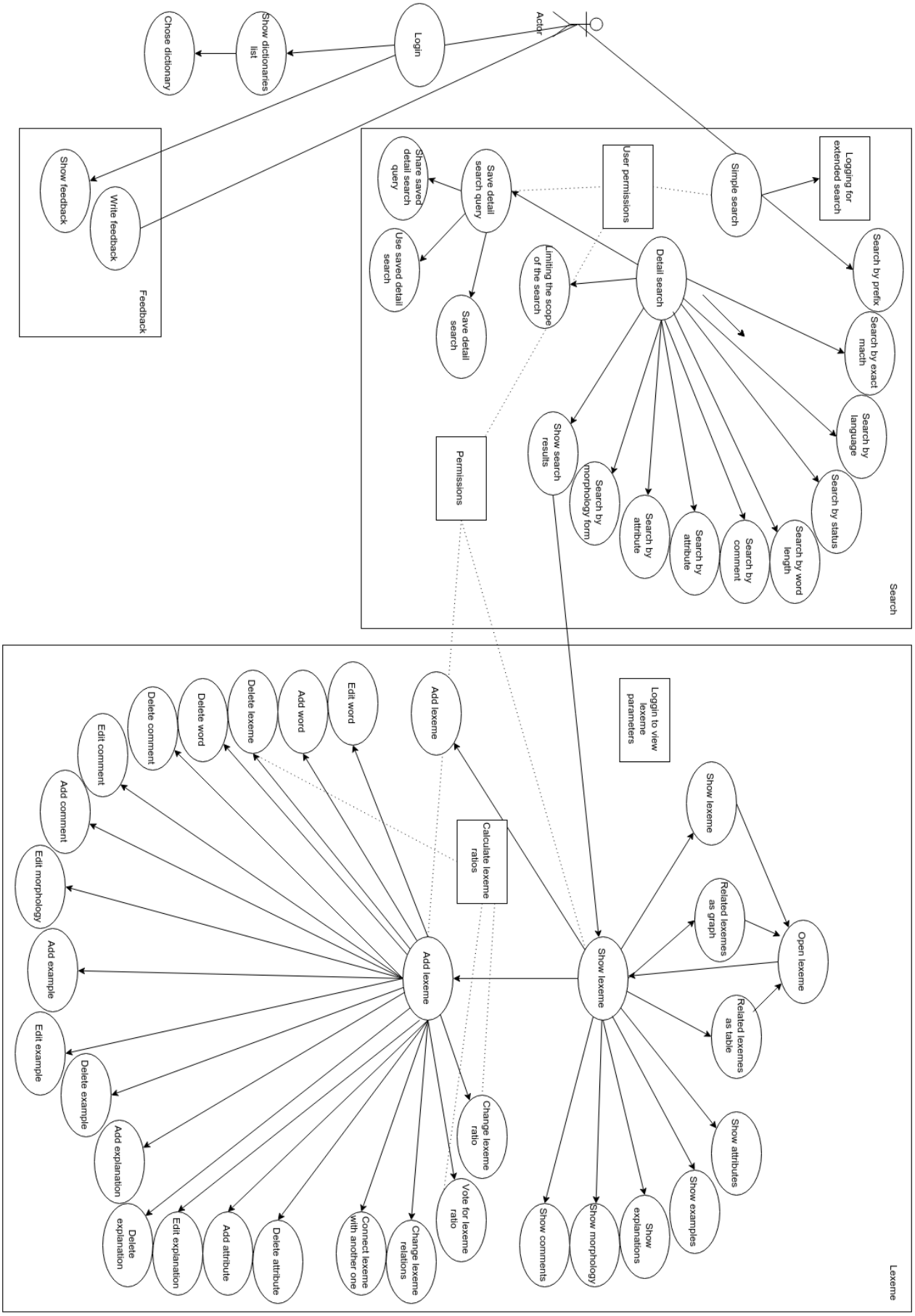

Figure 6. Main system use-cases.

3.1.2 User and dictionary management

There should be a possibility to use the system as anonymous or registered user. Apart from this, Users can also manage their own and public dictionaries. All possible interactions are shown in the Figure 7.

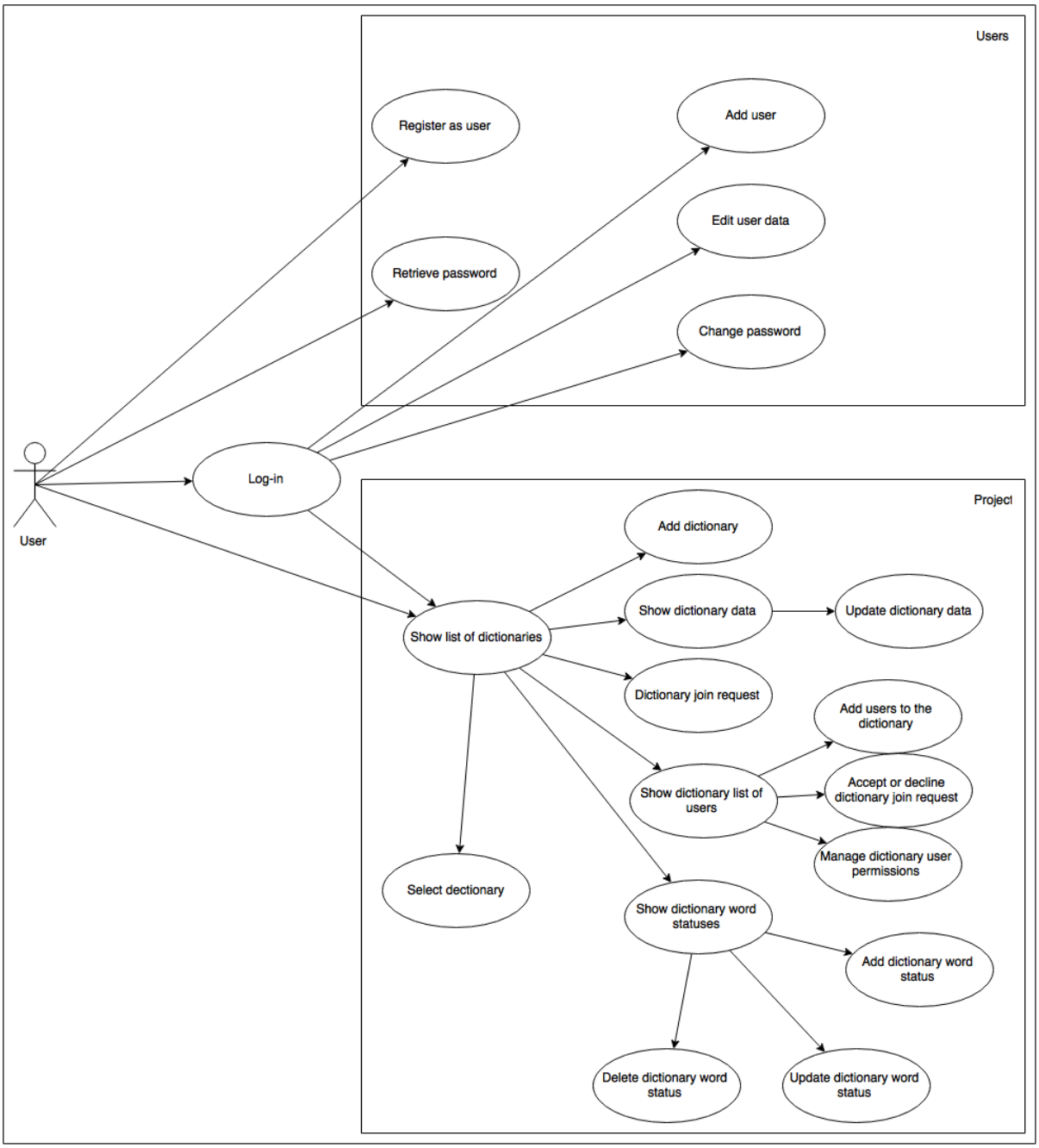

Figure 7. User and dictionary management.

As shown in the Figure 7 all users can register for the system users. User password can be restored and changed in the system. Users can add their own dictionaries and words. Dictionary owners can add other users to administer their dictionaries or users can send a request to become a dictionary administrator. On the dictionary page users can add, delete and edit word statuses and choose a default status. One of the interesting functionalities is that users can merge their accounts. For example, if a user has two accounts with different e-mails and he wants to remove one, but without losing his permissions and purchases.

3.2 Data model

The interesting part here is how Qlaara's data is represented in database. Figure 8 shows Qlaara's data structure representations in the PostgreSQL.

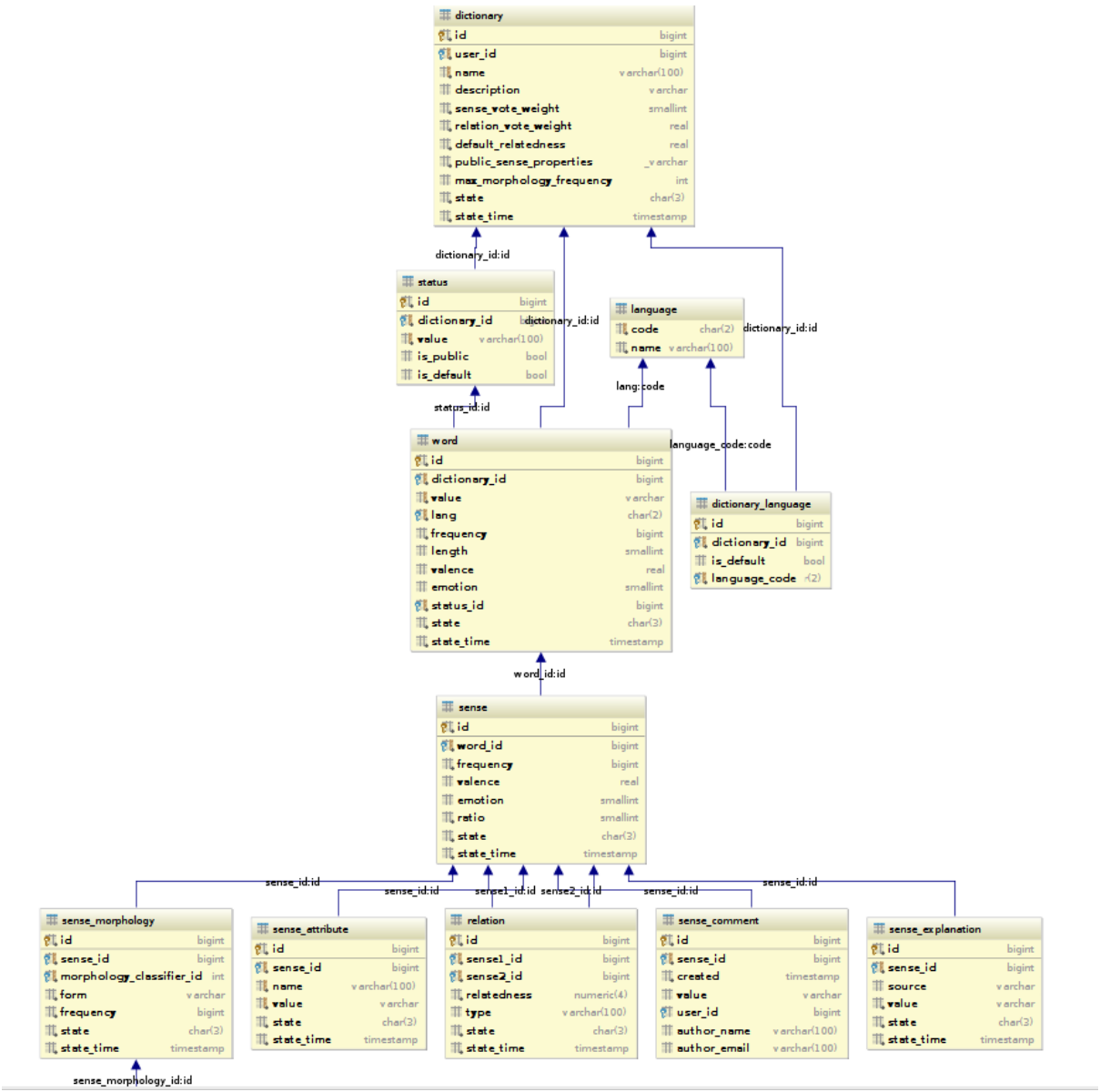

Figure 8. Qlaara's entity relationship diagram.

In the Figure 8 we can see, that there are dictionaries at the top of the model. It was meant because Qlaara can have more than one dictionary. There are public and user private dictionaries for holding different kinds of words. So dictionary can have zero or more words. Words can be in different languages and have such attributes like frequency of usage, length, valence and emotion. Also it has a status, which can be public or private. In its way word can

have zero or more lexemes or within our project we would name them senses. Senses are lexemes or the words' meaning. For example the word "type" has multiple meanings or lexemes. The first one is to write something on a typewriter or computer by pressing the keys. The second one is a category of people or things having common characteristics. Lexemes also have attributes like frequency, valence and emotion. Besides they have ratios. The ratio is the number, which defines the strength of the lexeme. The total sum of ratios of word's lexemes is also 100. So the mentioned two lexemes of word type can have ratios like 50/50 of 40/60 for example. The senses in its way have attributes, morphology forms, relations of other senses, comments of the users and explanations.

3.3 Data access layer

Spring offers good solutions for proper data access layers, using IoC pattern through templating and applying AOP interceptors for the ORM technologies [4]. The purpose of this layer is to get data from the database with querying and writing entities. In Qlaara all the database access methods and relational mapping are written in *DbService classes, which only can be used from the *Service classes. This allows Data access layer to be protected from the access outside of the layer. Hibernate provide option to execute native SQL queries through the use of SQLQuery object. The default method for accessing data is very simple, shown in the Figure 9:

```
335     public int getWordsCount(String prefix, Status status) {
336
337         String sqlQueryStr =
338                 "select count(w.id) cnt "
339                 + "from "
340                 + WORD_TABLE_NAME + " w, "
341                 + DICTIONARY_TABLE_NAME + " d "
342                 + "where "
343                 + "d.id = w.dictionary_id "
344                 + "and d.status = :status "
345                 + "and w.value like :prefix "
346                 + "and w.status = :status ";
347         Session session = sessionFactory.getCurrentSession();
348         SQLQuery sqlQuery = session.createSQLQuery(sqlQuerySt
349         sqlQuery.addScalar("cnt", new IntegerType());
350         sqlQuery.setString("prefix", prefix + "%");
351         sqlQuery.setString("status", status.name());
352         Integer count = (Integer) sqlQuery.uniqueResult();
353
354         return count;
355     }
```

Figure 9. Qlaara's default Hibernate native query.

This method (Figure 9) is using a native SQL query. As you can see it is very simple and handy, because we can execute database specific queries that are not supported by Hibernate API.

Here we are using Session.createSQLQuery(String query) to create the SQLQuery object and execute it. This simple method counts the number of words in dictionaries by word prefix and status. As a result we only need the number, so we execute uniqueResult() method to get the Integer number. But it also allows us to get objects straight from the database.

So let us have a look at the default Qlaara method with getting the list of objects or, in our case, the list of words (Figure 10).

```
357     public List<Word> getWords(String prefix, Status status, int limit) {
358
359         String sqlQueryStr =
360                 "select w.id as id, w.value as value, w.lang as lang, w.status as status, d.name as dictionaryName "
361                 + "from "
362                 + WORD_TABLE_NAME + " w, "
363                 + DICTIONARY_TABLE_NAME + " d "
364                 + "where "
365                 + "d.id = w.dictionary_id "
366                 + "and d.status = :status "
367                 + "and w.value like :prefix "
368                 + "and w.status = :status "
369                 + "order by value ";
370         if (limit > 0) {
371             sqlQueryStr += "limit :limit";
372         }
373         Session session = sessionFactory.getCurrentSession();
374         SQLQuery sqlQuery = session.createSQLQuery(sqlQueryStr);
375         sqlQuery.addScalar("id", new LongType());
376         sqlQuery.addScalar("value", new StringType());
377         sqlQuery.addScalar("lang", new StringType());
378         sqlQuery.addScalar("status", new StringType());
379         sqlQuery.addScalar("dictionaryName", new StringType());
380         sqlQuery.setString("prefix", prefix + "%");
381         sqlQuery.setString("status", status.name());
382         if (limit > 0) {
383             sqlQuery.setInteger("limit", limit);
384         }
385         sqlQuery.setResultTransformer(new AliasToBeanResultTransformer(Word.class));
386         List<Word> words = sqlQuery.list();
387
388         return words;
389     }
```

Figure 10. Qlaara's default ORM method.

In the Figure 10 method is used to get a list of words by prefix and status with the limit. It is used then we need the limited number of words, for example if we need to get top5 relations of the word to draw in the graph and we do not need others, because the graph area is not so huge to reflect all the words and this will not be clear for a final user. So, as in the previous example, we are using the same structure except from using uniqueResult() method. Here we need to get objects so we transform our results to the Word.class as setResultTransformer(new AliasToBeanResultTransformer(Word.class)) and this successfully finds the needed setters in our class.

The Word.class is shown on the Figure 11:

```
14  @NodeEntity
15  @JsonSerialize
16  @JsonInclude(JsonInclude.Include.NON_NULL)
17  public class Word implements Serializable {
18
19      private static final long serialVersionUID = 1L;
20
21      @GraphId
22      private Long id;
23
24      @Indexed(indexType = IndexType.FULLTEXT, indexName = "value")
25      private String value;
26
27      private List<Sense> senses;
28
29      private String lang;
30
31      private String logicalId;
32
33      private String status;
34
35      private String dictionaryName;
36
37      public Long getId() { return id; }
40
41      public void setId(Long id) { this.id = id; }
44
```

Figure 11. Qlaara's Word.class structure.

So all the parameters like id, value, language, status, dictionaryName are set as they have same names as in the Word.class (Figure 11) and query.

3.4 Loading data to the PostgreSQL database

As soon as there are millions of words in different languages and dictionaries, it is very hard to add words manually. So there was a csv file parser made in order to save available data from file to database. The basic concept of this is to read every single row in a file, to split it for pieces of data and save it to the database. See Figure 12.

```
DataParser dataParser = applicationContext.getBean(DataParser.class);

logger.debug("Reseting database...");
final String dbCreateFilePath = "./fileresources/sql/create_tables.sql";
dataParser.executeScriptsFile(dbCreateFilePath);
final String dbDefaultDataFilePath = "./fileresources/sql/data.sql";
dataParser.executeScriptsFile(dbDefaultDataFilePath);
final long dictionaryId = 1000;
dataParser.transformFromCsv(dataFilePath, dictionaryId, lang);
```

Figure 12. Qlaara's DataLoader.class

First of all, we should reset our database by creating new tables and indexes in our database (Figure 12). So as long as all changes in database are created in the tables.sql we can easy execute them before loading new data to the database. The second step is creating default records. They are test users, default dictionaries, languages etc. As in the beginning we had only Estonian dictionary's data, we chose the default dictionary id 1000 for it. The final step is to transform data and save it. Needed csv file format for it is simple and has 5 columns: word_value, sense_id, sense_ratio, related_sense_id and relatedness of these two lexemes.

The data parser is shown in the Figure 13.

```
private DataFileRow readLine(BufferedReader dataBufferedReader) throws Exception {

    String dataLine = dataBufferedReader.readLine();
    if (dataLine == null) {
        return null;
    }
    dataLine = StringUtils.remove(dataLine, '\"');
    String[] cellValues = StringUtils.split(dataLine, csvValueSeparator);

    if (cellValues.length != 6) {
        throw new QlaaraDataException("Wrong number of elements at data row \"" + dataLine + "\"");
    }

    try {
        String word = cellValues[1];
        long senseId = Long.parseLong(cellValues[2]);
        int senseRatio = Integer.parseInt(cellValues[3]);
        long relatedSenseId = Long.parseLong(cellValues[4]);
        float relatedness = 0;
        if (NumberUtils.isNumber(cellValues[5])) {
            relatedness = Float.parseFloat(cellValues[5]);
        }
        return new DataFileRow(word, senseId, senseRatio, relatedSenseId, relatedness);

    } catch (Exception e) {
        throw new QlaaraDataException("Illegal data format at data row \"" + dataLine + "\"", e);
    }
}
```

Figure 13. Qlaara's DataParser.class.

DataFileRow is an object made specially for holding an information about the parsed file row.

Then, when we got a list of DataFileRows with the size equal to the file rows we could easily start saving them to our tables. We already got a default dictionary where our words would be related, consequently then we had to save words, senses and relations in tables. We will do it one by one.

First of all, we create words in database (Figure 14).

```
public void createWord(Long id, Long dictionaryId, String word, String lang, Status status) {

    Session session = sessionFactory.getCurrentSession();
    String sqlQueryStr = "insert into " + WORD_TABLE_NAME + " (id, dictionary_id, value, lang, status) values (:id, :dictionaryId, :value, :lang, :status)";
    SQLQuery sqlQuery = session.createSQLQuery(sqlQueryStr);
    sqlQuery.setLong("id", id);
    sqlQuery.setLong("dictionaryId", dictionaryId);
    sqlQuery.setString("value", word);
    sqlQuery.setString("lang", lang);
    sqlQuery.setString("status", status.name());
    sqlQuery.executeUpdate();
}
```

Figure 14. Saving words into the database.

As we discussed in Data access layer paragraph we simply define native sql query with giving it parameters: dictionary_id, word_value, word_language and the status (Figure 14).

Status is the value marked as Active or Invalid, as long as we did not delete data from database straightly we define it as invalid. Further we can delete marked data with asynchronous cron service.

Next we save senses in the database (Figure 15).

```
public void createSense(Long id, Long wordId, Integer ratio, Status status) {

    Session session = sessionFactory.getCurrentSession();
    String sqlQueryStr = "insert into " + SENSE_TABLE_NAME + " (id, word_id, ratio, status) values (:id, :wordId, :ratio, :status)";
    SQLQuery sqlQuery = session.createSQLQuery(sqlQueryStr);
    sqlQuery.setLong("id", id);
    sqlQuery.setLong("wordId", wordId);
    sqlQuery.setInteger("ratio", ratio);
    sqlQuery.setString("status", status.name());
    sqlQuery.executeUpdate();
}
```

Figure 15.Saving senses into the database.

Here is the same pattern. Senses have word_id, this is an id of related word, sense_ratio, discussed in the Data model and the status (Figure 15).

Finally, we save relations. See Figure 16.

```
public Long createRelation(Long sense1Id, Long sense2Id, Float relatedness, Status status) {

    Session session = sessionFactory.getCurrentSession();
    String sqlQueryStr = "insert into " + RELATION_TABLE_NAME + " (sense1_id, sense2_id, relatedness, status) "
            + "values (:sense1Id, :sense2Id, :relatedness, :status) returning id";
    SQLQuery sqlQuery = session.createSQLQuery(sqlQueryStr);
    sqlQuery.addScalar("id", new LongType());
    sqlQuery.setLong("sense1Id", sense1Id);
    sqlQuery.setLong("sense2Id", sense2Id);
    sqlQuery.setFloat("relatedness", relatedness);
    sqlQuery.setString("status", status.name());
    Long id = (Long) sqlQuery.uniqueResult();
    return id;
}
```

Figure 16. Saving relations into the database.

Relations have sense1_id and sense2_id. These are ids of related senses. For example, a school and a university. The relatedness is the number between 0 and 1 and define the strength of the connection. As an example a school and a university has stronger connection than a school and a prison (Figure 16).

Finally, we got a database with a table of 90000 words, 90000 senses and 4410000 records with relations.

4 Database design and development

This section will be the main part of the work. Here we will make our changes by connecting same application layer with both PostgreSQL and Neo4j databases. The idea is to write two services which will work with different databases and that can retrieve different amount of data for further testing.

4.1 Neo4j installation

First of all, we need to install Neo4j database. It is very simple we just need to download the latest release from the manufacturer's server and then start it. All the instructions are on www.neo4j.com.

4.2 Cypher query language

Cypher is a declarative graph query language that allows efficient querying of the graph database. It is a relatively simple and very powerful language, but still different from SQL.

The creators of Cypher claim that it is very simple as for developers as for operations professionals, so even complex things are possible with it. The language itself is based on the English prose and most of the keywords like WHERE and ORDER BY are inspired by SQL. Pattern matching borrows expression approaches from SPARQL and some of the collection semantics have been borrowed from languages such as Haskell and Python.

4.3 Loading data to the Neo4j database

So when we have our Neo4j database up and running let us try to load some data to it.

It took a couple of times to set up libraries and try out a couple of examples found on the Internet. But after a while we were able to write solution.

As with relational database, first of all after the shutdown we will need to reset our database simply be deleting the database file. It was made for that case if we had already something in it after a couple of experiments. Figure 17 shows the initializing process.

```
@Value("${neo4j.datastore}")
String graphDbDirectory;

public void initGraphDatabaseService(boolean allowDbShutdown) throws IOException {

    this.allowDbShutdown = allowDbShutdown;
    if (allowDbShutdown) {
        if (graphDatabaseService != null) {
            graphDatabaseService.shutdown();
        }
        File graphDb = new File(graphDbDirectory);
        if (graphDb.exists()) {
            try {
                FileUtils.deleteRecursively(graphDb);
                logger.info("Removed existing database.");
            } catch (IOException e) {
                logger.error("Unable to delete database: " + graphDbDirectory);
            }
        }

        graphDatabaseService = new GraphDatabaseFactory().newEmbeddedDatabase(graphDbDirectory);
        neo4jTemplate = new Neo4jTemplate(graphDatabaseService);
    } else {
        try {
            startTransaction();
            neo4jTemplate.query("MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,r", Collections.EMPTY_MAP);
            endTransaction();
        } catch (Exception e) {
            logger.error("Error occurred when starting transaction.", e);
        }
    }
    startTransaction();
}
```

Figure 17. Resetting Neo4j database

First we close all active sessions in our database and delete the file with our data (see Figure 17). Then we create a new database using our graphDirectory. All the functionalities are very simple straight, and available by org.neo4j java libraries. So we do not even need to write our own methods for resetting database. All the examples can be found on the developers' page and developers' forums.

The second part of the statement is made for the test database, there we delete all records by hand with using a transaction method. This method matches all records in the database and then deletes them.

When our database is empty and ready to use we start to add data to it.

As earlier with relational database we will add all the words, next the senses and finally relations. See Figure 18.

```
logger.info("Start loading word and sense nodes ...");
do {
    csvRowValues = readLine(dataBufferedReader);
    try {
        if (csvRowValues == null || csvRowValues.length < 5) {
            logger.error("Line contains less than 5 tokens: line number=" + lineCounter);
            continue;
        }
        wordId = Long.parseLong(csvRowValues[2]);
        senseId = wordId;
        word = csvRowValues[1];
        ratio = Integer.parseInt(csvRowValues[3]);

        if (!senseIds.containsKey(senseId)) {
            Word wordNode = dbService.saveWord(wordId, word, lang);
            Sense senseNode = dbService.saveSense(senseId, wordNode, ratio);
            senseIds.put(senseId, senseNode);
        }

        if (lineCounter % 500000 == 0) {
            t2 = System.currentTimeMillis();
            logger.info("{} lines processed. Spent time {} ms", lineCounter, (t2 - t1));
        }
    } catch (Exception e) {
        logger.error("Unable to import line " + lineCounter + ". " + e.toString());
    }
    /*
    if (lineCounter == 100){
        break;
    }
    */
} while (csvRowValues != null);
```

Figure 18. Saving words and senses to Neo4j database.

The same way as with relational database. We parse all the rows in csv file and then save them (Figure 18). Neo4j library suggests pretty easy way for storing data. All we need is to define a method for storing our data with giving it needed values.

For saving words and senses we created such methods. See Figure 19.

```
public Word saveWord(long id, String wordValue, String lang) {

    Map<String, Object> properties = new HashMap<>();
    properties.put("value", wordValue);
    properties.put("logicalId", "w_" + id);
    properties.put("lang", lang);

    prepareTransaction();
    Word word = neo4jTemplate.createNodeAs(Word.class, properties);

    return word;
}

public Sense saveSense(long id, Word word, int ratio) {

    Map<String, Object> properties = new HashMap<>();
    properties.put("logicalId", "s_" + id);

    prepareTransaction();
    Sense sense = neo4jTemplate.createNodeAs(Sense.class, properties);

    Map<String, Object> relationProperties = new HashMap<>();
    relationProperties.put("ratio", ratio);

    Node wordNode = neo4jTemplate.getPersistentState(word);
    Node senseNode = neo4jTemplate.getPersistentState(sense);

    prepareTransaction();
    neo4jTemplate.createRelationshipBetween(wordNode, senseNode, RelationshipTypes.CONNECTED.toString(), relationProperties);

    return sense;
}
```

Figure 19. Methods for saving words and senses into the Neo4j.

What we have to do is to put our parameters into the map and then create an object into the database with the same class (Figure 19). The neo4jTemplate will create the needed node in the database. There is similar logic in creating word and sense, except of that while creating sense node we have to create a relation between sense and the word. That relation will be set with property ratio, so we could find needed senses by ratio. The connection will be named simply CONNECTED. Neo4j allows choosing connection names.

After words and senses we save relations. See Figure 20.

```
public void saveRelation(Sense sense1, Sense sense2, double relatedness) {
    Node sense1Node = neo4jTemplate.getPersistentState(sense1);
    Node sense2Node = neo4jTemplate.getPersistentState(sense2);

    Map<String, Object> relationProperties = new HashMap<>();
    relationProperties.put("relatedness", relatedness);

    prepareTransaction();
    neo4jTemplate.createRelationshipBetween(sense1Node, sense2Node, RelationshipTypes.RELATED.toString(), relationProperties);
}
```

Figure 20. Saving relations into the Neo4j database.

When words and senses are created, the next step is creating relations. The relation is a connection between two senses with the property relatedness. So we get needed senses from the database as nodes and create a relation between them. We name the connection as RELATED.

We created some additional methods for helping us with transactions in graph database. See Figure 21.

```
public void prepareTransaction() {
    incrementOperationsCount();
    if (shouldCreateNewTransaction()) {
        commitAndStartNewTransaction();
    }
}

private boolean shouldCreateNewTransaction() {
    if (operationsCount.get() >= operationsThreshold) {
        return true;
    }
    return false;
}

private void commitAndStartNewTransaction() {
    int totalNoOfTransactions = totalOperationsCount.addAndGet(operationsCount.getAndSet(0));
    //logger.debug("========== Refreshing transaction, total: " + totalNoOfTransactions);
    endTransaction();
    startTransaction();
}

private void incrementOperationsCount() { operationsCount.incrementAndGet(); }
```

Figure 21. Neo4j transaction creating methods

As we make thousands of transactions in a row we need to be sure that all of them will be delivered to the database. So we commit every 100000 query to the database, end the transaction and start new.

4.4 Data access layer refactoring

Now when we have our data in the database, let us rewrite a little data access layer so we could retrieve data as from PostgreSQL database as from Neo4j.

For testing purposes we would create some controller methods for retrieving data from both databases in JSON format.

4.4.1 Retrieving data from PostgreSQL database

There would be two methods same for PostgreSQL and for Neo4j. See Figure 22.

```
@RequestMapping(value = "searchlike/{wordValue}/{graphDepth}/{relationLimit}", method = RequestMethod.GET)
public void searchLike(
        @PathVariable("wordValue") String wordValue, @PathVariable("graphDepth") int graphDepth, @PathVariable("relationLimit") int relationLimit,
        HttpServletResponse response) throws Exception {

    composeResult(response, wordValue, graphDepth, relationLimit, false);
}

@RequestMapping(value = "searchexact/{wordValue}/{graphDepth}/{relationLimit}", method = RequestMethod.GET)
public void searchExact(
        @PathVariable("wordValue") String wordValue, @PathVariable("graphDepth") int graphDepth, @PathVariable("relationLimit") int relationLimit,
        HttpServletResponse response) throws Exception {

    composeResult(response, wordValue, graphDepth, relationLimit, true);
}
```

Figure 22. Controller methods for retrieving data from databases.

In these two methods we will test if there are any database performance differences in searching for exact match or by word prefix (Figure 22). These methods will be mapped for reldb/** and graphdb/** controllers. Methods themselves have 3 parameters. They are word value or prefix, graph depth, and relations limit. Graph depth is the parameter which defines how deep we would like to search, as sense have their own senses we would search for any found sense relations their own relations, so the graph can grow exponentially. That is why we are adding this parameter in order to limit the size of the graph. The last parameter is the relation limit; this will limit the count of the relations connected to one sense. As well as one sense can have zero or more relations, for testing we would like to limit this parameter.

Then let us have a look at the SQL and this is the most problematic place here. As far as we need a recursive search in database, because every single found related sense can have its own relations. The second problem is that we cannot simply get needed word as object from database, because word has a sense as object and sense has relations as object, in their way relations have words as object and so on. So the solution here is to make so called graph tuples, which would have all the needed ids and represent nodes in the graph.

Graph tuple will have a structure, as shown on the Figure 23.

```
public class GraphTuple {

    private String dictionaryName;

    private Long wordId;

    private String word;

    private String wordLang;

    private String wordStatus;

    private Long relatedWordId;

    private String relatedWord;

    private String relatedWordLang;

    private Long senseId;

    private Long relatedSenseId;

    private Integer senseRatio;

    private Float relatedness;

    private Integer step;
```

Figure 23. Graph tuple for retrieving data from PostgreSQL.

As shown in Figure 23 graph tuple is having all the needed information about the words, senses and relations. The step here shows if this node is selected or not. This information is needed for composing these nodes as word, sense and relation objects, also for drawing the graph.

Here is an SQL for retrieving these tuples. See Figure 24.

```
String sqlQueryStr =
        "with recursive graph_node("
            + "dictionary_name, word_id, word_value, word_lang, word_status, related_word_id, "
            + "sense_id, related_sense_id, sense_ratio, relatedness, step, pairs, cycle"
            + ") as ( "
            + "(select "
                + "d1.name, w1.id, w1.value, w1.lang, w1.status, w2.id, "
                + "s1.id, s2.id, s1.ratio, r.relatedness, 1, array[row(w1.id, w2.id)], false "
            + "from "
            + WORD_TABLE_NAME + " w1, "
            + WORD_TABLE_NAME + " w2, "
            + DICTIONARY_TABLE_NAME + " d1, "
            + SENSE_TABLE_NAME + " s1, "
            + SENSE_TABLE_NAME + " s2, "
            + RELATION_TABLE_NAME + " r "
            + "where "
            + wordMatchFragment
            + dictionary1StatusMatchFragment
            + "and d1.id = w1.dictionary_id "
            + wordStatusMatchFragment
            + "and w1.id = s1.word_id "
            + "and w2.id = s2.word_id "
            + senseStatusMatchFragment
            + "and s1.id = r.sense1_id "
            + "and s2.id = r.sense2_id "
            + relationStatusMatchFragment
            + "order by r.relatedness desc) "
        + "union all "
            + "(select "
                + "d2.name, w2.id, w2.value, w2.lang, w2.status, w1.id, "
                + "s2.id, s1.id, s2.ratio, r.relatedness, g.step + 1, pairs || row(w2.id, w1.id), row(w2.id, w1.id) = any(g.pairs) "
            + "from "
            + WORD_TABLE_NAME + " w1, "
            + WORD_TABLE_NAME + " w2, "
            + DICTIONARY_TABLE_NAME + " d2, "
            + SENSE_TABLE_NAME + " s1, "
            + SENSE_TABLE_NAME + " s2, "
            + RELATION_TABLE_NAME + " r, "
            + "graph_node g "
            + "where 1=1 "
            + dictionary2StatusMatchFragment
            + "and d2.id = w2.dictionary_id "
            + wordStatusMatchFragment
            + "and w1.id = s1.word_id "
            + "and w2.id = s2.word_id "
            + senseStatusMatchFragment
            + "and s1.id = r.sense1_id "
            + "and s2.id = r.sense2_id "
            + relationStatusMatchFragment
            + "and r.sense1_id = g.related_sense_id "
            + "and g.step < :depth "
            + "order by r.relatedness desc) "
        + ") "
        + "select "
        + "dictionary_name dictionaryName, "
        + "word_id wordId, "
        + "word_value word, "
        + "word_lang wordLang, "
        + "word_status wordStatus, "
            + "related_word_id relatedWordId, "
            + "sense_id senseId, "
            + "related_sense_id relatedSenseId, "
            + "sense_ratio senseRatio, "
            + "relatedness relatedness, "
            + "step step "
            + "from graph_node "
            + "where not cycle"
            ;
return sqlQueryStr;
```

Figure 24. Hibernate SQL for retrieving graph tuples from PostgreSQL.

This is the main part of this layer. As we can see in the Figure 24, it is not the simplest SQL query as far as it has a lot of connections between tables, temporary tables connected together. For a skilled developer this would take a half of the day or even more to write such SQL query.

The problem here is that this is a recursive SQL and we cannot limit the number of found relations. So it would search for all relations in the database.

We will add an if clause for exact and the prefix search (Figure 25)

```
String wordValueMatchFragment;
String wordValueMatchCriterion;
if (exactMatch) {
    wordValueMatchFragment = "w1.value = :wordValue ";
    wordValueMatchCriterion = wordValue;
} else {
    wordValueMatchFragment = "w1.value like :wordValue ";
    wordValueMatchCriterion = wordValue + "%";
}
```

Figure 25. SQL's exact and prefix search.

And this is not the end point of getting data. After that we need to compose our tuples for the Word, Sense and Relation objects. So in the final result we would have a single word object which will contain all the needed information.

First of all, we need to know which tuples are root nodes and which are the connections. That is why we needed the step in the graph tuples. It is an integer value and it can have only 2 values 1 or 2. 1 means that it is a root tuple, and 2 meant that it is a relation.

The method is shown in the Figure 26.

```
public ResultBuilderData collect(List<GraphTuple> graphTuples) {

    Map<Long, List<GraphTuple>> graphTupleMap = new HashMap<~>();
    List<Long> rootWordIds = new ArrayList<~>();
    ResultBuilderData resultBuilderData = new ResultBuilderData();
    resultBuilderData.setGraphTupleMap(graphTupleMap);
    resultBuilderData.setRootWordIds(rootWordIds);

    List<GraphTuple> mappedGraphTuples;
    Long wordId;

    for (GraphTuple graphTuple : graphTuples) {
        wordId = graphTuple.getWordId();
        mappedGraphTuples = graphTupleMap.get(wordId);
        if (mappedGraphTuples == null) {
            mappedGraphTuples = new ArrayList<GraphTuple>();
            graphTupleMap.put(wordId, mappedGraphTuples);
            if (graphTuple.getStep() == null) {
                rootWordIds.add(wordId);
            } else if (graphTuple.getStep() == 1) {
                rootWordIds.add(wordId);
            }
        }
        mappedGraphTuples.add(graphTuple);
    }

    return resultBuilderData;
}
```

Figure 26. Collecting the root tuples.

Here we will create an object which will help us in collecting needed data. We named it ResultBuilderData (Figure 26) and it is having the map with keys as collected word root id's and the value as a list of connected to that word graph tuples. Also it has a list of root id's, this will simplify our work further.

The next step is to check whether the list of id is empty or not. This will tell us if we have found something or not. So we could easily return null as a result and not proceed with further steps.

The final step is a population of the graph tuples to the word, sense, relation objects. See Figure 27.

```
public Word populate(Long wordId, Map<Long, List<GraphTuple>> graphTupleMap, int relationLimit, int graphDepth, int graphStep) {
    Word word = null;

    if (graphStep <= graphDepth) {

        List<GraphTuple> mappedGraphTuples = graphTupleMap.get(wordId);
        if (mappedGraphTuples != null) {
            List<Sense> senses;
            List<Relation> relations;
            Sense sense;
            Relation relation;
            Word relatedWord;
            Long relatedWordId;
            int relationCount = 0;
            Set<Long> alreadyBindedRelatedWordIds = new HashSet<Long>();

            for (GraphTuple graphTuple : mappedGraphTuples) {
                relatedWordId = graphTuple.getRelatedWordId();
                if (alreadyBindedRelatedWordIds.contains(relatedWordId)) {
                    continue;
                }
                alreadyBindedRelatedWordIds.add(relatedWordId);
                if (word == null) {
                    word = convertWord(graphTuple);
                }
                relatedWord = populate(relatedWordId, graphTupleMap, relationLimit, graphDepth, graphStep + 1);
                if (relatedWord == null) {
                    continue;
                }
                senses = word.getSenses();
                if (senses == null) {
                    senses = new ArrayList<Sense>();
                    word.setSenses(senses);
                }
                sense = findSense(graphTuple.getSenseId(), senses);
                if (sense == null) {
                    sense = convertSense(graphTuple);
                    senses.add(sense);
                }
                relations = sense.getRelations();
                if (relations == null) {
                    relations = new ArrayList<Relation>();
                    sense.setRelations(relations);
                }
                relation = convertRelation(relatedWord, graphTuple);
                relations.add(relation);
                if (++relationCount >= relationLimit) {
                    break;
                }
            }
        }
    }
```

Figure 27. Populate graph tuples to the word object

This is the recursive method which calls himself as we need to compose the next connected word (Figure 27). The goal of this method is to return a word object which has senses, sense it their way will have relation objects. Relation objects will have word objects and so on, until the graph depth is exceeded.

In the Figure 28 shows the method, which composes the result object to the JSON

```
protected void composeResult(
        HttpServletResponse response, String wordValue, int graphDepth, int relationLimit, boolean exactMatch) throws Exception {

    logger.debug("Search by word \"{}\" with graph depth {} and relations limit {}", wordValue, graphDepth, relationLimit);

    Object result;

    if (exactMatch) {
        result = getDataService().getWord(wordValue, graphDepth, relationLimit, null);
    } else {
        result = getDataService().getWords(wordValue, graphDepth, relationLimit, null);
    }

    response.setHeader("Content-Type", "application/json; charset=UTF-8");
    response.setCharacterEncoding("UTF-8");

    ServletOutputStream responseOutputStream = response.getOutputStream();

    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.writerWithDefaultPrettyPrinter().writeValue(responseOutputStream, result);

    response.flushBuffer();
    responseOutputStream.close();
}
```

Figure 28. Compose the collected results as JSON.

So when the needed word object is composed for an exact search or list of word objects composed for prefix match we will return them to the end user as a JSON format (Figure 28). This is very comfortable format for us, because it is human-readable and it will be easy to test further our services.

4.4.2 Retrieving data from Neo4j database

So we already have a RelDbController class which was mapped to get PostgreSQL data. Here we will use the same pattern, apart from that we will create a DataService interface to map needed methods for relational and graph databases. See Figure 29.

```
public interface DataService {

    /**
     * Returns the specified word with senses and relations
     * according to specified graph depth and maximum relations limit
     */
    Word getWord(String wordValue, int graphDepth, int relationLimit, Status status) throws Exception;

    /**
     * Returns the identified word with senses and relations
     * according to specified graph depth and maximum relations limit
     */
    Word getWord(Long wordId, int graphDepth, int relationLimit, Status status) throws Exception;

    /**
     * Returns list of matching words with senses and relations
     * according to specified graph depth and maximum relations limit
     */
    List<Word> getWords(String prefix, int graphDepth, int relationLimit, Status status) throws Exception;

    /**
     * Returns count of active words matching the prefix
     */
    int getWordsCount(String prefix) throws Exception;

    /**
     * Returns plain list of active words matching the prefix
     */
    List<Word> getWords(String prefix, int limit) throws Exception;

    /**
     * Sets word to invalid status
     */
    void deleteWord(Long wordId) throws Exception;

    /**
     * Creates and returnes new sense id with associated word and first sense relation.
     */
    Long relateSense(Long wordId, Long senseId, Long relatedSenseId, Float relatedness) throws Exception;
}
```

Figure 29. DataService interface.

These methods in the interface will be mapped to retrieve data as from graph database, as from PostgreSQL (Figure 29).

The next step here is to implement the first getWord method in the interface. The problem here is that we cannot simply use standard repository methods suggested by neo4j repository. So as with PostgreSQL we will first get the needed words as objects and then traverse through the database for finding needed connections.

First of all, let us implement the repository methods which will help us to find the needed words. See Figure 30.

```
public interface WordRepository extends GraphRepository<Word> {

    @Query("MATCH (w:Word) WHERE w.value =~ {value} RETURN w")
    List<Word> findByValue(@Param("value") String value);

    @Query("MATCH (w:Word) WHERE w.value =~ {regex} RETURN w ORDER BY w.value")
    List<Word> findByValueRegex(@Param("regex") String value);

    @Query("MATCH (w:Word:Word {value:{0}}) RETURN w")
    Word getWordByValue(String word);

    @Query("MATCH (w:Word) RETURN count(w)")
    Long getWordNodeCount();

}
```

Figure 30. WordRepository interface.

The Cypher language is not very hard to understand so it takes a couple of minutes to start writing neo4j queries. The first and the second methods are similar, apart from that we will use first for the exact search and the second for regular expression search needed to search by prefix (Figure 30).

So in the first step we will use one of these methods to find the list of searching words. For the exact search we will have:

```
List<Word> words = wordRepository.findByValue(wordValue);
```

And in the second case:

```
List<Word> words = wordRepository.findByValueRegex(prefix + '.*');
```

As with relational database in the next step we will check whether the list is empty or not, so we could return a feedback to the user that the search did not give any results.

The next step is very important for understanding, so we will compose our results and travel through the graph with the TraversalDescription class to find needed data.

So we get a composing method in final results, as shown on the Figure 31.

```
@Transactional
public Word getRelationsFor(Word word, int depth) {

    int countNodes = 0;
    // Do not count the first connection level between word and sense
    depth = depth + 2;
    Node wordNode = neo4jTemplate.getPersistentState(word);
    RelatedWordsEvaluator calculateRelatedWords = new RelatedWordsEvaluator(depth);

    TraversalDescription traversalDescription = graphDatabaseService.traversalDescription()
            .depthFirst()
            .relationships(RelationshipTypes.CONNECTED)
            .relationships(RelationshipTypes.RELATED)
            .evaluator(Evaluators.toDepth(depth))
            .evaluator(calculateRelatedWords);
    Traverser traverser = traversalDescription.traverse(wordNode);

    for (Node node : traverser.nodes()) {
        countNodes++;
    }
    logger.info("TraversalDescription query all nodes count: " + countNodes);
    logger.info("TraversalDescription query sense relations count: " + calculateRelatedWords.getCountSenseRelations());

    return calculateRelatedWords.getWordWithRelations();
}
```

Figure 31. Composing relations of the Word object.

This step is very important and it needs some time to understand how it works (Figure 31). Although the Cypher language is very simple for learning the usage of the Spring Data Neo4j project is not so trivial.

The interesting part here is the TraversalDescription object, which helps us to travel through the database and collect the needed nodes. First of all we define which algorithm we will use to search for the data. It suggests depth first and breadth first search. We will use the first one. Next we will define all relationships to be traversed for any given node: CONNECTED and RELATED. And in the final step we define how deep we will go in the graph and what we will do with the collected data.

Now let us take a look on what we are doing while traversing the nodes. We created a helper class named RelatedWordsEvaluator. See Figure 32.

```
public class RelatedWordsEvaluator implements Evaluator {

    private static Logger logger = LoggerFactory.getLogger(GraphImportDbService.class);

    Word rootWord;
    Sense currentSense;
    Sense previousLevelSense;
    Relation currentRelation;
    List<Sense> rootSenses = new ArrayList<>();
    Map<Long, String> senseWords = new HashMap<>();
    int depth = 0;
    int countDistance4 = 0;
    int previousSenseDistance = 0;
    int countSenseRelations = 0;
    int countSenses = 0;

    public RelatedWordsEvaluator(int depth) { this.depth = depth; }

    public Word getWordWithRelations() {
        logger.info("TraversalDescription query all senses count: " + countSenses);
        rootWord.setSenses(rootSenses);
        return this.rootWord;
    }

    @Override
    public Evaluation evaluate(Path path) {

        final int distance = path.length();
        Node node = path.endNode();
        Relationship relationship = path.lastRelationship();

        // debugging:
        debugRow(path, node, relationship, distance);

        // create root word object
        if (distance == 0) {
            rootWord = new Word();
            rootWord.setId(node.getId());
            if (node.hasProperty("value")) {
                rootWord.setValue((String) node.getProperty("value"));
            }
            logger.info("Root node: " + node);
        }
        // root word senses
        else if (distance == 1 && node.hasLabel(GraphLabels.Sense) && relationship != null && relationship.isType(RelationshipTypes.CONNECTED)) {
            Sense rootSense = new Sense();
            rootSense.setId(node.getId());
            if (relationship.hasProperty("ratio")) {
                rootSense.setRatio((Integer) relationship.getProperty("ratio"));
            }
            rootSenses.add(rootSense);
            currentSense = rootSense;
            countSenses++;
        }
```

Figure 32. RelatedWordsEvaluator part 1.

Here we defined all the needed variables, which will help us with collecting data, but the main part here is to evaluate method which is overridden by Neo4j Evaluator class (Figure 32). That is the place where we collect needed data depending on the level of the search, the type of the node and the type of the connection.

The second part of RelatedWordsEvaluator is shown in the Figure 33.

```
// new related sense
else if (distance < depth && node.hasLabel(GraphLabels.Sense) && relationship != null && relationship
        .isType(RelationshipTypes.RELATED)) {

    // It's a new level sense relation
    if (distance > previousSenseDistance) {
        previousLevelSense = currentSense;
        //logger.debug("move up");
    }
    // move back in hierarchy
    else if (distance < previousSenseDistance) {
        int relationDepth = 1;
        previousLevelSense = rootSenses.get(rootSenses.size() - 1);
        while (relationDepth < distance - 1) {
            previousLevelSense = previousLevelSense.getRelations().get(previousLevelSense.getRelations().size() - 1).getSense();
            relationDepth++;
        }
        //logger.debug("move back to " + previousLevelSense.toString());
    }
    Sense sense = new Sense();
    sense.setId(node.getId());
    Relation relation = new Relation();
    relation.setId(relationship.getId());
    if (relationship.hasProperty("relatedness")) {
        relation.setRelatedness(((Double) relationship.getProperty("relatedness")).floatValue());
    }
    relation.setSense(sense);
    List<Relation> senseRelations = previousLevelSense.getRelations();
    if (senseRelations == null) {
        senseRelations = new ArrayList<>();
        previousLevelSense.setRelations(senseRelations);
    }
    senseRelations.add(relation);
    //logger.debug("add sense: " + node.getId());

    // currentSense and currentRelation is needed for getting Word relation from the next level.
    currentSense = sense;
    currentRelation = relation;

    countSenseRelations++;
    previousSenseDistance = distance;
    countSenses++;
}
// get related sense word
else if (node.hasLabel(GraphLabels.Word) && relationship != null && relationship.isType(RelationshipTypes.CONNECTED)) {
    Word word = new Word();
    word.setId(node.getId());
    if (node.hasProperty("value")) {
        word.setValue((String) node.getProperty("value"));
    }
    if (relationship.hasProperty("ratio")) {
        currentSense.setRatio((Integer) relationship.getProperty("ratio"));
    }
    currentRelation.setWord(word);
}
return Evaluation.INCLUDE_AND_CONTINUE;
```

Figure 33. RelatedWordsEvaluator part 2.

While the data is collected the node can be included to or excluded from the search, we can also continue or prune the search (Figure 33).

After all the data is collected in the single word object, we return it in JSON to the user, as it was made with PostgreSQL.

At last we acquired 4 URL's for testing:

- /reldb/searchexact/{word}/{graph depth}/{relations limit}
- /reldb/searchlike/{prefix}/{graph depth}/{relations limit}
- /graphdb/searchexact/{word}/{graph depth}/{relations limit}
- /graphdb/searchlike/{prefix}/{graph depth}/{relations limit}

5 Testing

To determine which database is better to use for such kind of systems, let us define what parameters are making our system better.

Response time. It is really important for the end user to get requested data fast. Nobody would wait minutes for the response. Even if the response takes several seconds it is really annoying if you would like to make dozens of requests.

CPU usage. Since our system should work not on the local machines only, but on the remote servers, so everybody could use it. It would be very nice if it was lightweight. Less the CPU usage is, the more users can use it, the more processes we can run on the server.

Memory usage. Here is the same problem. Less the memory usage is the better it is.

Environment settings:

- Memory: 8 GiB
- Processor: Intel Core i5-4310U CPU @ 2.00Ghz x 4
- OS: Ubuntu 15.4
- RDBMS: PostgreSQL 9.4.5
- GraphDB: Neo4j 2.2.0

Data:

- Word count: 90000
- Senses count: 90000
- Relations count: 4410000

5.1 Response time

Here we will measure the response time of the system with different databases, relations limit and the graph depth. First we will test the relational database by increasing the number of relations and graph depth. To get more precise results we will take an average of 3 tests for a response time.

Table 6. PostgreSQL response time with graph depth 2.

Test №	Graph depth	Relations limit	Query time (ms) x 3			Structure build (ms) x 3			Avg. total time (ms)
1	2	5	1166	1198	1144	1	2	1	1171
2	2	10	1157	1216	1320	1	1	1	1232
3	2	15	1136	1179	1156	1	1	1	1158
4	2	20	1188	1178	1208	1	1	2	1193
5	2	25	1144	1181	1364	1	2	3	1232

Table 7. PostgreSQL response time with graph depth 3.

Test №	Graph depth	Relations limit	Query time (ms) x 3			Structure build (ms) x 3			Avg. total time (ms)
1	3	5	3044	3092	3377	29	18	10	3190
2	3	10	3080	3025	3028	25	26	19	3068
3	3	15	3088	3057	3055	30	28	30	3096
4	3	20	3086	2999	3114	33	35	32	3100
5	3	25	3107	3005	3393	40	38	34	3906

As it was mentioned in the Paragraph 3.4.1 the recursive sql is searching for all relations in the database and there is no way to limit the connections in the SQL query time. This caused a problem, because the query does not work with the graph depth more than 3. The amount of data is so huge that it gives OutOfMemory exception. In fact, our system's requirements do not mean to work with data further when first level relations, because we show for the users only the connections of the searched word.

Let us have a look at the response time of the Neo4j database.

Table 8. Neo4j response time with graph depth 2.

Test №	Graph depth	Relations limit	Query time (ms) x 3			Structure build (ms) x 3			Avg. total time (ms)
1	2	5	164	105	130	32	32	31	165
2	2	10	164	106	102	32	31	35	157
3	2	15	185	132	102	34	30	30	171
4	2	20	155	137	116	30	30	30	166
5	2	25	194	120	132	44	44	32	189

Table 9. Neo4j response time with graph depth 3.

Test №	Graph depth	Relations limit	Query time (ms) x 3			Structure build (ms) x 3			Avg. total time (ms)
1	3	5	98	171	109	68	79	96	207
2	3	10	145	189	131	73	77	68	228
3	3	15	107	188	101	82	73	65	205
4	3	20	107	174	134	67	77	68	209
5	3	25	104	102	104	150	66	67	198

So it is very fast and despite of the PostgreSQL it works with very big graph depth numbers.

The difference is very huge. So it will be clearer if we translate the same data on the line graph (Figure 34, 35).

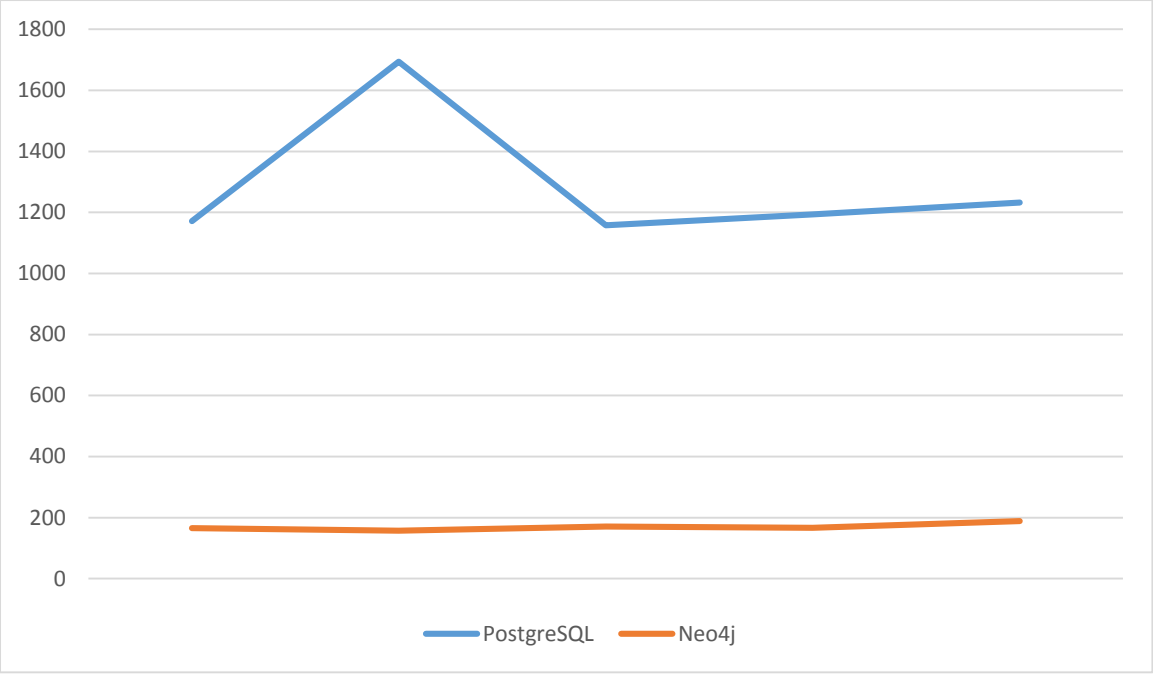

Figure 34. PostgreSQL vs. Neo4j with graph depth 2.

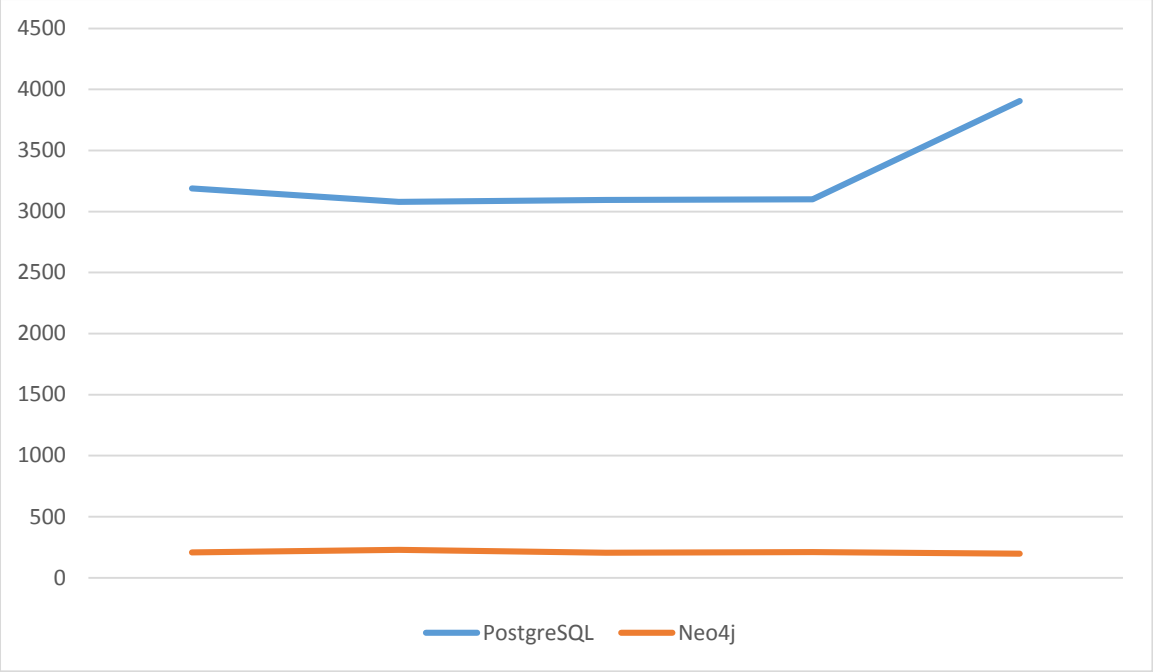

Figure 35. PostgreSQL vs. Neo4j with graph depth 3.

The result is to the fore. In case of the graph depth 2 (Figure 34) the Neo4j database is around 7 times faster, and in case of the graph depth 3 (Figure 35) approximately in 15 times. Superiority is huge.

What we can notice in that test is that the average response time does not depend on relations graph limit. It is because in both cases we pass all the nodes in the graph depth and sort the needed ones in the end. So it does not matter whether we limit with 5 relations or with 25. The response time is growing immediately with the graph depth and the PostgreSQL takes much more time to retrieve data.

5.2 CPU and the memory usage

First of all, we will find out what are the minimum hardware requirements for both databases.

It is said that PostgreSQL minimum requirements are:

- 64bit CPU
- 64bit Operating System
- 2 Gigabytes of memory
- Dual CPU/Core
- RAID 1

And for the Neo4j system should have:

- Intel Core i3
- 2 Gigabytes of memory
- 10 Gigabytes of SATA
- Ext4 or similar filesystem

The requirements are more or less similar. In fact, that servers and computers are nowadays much faster, it should not be a problem of using and installing these databases.

But let us have a look at the project and service performance in particular. For this test we installed a YourKit Java Profiler, so we could measure the CPU usage of our services working with Neo4j or PostgreSQL.

YourKit is an intelligent tool for profiling Java & .NET applications. It allows users to measure CPU and memory usage with maximum productivity and zero overhead and is used by a lot of big and small companies. It can be used at both development and production stages. [14]

First of all, let us measure the CPU usage of our application working with PostgreSQL.

Table 10. PostgreSQL CPU usage with graph depth 2.

Test №	Graph depth	Relations limit	CPU usage x 3			Avg. CPU usage
1	2	5	10%	3%	15%	9%
2	2	10	5%	7%	5%	6%
3	2	15	4%	3%	11%	6%
4	2	20	3%	3%	4%	3%
5	2	25	3%	4%	4%	4%

Table 11. PostgreSQL CPU usage with graph depth 3.

Test №	Graph depth	Relations limit	CPU usage x 3			Avg. CPU usage
1	3	5	16%	15%	14%	15%
2	3	10	22%	23%	19%	21%
3	3	15	22%	14%	10%	15%
4	3	20	16%	17%	23%	19%
5	3	25	20%	43%	37%	33%

And then with the Neo4j.

Table 12. Neo4j CPU usage with graph depth 2.

Test №	Graph depth	Relations limit	CPU usage x 3			Avg. CPU usage
1	2	5	6%	6%	13%	8%
2	2	10	10%	5%	6%	7%
3	2	15	5%	10%	6%	7%
4	2	20	12%	10%	4%	9%
5	2	25	5%	7%	9%	7%

Table 13. Neo4j CPU usage with graph depth 3.

Test №	Graph depth	Relations limit	CPU usage			Avg. CPU usage
1	3	5	21%	5%	10%	12%
2	3	10	9%	14%	7%	10%
3	3	15	16%	10%	8%	11%
4	3	20	6%	6%	5%	6%
5	3	25	9%	9%	6%	8%

As we can see the CPU usage does not increase with the number of relations. It grows if we go deeper to the graph. In case of the PostgreSQL the average CPU usage with graph depth 2 was 5,6% of all CPU and with graph depth 3 – 20,6%. The CPU usage with a Neo4j in graph depth 2 was a little more – 7,6%, but the impact of the depth was not bigger. With graph depth 3 the usage is 9,4%.

Because of that java is a JVM language all the memory usage is under Garbage Collector's control. So the collector decides whether to reclaim the memory or not. So it is very hard to draw conclusion, because it might be close to 0 MB or might take a half of GB. But let us try to measure the Eden Space of the JVM in the time after the request is made and take the difference between memory allocated before the request. In total we have allocated 2048 MB of RAM for our Java application. Let us have a look at the results with the PostgreSQL.

Table 14. PostgreSQL heap memory increase with graph depth 2.

Test №	Graph depth	Relations limit	Memory increase (MB)			Avg. memory increase (MB)
1	2	5	11	10	5	9
2	2	10	7	5	12	8
3	2	15	8	9	6	8
4	2	20	7	9	10	9
5	2	25	8	6	12	9

Table 15. PostgreSQL heap memory increase with graph depth 3.

Test №	Graph depth	Relations limit	Memory increase (MB)			Avg. memory increase (MB)
1	3	5	244	253	240	246
2	3	10	244	240	242	242
3	3	15	242	249	246	246
4	3	20	244	247	258	250
5	3	25	247	246	242	245

And the same tests for Neo4j.

Table 16. Neo4j heap memory increase with graph depth 2.

Test №	Graph depth	Relations limit	Memory increase (MB)			Avg. memory increase (MB)
1	2	5	168	167	161	165
2	2	10	164	161	164	163
3	2	15	164	161	163	163
4	2	20	160	163	164	162
5	2	25	160	161	160	160

Table 17. Neo4j heap memory increase with graph depth 3.

Test №	Graph depth	Relations limit	Memory increase (MB)			Avg. memory increase (MB)
1	3	5	180	179	180	180
2	3	10	185	183	179	182
3	3	15	181	183	183	182
4	3	20	181	184	187	184
5	3	25	181	180	184	182

So the average growth of the memory for the depth 2 is very impressive – 8,6 Megabytes. But it is increased a lot with the graph depth 3 and reached 245,8 Megabytes. And we can believe that it will grow rapidly if the data increases. In case of the Neo4j the usage is more or less stable and does not grow so much with the depth. 162,6 megabytes in first case and 182 with the depth 3.

6 Thesis summary

We have looked at the problem from different sides, starting with installation, learning a query language and writing server side services for retrieving data from two different databases to compare. We changed our project's codebase so it was working with two different databases in parallel. With two absolutely identical services we managed to test two databases for a response time, CPU and memory usage.

Let us sum up what results we got considering the work. As we saw in the test phase the response time of the Neo4j database was much faster compared to the PostgreSQL, so with the depth 3 it was approximately 15 times faster. It is a huge advantage, because such systems should handle thousands requests at a time. Also we noticed that PostgreSQL cannot handle requests with the graph depth more than 3, because of the huge amount of data and recursive SQL query. Concerning the CPU and memory usage, our tests showed that the PostgreSQL system consumption is rather exponential as it needs more CPU and memory with the growth of the amount of data, but it is less on the depth 2 especially with the memory usage. We believe that it depends on the processes after retrieving data, and traversing the graph.

As for developer it was not so hard to learn the basics of new technology, as it could have happened. The Neo4j documentation is clear enough for understanding. It is simple and interesting in use. It is definitely better for such kind of systems when you have a lot of different connection types and complex relations between objects, because they store relationships at the data level whereas RDBMS use a declarative approach. This is the main advantage of graph databases, because instead of doing tons of joins, they just pick a record and follow his relationships.

The main idea for this work was to check if the system can work better with using different database approach and how hard it is to refactor the system for that changes. The conclusion is that system can definitely work faster with such kind of data. Graph database algorithms are specially made for the system like ours, and should be used further.

6.1 Future work

The problem is that the latest developing project version today is far from the tested version and there are much more functionalities that it can handle beside of the showing the relations between word lexemes. So it is not so easy to just throw the database and rewrite the codebase. The best approach for this problem is to use graph database for the relations and nodes - for the fast querying and PostgreSQL database for the rest like user and dictionaries management systems.

Kokkuvõtte

Vaatasime probleemi peale erinevaatest suunatest, alustades andmebaasi paigaldamisega, päringu keele õppimisega ja serveri teenuste arendamisega. Me muutsime meie koodi, et see töötaks kahe erinevate andmebaasidega paraleelselt. Kahe teenusega, mis töötavad samamoodi me suutsime testida andmebaaside reaktsiooniajad, CPU ja mälukasutust.

Vaatame lühidalt mis tulemused me saime selles töös. Nagu me nägime testfaasil reaktsiooniaeg Neo4j andmebaasil oli palju kiirem, kui PostgreSQL, otsingu sügavusega 3 see oli 15 korda kiirem. See on väga suur eelis, sest sellised süsteemid peavad taotlema tuhandeid päringud. Samuti, me märkasime, et meie süsteem isegi ei saa hakkama otsinguga, mille sügavus on rohkem kui 3, sellepärast, et PostgreSQL'i paring on rekursiivne laadib väga palju andmeid. Vaadates CPU ja mälu kasutamisele, meie tesitid näitasin, et PostgreSQL-i süsteemi kasutamise kasv on pigem eksponentsiaalne ja kiiresti kasvab andmete suurusega, kuid on väiksem alguses võrreldes Neo4j-ga. Me arvame, et see on seotud protsessidega, mis toimuvad peale päringut ja Neo4j andmete tootlemisega.

Arendajana, see ei olnud nii keeruline õppida uue tehnoloogiat, nagu see võiks juhtuda. Neo4j dokumentatsioon on päris lihtne ja arusaadav. See on huvitav ja lihtne kasutamises. See on absuluutselt kõige parem lahendus selliste süsteemide jaoks nagu meie, kui teil on väaga palju erineväid ühendusi objektide vahel, sest andmebaas salvestab neid andmete tasemel, kusjuures RDBMS kasutab deklaratiivle lähenemise. See on kõige suurem eelis graafi andmebaasidel, sellepärast, et ei ole vaja teha tuhandeid tabelite ühendusi, andmebaas lihtsalt võtab objecti ja jälgib selle ühendust tiste objektidega.

Selle töö peamine ülesanne oli vaadata, kas saab teha süüsteemi paremini kasutades erineva andmebaasi süsteemi ja kui raske on muuta süsteemi selleks, et ta töötaks erineva andmebaasiga. Kokkuvõttes me saime, et süsteem saab kindlasti töötada paremini meie andmetegal. Graafi andmebaasid on spetsiaalselt tehtud selle süsteemide jaoks nagu meie, ja peaks võtta kasutusele edaspidises süsteemie arendamises.

Edasine töö

Probleem on selles, et praeguse süsteemi versioon on palju kaugemal testitud süsteemi versioonilt ja on lisatud palju rohkem funktsionaalsust millega süsteem saab hakkama peale sõnade lekseemi ühenduste näitamist. See tähendab, et me ei saa nii lihtsalt kustutada olemasoleva andmebaasi ja kirjutada ümber koodi. Kõige parem variant on kasutada graafi andmebaasi lekseemide ja seotse jaoks ja PostgreSQL teiste andmete jaoks nagu kasutajate ja sõnastikute haldamine.

References

[1] Shalini Batra, Charu Tyagi. Comparative analysis of relational and graph databases. International Journal of Soft Computing and Engineering (IJSCE), May 2012.

[2] Quantitative data definition. [WWW] http://www.businessdictionary.com/definition/quantitative-data.html (07.03.2015)

[3] Lexeme definition. [WWW] http://www.merriam-webster.com/dictionary/lexeme (07.03.2015)

[4] Data-Driven Documents. [WWW] http://d3js.org/ (07.03.2015)

[5] Spring. [WWW] http://docs.spring.io/ (07.03.2015)

[6] Qlaara's blog. [WWW] http://blog.qlaara.com/ (07.03.2015)

[7] Neo4j. [WWW] http://neo4j.com/ (07.03.2015)

[8] Oracle Database. [WWW] https://en.wikipedia.org/wiki/Oracle_Database (29.12.2015)

[9] Level of support definition. [WWW] http://searchcrm.techtarget.com/definition/level-of-support (22.02.2016)

[10] Release Notes: Neo4j 2.2.0 [WWW] http://neo4j.com/release-notes/neo4j-2-2-0/ (1.03.2016)

[11] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, Dawn Wilkins. A Comparison of a Graph Database and Relational Database, 2010.

[12] Apache Lucene. [WWW] https://lucene.apache.org/ (7.03.2016)

[13] Justin J. Miller. Graph Database Applications and Concepts with Neo4j, March 2013

[14] YourKit Java Profiler. [WWW] https://www.yourkit.com/ (21.04.2016)

[15] How Graph Databases Relate To Other NoSQL Data Models. [WWW] http://neo4j.com/developer/graph-db-vs-nosql/ (2.05.2016)

[16] NoSQL Databases: An Overview. [WWW]
https://www.thoughtworks.com/insights/blog/nosql-databases-overview (2.05.2016)

[17] Jeevan Joishi, Ashish Sureka. Performance Comparison and Programming Process Mining Algorithms in Graph-Oriented and Relational Database Query Languages, 2015

[18] Mario Miller, Damir Medak, Dražen Odobašic. The shortest path algorithm performance comparison in graph and relational database on a transportation network, 2013