Storing actionable metrics in a graph database

Sonja Malmström

Degree Thesis

Information Technology

2020

Sonja Malmström

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	8063
Författare:	Sonja Malmström
Arbetets namn:	Storing actionable metrics in a graph database
Handledare (Arcada):	Jonny Karlsson
Uppdragsgivare:	Limber AS

Sammandrag:
Detta examensarbete gjordes i samarbete med det norska företaget Limber AS, som har utvecklat den molnbaserade webbapplikationen Limber Projects. Denna webbapplikation kan användas av företag som arbetar med komplexa och leverantörbaserade projekt. Limber intresserade sig av att i framtiden ha möjlighet att kunna undersöka användarbeteende i sin webbapplikation genom att lagra relevant användardata i en grafdatabas. Grafdatabaser är en typ av NoSQL-databas som huvudsakligen fokuserar på samband mellan dataobjekt och lagrar dessa i from av noder och bågar. Målet med detta examensarbete var att tillhandahålla en översikt över hur användardata kan lagras i en grafdatabas samt hur datamodelleringsprocessen kan utföras för detta syfte. Under datamodelleringsprocessen avgjordes vilken data som bildar noder och bågar för att på bästa sätt kunna framställa information om hur användaren rör sig under en session i webbapplikationen. Den praktiska delen av examensarbetet gick ut på att skapa en prototyp som beskriver en fungerande grafdatabas i syfte att lagra specifika användardata. Denna prototyp utvecklades i ArangoDB och testdata skapat av skribenten användes för att evaluera datamodellens uppsättning. Med hjälp av denna prototyp fick man en uppfattning om vilken data som kan anses tillräckligt viktig för att skapa egna dataobjekt samtidigt som man hade möjlighet att undersöka prototypens prestationsförmåga med varierande mängd dataobjekt.

Nyckelord:	Grafdatabas, Datamodellering, Limber AS, användarbeteende, ArangoDB, NoSQL, webbmätvärden
Sidantal:	59
Språk:	Engelska
Datum för godkännande:	17.12.2020

DEGREE THESIS	
Arcada	
Degree Programme:	Information technology
Identification number:	8063
Author:	Sonja Malmström
Title:	Storing actionable metrics in a graph database
Supervisor (Arcada):	Jonny Karlsson
Commissioned by:	Limber AS

Abstract:

This thesis was done in collaboration with the Norwegian company Limber AS, developer of the cloud-based web application Limber Projects. This web application can be used by companies who work with complex and supplier dependent projects. Limber was interested in future possibilities to study and evaluate user behaviour in their web application by storing relevant user data in a graph database. A graph database is a type of non-relational database that mainly focuses on storing information about relationships between data objects as nodes and edges. The goal with this thesis was to provide an overview of how user behaviour data can be stored in a graph database and how the data modeling process can be done. During the data modeling process of this thesis it was decided what data would be stored as nodes and edges in order to provide information of how users move in the web application during a session. The practical part of this thesis consisted of creating a prototype of a working graph database with the purpose of storing user behaviour data. This prototype was developed in ArangoDB and test data created by the author was used to evaluate the setup of the data model as well as test the performance of the database. This prototype made possible an understanding of what data is important enough to become its own data object.

Keywords:	Graph database, Data modeling, actionable metrics, Limber AS, ArangoDB, user behaviour, NoSQL
Number of pages:	59
Language:	English
Date of acceptance:	17.12.2020

CONTENTS

Figures

Tables

1 INTRODUCTION

This study was done in collaboration with the Norwegian company Limber AS. The company has developed the product Limber Projects, which is a cloud-based platform/web application that can be used in complex and supplier dependent projects. Limber Projects is currently mainly focused on the oil and gas industry. Since the company is quite new (founded in 2017), it has become more relevant to analyse user behaviour by collecting data to improve further development of Limber Projects.

Limber is interested in improving its understanding of how Limber Projects is used by its users and to find out whether the product is used as designed or if there are any irregular patterns. This could be done by validating collected data with pre-defined patterns. This thesis worked as a foundation for an upcoming system where user data is collected either from the client side or the server side and afterwards stored in a graph database. To be able to do the data modeling and plan the graph database, test data was used. This test data was created by the author to represent movements between views in the user interface and button clicks.

A graph database is a NoSQL database which is built to manage relationships between data nodes containing information and used to represent complex networks. A graph database is built of "node-edge-node" triples, which means two nodes are connected with an edge, that represents the relationship between these two nodes. (Hurlburt et al. 2017)

Since graph databases focus and provide values from relational data, they are useful in finding possible and new connections and relations between existing data which has not been visible before. This was the reason why Limber wanted to use a graph database to store user behavior data. The interest lies in finding out whether a user has used Limber Projects as designed, if there are any strange and irregular behavior or if there are features that are seldom or never used.

1.1 Purpose and goal

The purpose with this thesis was to provide an overview of how data can be stored in a graph database and to describe how the data modeling process can be done when the purpose is to store specific user behavior data in a graph database. The goal of the practical work in this thesis was to create a model and prototype for how Limber should collect useful user data, what kind of data is interesting and finally how it should be stored in a graph database. The key is to not only store data just because data should be collected and stored, but to collect the right data based on predefined patterns.

1.2 Limitations

This thesis focused on graph databases and how data is stored in them. As implied earlier, one of the main focuses was to show how and what kind to data should be collected to be able to store it in graph database. However, no real user data was collected during this study since the goal with this thesis was to create a prototype. The data that was used in the data modeling process was created by the author to represent real data.

1.3 Methods

In order to reach the goals for this thesis I decided to build a model and prototype of how specific user behavior data can be stored in a graph database. It started with the data modeling process and afterwards moved on to test this data model in a graph database. The prototype was built in ArangoDB, which is a NoSQL database. To use ArangoDB was a decision Limber and I made together. In the practical part of this thesis ArangoDB's Community Edition version was used as the test environment and the test data was stored locally on a computer.

To be able to test the prototype, the database had to contain some test data. In this thesis, all test data was created by the author to represent actual user data for the purpose of testing the setup, run AQL queries and test the performance of the database.

1.4 Thesis structure

This thesis was structured as follows. Chapter 2 provides information about different kind of database models and how they differ from each other. This chapter mainly focuses on relational and non-relational databases. Of all different non-relational database models, this chapters emphasizes graph databases. In this chapter there will also be a review of the graph databases provider ArangoDB, since it was used for the practical part of this thesis.

Chapter 3 is about data modeling and how the data modeling process is usually performed with different kinds of database models. Although, this chapter focuses on how data modeling can be done in a NoSQL database, especially a graph database since one of the main focuses in this thesis was graph databases.

Chapter 4 contains information about two different kinds of web metrics, actionable- and vanity metrics. In this chapter there is a review of these two web metrics and how they differ from each other.

The practical part of this thesis is described in Chapter 5 and 6. Chapter 5 presents a walkthrough of how the data modeling process for the database was done and what kind of data was considered useful. Chapter 6 provides a description of the implementation process and how the prototype was created in ArangoDB's Community Edition version, and how it was tested and what the results were.

Finally, chapter 7 is the conclusion of the entire thesis with a summary of the results and a discussion of the benefits and challenges of using a graph database for storing actionable metrics focusing on user behaviour data and what kinds of possible future studies there might be based on this thesis.

2 DATABASE MODELS

In today's world most of us will probably encounter activities during our everyday lives that require some kind of interaction with a database. It is difficult to not see databases as an important part of computer usage when living in a modern society. Examples of interactions with databases in our everyday life is making an online purchase or making a flight reservation. (Elmasri & Navathe 2016:33-35)

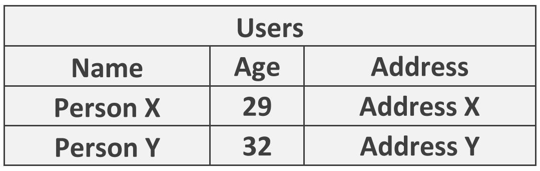
Then what is a database? It can be described as a collection that contains data related to each other. The word data is used to describe facts, and these facts can be stored, and they have some kind of important meaning to why they should be stored. Typically, a database stores coherent collections of data. Randomly collected data cannot be mentioned or identified as a database. (Elmasri & Navathe 2016:33-35)

In other words, a database can be described as a collection of data that is well organized. It is an electronical system that easily gives access to the collected data and is a great way for organisations to store and manage information. (Vázques 2019)

There are two common types of databases: Relational databases and non-relational databases, also known as NoSQL databases. In this chapter, there will be a closer look on relational databases and NoSQL databases, with placing focus on graph databases. (What is a database in under 4 minutes 2019)

2.1 Relational databases

The first person to introduce a relational data model was Ted Codd of IBM research in 1970. His paper immediately raised attention. This paper describes a database as a collection of relations. The model consists of tables of values containing columns and rows. Each row represents an entry, and the columns represent different kind of data that are related to each other. The names of the tables and columns are used to present the values so that they are easier to understand. In figure 1 you can see a visualisation of a table in a relational database. (Elmasri & Navathe 2016:179-180)



Users		
Name	Age	Address
Person X	29	Address X
Person Y	32	Address Y

Figure 1 Visualization of a table with rows and columns in a relational database.

In a table we find a set of columns and rows, and when you look inside a specific row and column, you find an entry. Relationships can be found between tables, but these relationships cannot be considered as the most important feature when it comes to relational databases. In these kinds of databases, we are more interested in the data which they contain. (Vázques 2019)

Unlike NoSQL databases, relational databases have relational schemas. The schema delineates the structure and relationships of a relational database i.e., how the data is arranged. It can for example be graphically illustrated or written in SQL programming language, which is described later on in this chapter. A schema usually describes whether or not a column in a table requires unique values or if there is a specific column that should work as the primary key of the table. With a schema it can also be specified if there are columns in tables that have references to other tables' data. (Melendez 2019)

Since the 1980s, the relational model has been the most significant and used model to store and retrieve data. Because relational databases heavily depend on their structure of schemas and tables, making modifications and queries out of context from the original schema design is very expensive and time consuming. Because of this, relational databases have nowadays become less important. (Benymol & Sajimon 2020)

Relational databases use SQL, which is a shortage of "Structured Quary Language". SQL is a programming language and can be used to manipulate the data stored in a relational database. In the beginning of the 1970s, IBM was first to develop the SQL language. The syntaxes available in SQL are comparable to the English language and are easy to read and write. (Bush 2020)

2.2 NoSQL databases

Hand in hand with the rise of big data and its irregular features, the need of more efficient databases has increased. These kinds of databases are called NoSQL databases and they have grown very popular because of their flexibility and robustness. NoSQL databases have dug into bigger industries as a supplement for relational databases. (Abdullahi et at. 2018)

NoSQL was launched in 1998 and its original meaning was "No SQL", which means that the query mechanism that is used is more similar to the source environment that the developers use. (Fowler 2015)

There are variations of NoSQL types. Some of the most common types are key-value store, document store and graph store. A key-value store uses, as the name indicates, keys to access data that is stored in the database. Key-value stores are for example used as image stores or as filesystems that are key-based. A document store is commonly used when the purpose is to store hierarchical data structures. This type can be used with high-variability data and document search. Finally, a graph store is mostly used for relationship-heavy data. More information about graph databases is described in the next chapter 2.3. (Kelly & McCreary 2013)

There are four features that usually apply to most of the NoSQL databases:

1. **Schema agnostics**. In a NoSQL database there is no requirement of using a schema, which is used in a relational database. By not using schemas, storing information is more flexible and you are able to retrieve data without having the knowledge of how the data is stored. (Fowler 2015)
2. **Nonrelational**. In relational databases the relations exist between tables of data. In a NoSQL database the data is stored as an aggregate, which in practice can be seen as a single entry where all the information is stored, without any tables. (Fowler 2015)
3. **Commodity hardware**. When using a NoSQL database, one does not have to use specialized storage and processing hardware like servers. (Fowler 2015)

4. **Highly distributable**. A database that is distributed has the ability to process and store information on several devices. When looking at a NoSQL database, one can use multiple servers to hold one large database. (Fowler 2015)

2.3 Graph database

Over the last years, the term network has become a common word for most people. It has been used to refer to electronic transmission systems, but nowadays we find the word network everywhere. There are social networks, supply chains and food chains to name a few. (Hurlburt et al. 2017)

Graph databases are built to manage information dense relationships to coexist in dynamic environments. These relationships are similar to networks and graph database can be used to represent and analyse complex networks. Graphs are described as “node-edge-node triples”, which means that two nodes are connected to each other with the help of a relationship (also known as an edge), see a visualization in figure 2 below. After all, this is a very straightforward concept of a graph. The nodes in a graph database can contain one or many properties. The edges that create the relationships can also be given properties. (Hurlburt et al. 2017)

When comparing a graph database to a relational database you realize that a graph database is a collection of nodes and edges instead of tables with rows and columns. Every node that exists in a graph database is unique and identified by a unique identifier. This identifier indicates key value pairs. An edge is also identified by a unique identifier which has details and properties of the starting and ending node. (Vázques 2019)

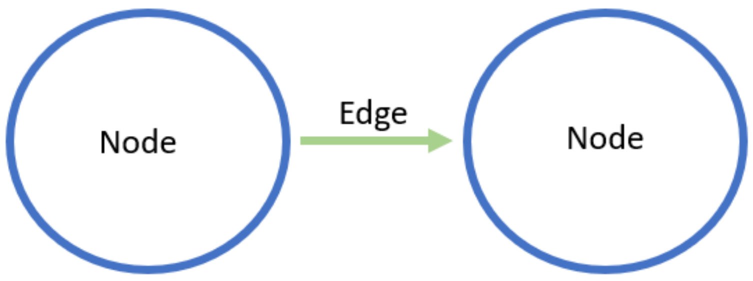


Figure 2 Visualization of relationship between two nodes

A graph database is able to store the same kind of data as in a relational database, but this kind of database is also able to store and map the relationships between different kind of data. Graph databases are useful when you have data that is highly related. When you are able to see relationships between data, it is easier to understand what the data contains and find useful insights. (Vázques 2019)

Graph databases are a great solution if you want to store and navigate datasets where the relationships between the data are equally important as the data. Lately, graph databases have been very popular solutions in data management. (What is a graph database 2019)

With graph databases it is easier to make content much more connected and smarter. For example, it is easier to create consistency of stories and knowledge and create meaning for structures that are interconnected. The output can be, for example, the ability to design more personalized experiences and enable seamless search and find interactions with the system. (What is a graph database 2019)

In a graph database one can store graphs, and a graph can be criss-crossed along some particular edges or across the entire graph. Criss-crossing relationships can be done in a very fast pace because the relationships are not calculated according to query times. When it comes to recommendation engines and social networks, a graph database is a good choice. (What is a graph database 2020)

Poorly designed relationships provide vague graph environments. Nodes', edges' (relationships) and properties' declarations should be clear and should match up to a generalized pattern existing in a specific graph model. A graph can easily grow in size and result in a dilemma regarding magnitude when designing a data model. (Hurlburt et al. 2017)

Graph databases have grown to be popular variants of a NoSQL databases and they have become to be effective tools in visualizing network related relationships. They also offer ability to comprehend growing network behaviours qualitatively and quantitatively. (Hurlburt et al. 2017)

Many big technology companies are using graph databases and graph technology to create, for example, recommendations of content to its users. Although, it is relevant to remember that graph databases do not fit everyone, and other database models will not entirely be replaced by graph databases. (What is a graph database 2019)

2.4 ArangoDB

In this chapter there will be a closer look at the graph database ArangoDB. It was decided for the practical part of this thesis that the prototype of a working graph database would be designed in ArangoDB. For Limber, it was important that the selected database would offer other database models than simply a graph database, which makes ArangoDB a great fit since it is a multi-model database.

ArangoDB can be described as a multi-model database, which is not very common with other NoSQL databases. What makes it a multi-model database is that it is equipped with graph database, key-value store, and document database. ArangoDB users are able to use and combine data models supported by ArangoDB with a single query. Because of this, ArangoDB can be called native. When querying data in ArangoDB, the users use ArangoDB's own query language: AQL (Advantages of the native multi-model database - ArangoDB 2020)

2.4.1 Data model

When storing documents in ArangoDB, one document may contain several attributes or zero attributes. These attributes contain their own values, and they can be a number, string, boolean or null. In this case the value is of an atomic type. The other option is that a value is of a compound type, which means an array or an embedded document. (Concepts 2020)

Once the documents are defined, one can add these to collections. When comparing collections to the setup in a relational database, a collection is the table, and the documents are the rows. What makes them different is that in a relational database defining the columns in a table prior to use is required. In ArangoDB defining in advance what attributes a document and what it may contain is not necessary, which means that ArangoDB is schema-less. What this means in practice is that each document's structure may differ from the other but can still be stored together in the same collection and therefore is not limited to a specific data structure. (Concepts 2020)

In ArangoDB there are two different kinds of collections. One is called "document collection", also known as vertex collections, when talking about graphs. The other kind of collection is a "edge collection". Both collections can store documents, but edge collections have two special attributes: *_from* and *_to.* These attributes are used when creating relationships between documents. For example, two or more documents are stored in a document collection. These documents are linked to each other by another document, but this document (or edge) is stored in an edge collection. This is an example of a graph data model in ArangoDB. (Concepts 2020)

2.4.2 Graph database

As mentioned in the previous section, ArangoDB is a multi-model database and provides a graph database. By looking at the graph capabilities provided by ArangoDB in its graph database, they are very close to a property graph database. (Advantages of Native Multi-Model 2020)

Every document has its own unique identifier, or *_id* attribute. This attribute is automatically stored. When it is time to create a relationship, or an edge as it is called, between two documents, these *_id* attributes are stored in an edge document with attributes *_from* and *_to*. These creates a connection between two vertices. Edges will be stored in edge collections. See figure 3 for visualization. (Basics and terminology 2020)

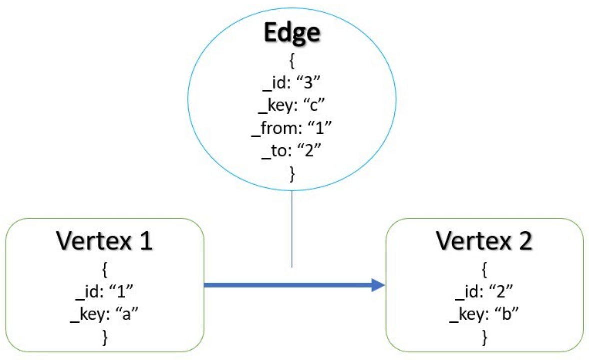


Figure 3 Example of relationship between two vertices in ArangoDB

A revision will be created for every document in ArangoDB. The revision is stored in the attribute *_rev* and is always unique in all collections and documents. Usually, this attribute is used together with queries and works as a pre-condition mainly to avoid loss of data due to situations where a document update was not executed correctly (Document revision, ArangoDB 2020).

There is a wide range of graph database features, for example, shortest_path, pattern matching and graph traversals. If graph visualization is desired it can easily be achieved within the ArangoDB WebUI, where the graph can also be manipulated. (Graphs in AQL 2020)

2.4.3 AQL query language

ArangoDB has its own query language AQL, which is a shortage of "ArangoDB Query Language". This query language was designed to be a query language that is readable by humans, be able to support complex query patterns, accomplish client independency and finally support all data models that ArangoDB provides without having to use several query languages. (Best practices for AQL graph queries 2020)

Looking at AQL one will find two types of queries: Data access and data modification. Data access means that one is able to read documents with the help of a query. Data modification is when one runs queries to modify data, for example update or create a document. (AQL introduction, 2020)

How does a query work in ArangoDB? Usually when a query is executed, the workflow follows a couple of steps. The query is sent from a client application (terminal, webUI) to the ArangoDB server. Everything the user wants to retrieve from the server should be included in the query. If the query contains errors, it will not be executed, and the user will receive an error. If the query looks correct, ArangoDB will parse through the query that was sent from the client application, execute the query, and finally compile. Once the query has been successfully executed, the user will see the results. (AQL introduction, 2020)

When accessing data from the server, the operation RETURN should always be used. The values that are returned as a result are always an array of values. Because there usually are several documents in a collection, a FOR loop is usually used along with the RETURN operation (see figure 4 below). (Data queries 2020)

```
FOR user IN users
   RETURN user
```

Figure 4 Example of query to return documents in a collection

To be able to modify documents, some specific data-modification operations are required. These operations are INSERT, REMOVE, UPDATE, REPLACE and UPSERT. INSERT is used when one wants to add a new document to an existing collection, as visualized in figure 5 below. If a document should be deleted, one should use the operation REMOVE. With the help of the operation UPDATE, parts of an existing document may be updated. REPLACE will replace all attributes in a document when UPDATE only updates specific attributes. Finally, the operation UPSERT is used when documents need to be updated or added with the help of conditions. (Data queries 2020)

```
INSERT {

        name: "John Doe",
        age: "29"

  } IN users
```

Figure 5 Example of an INSERT query, where a document is added to a collection

2.4.4 Indexing

Indexes are used to gain access to documents faster. ArangoDB provides some indexes automatically, but database users are also allowed to create their own indexes. All indexes are created on a collection level. Document attributes *_id, _key, _to* and *_from* have an index automatically. The *_id* attribute is not usable in indexes that are defined by database users, but all the previous mentioned attributes can be used. (Index basics 2020)

Primary index is the persistent index for the *_key* attribute. This primary index allows quick selection of documents when using either *_key* or *_id.* The primary index cannot be modified by a user. The edge index is automatically created in every edge collection. This edge index provides fast access to documents stored in an edge collection by using either the *_from* or the *_to* attribute. Database users do not have the possibility to create their own edge indexes. (Index basics 2020)

When it comes to queries and indexes, ArangoDB usually uses only one index per collection when a query is executed. Although, several indexes can be used if there are multiple FILTER conditions combined with an OR condition in an AQL query. If a FILTER condition is combined with and AND condition, only one index will be used.

The query optimizer has more options to use when picking an index if there are in a collection multiple indexes on different attributes. (Index utilization 2020)

It is often useful to have an index added to several attributes, because the index has now more choices and might become more selective when a query is executed. Every index has a selectivity estimate. Depending on how selectivity an index is, it will filter more documents. The query optimizer usually chooses an index based on its selectivity estimate when the query execution plan is created. Usually are the indexes with the highest estimated selectivity chosen. (Index utilization 2020)

As mentioned earlier, users are able to create their own indexes if needed. ArangoDB provides users with some index types and they differ from each other and are supposed to be used in different scenarios. (Which index to use 2020).

1. **Persistent index** – this index does not affect the loading time of collections since it is persisted on disk and is not needed to be built again in memory if there is a restart of the server or a reload of the index collection.
2. **TTL index** – this index automatically removes expired documents from a collection. The TTL index consists of an *expireAfter* value and documents expire seconds after their *expireAfter* value is met. This value can be a numeric timestamp or in date string format.
3. **Fulltext index** – if what is needed is to index all words in a specific attribute of all documents in a collection, fulltext index can be used. The words need a minimum length to be indexed. This index is used with complete match queries and prefix queries. Fulltext index will only be called by special functions.
4. **Geo index** – this index allows the users to search for a document that are close to a specific area or coordinate. Like the fulltext index this index will be called with the help of special functions or AQL optimization.

3 DATA MODELING

Today, gathering data from different sources and storing it securely is a major challenge for organizations. With today’s technology, data can be gathered from almost anywhere and the challenge is to know what data to get. (Kuldeep & Singh 2018)

When designing and developing a database, one of the most important steps is data modeling. It is especially important in the logical design process. Data modeling also offers support when it comes to requirement analysis, because it is a helpful tool in creating a structure to the process. In terms of coding and maintenance, data modeling gives documentation. (Atzeni et at. 2020)

A data model is built up of entities. These entities are the objects that are interesting to track data of. Every entity has attributes. For example, in a relational database these entities are the tables, and the attributes are the columns and rows inside a table. These attributes describe details vital to track within every entity. As an example, a person can be described as an entity, and the person’s name is an attribute. Entities are connected to each other and these connections are described as relationships. (Franklin 2018)

Conceptual and logical data modeling were developed with interest in the relational systems. After a while, the demand to develop other modeling features begun to increase. The result was more complex and flexible models. In this chapter, I will only focus on how data modeling is used in a NoSQL database model and in a graph database. (Atzeni et at. 2020)

3.1 NoSQL data model

Representing a dataset in a NoSQL database often means organizing the data as aggregates. An aggregate is considered a collection of similar objects, for example users or players in a game. These collections or groups represent a unit of data access. Aggregates can be seen as complex value objects. NoSQL data models can be different depending on the database system. Because of the wide variations of NoSQL database systems and data models, the data is usually organized differently in every database system. (Atzeni et at. 2020)

3.2 Modeling a graph database

To be able to do graph visualizations, the first step to take is to do the data modeling. In a graph data modelling process, one has to go through all entities of a dataset and then decide which entities should become nodes. What you will get from this is a map that will help you visualize a model for the charts. (Disney 2020)

In a graph database you have nodes, labels, relationships, and properties, and these are used when modeling a property graph. A node is a representation of an entity, for example a person or a location. A label represents the role of the node. There can be many labels for each node. Relationships are the connections between two nodes. Finally, properties are key-value pairs with information of either the node or the relationship between two nodes. (Lal 2015)

In a relational database, the data is stored in tables containing columns and rows. When changing this kind of relational data to fit graph data format can be a time-consuming task. An important thing to consider when deciding what data should be nodes or edges is what questions they should answer. Which questions are important enough to become nodes, or should they just be a property of a node? (Disney 2020)

The relationship between nodes can be described as edges, and these links can be single, directed, multiple or self-linking. A **single link** represents a flexible relationship, and the most important thing is that the link exists. When there is a direct flow of information between two nodes, we are talking about a **Direct link**. If a node has one or several links from and back to itself, it is called a **Self-Link**. When there are several relationships between nodes and are visualized separately, it is called **Multiple**. Finally, when talking about properties, they represent characteristics of nodes and edges, but are not nodes themselves because they lack importance. (Disney 2020)

The columns that exist in a relational database table can be represented in a graph database as a property in a node. It is important to include properties that give your graph value and try to avoid unimportant data. Understanding what your users wants to achieve is a very important step when designing a model for a graph database. Without this, the visualization is useless. (Disney 2020)

3.2.1 Creating relationships

One way to visualize a relationship between nodes is to transform the possible relationship to a sentence. In this sentence one can see the nouns/objects as nodes and the verbs as relationships/edges. So, for example, let us use the sentence "*Person clicks on button*". In this example, person and button are nouns and click is the verb, so in a graph database, person and button would be nodes, and clicks would be an edge. (Allen 2019)

There can be several different kinds of relationships. What is interesting is knowing which types should be used, i.e., which are the most preferable relationships. Every relationship has a starting point (where it is coming from), and a range, which represents where the relationship is going. A relationship can be seen as a function. (Allen 2019)

3.2.2 Relationship types

When deciding which relationship type to use, visualizing of what type of relationship this would be in the real world can be very helpful for both oneself and the model users since they have a better understanding on what they are allowed do with the data. The following relation types are presented in Allen (2019)

1. "**Has a" – relationship**, which represents a whole relationship or a part of it. A "Has a"-relationship can be, for example, in social media an interest in something, or in other words a person has an interest in something, and it is typical that one person can have many interests.
2. **"Is a" -relationship**, which is an inherited relationship existing between a child and parent. These kinds of relationships are not that common in property graph modeling, because they can be replaced with labels inside a node.

3. **“Functional” -relationships** are more similar to true functions, which means that one node can only have one range node, for example a relationship that covers a home address to a person is a functional relationship, because one usually has only one home address.
4. **“Transitive” -relationships** cover related relationships, for example if there is a relationship between Node A and B, and there is also a relationship between node B and C, then the relationship between A and C is probably also true. This can be used when figuring out who is related to each other.
5. **“Reflexive” -relationships** are more unusual in property graph modeling. What a “Reflexive” relationship means is that a node has a reflexive relationship to itself. For example, A person always knows a person, because everyone knows themselves. The target label is the same as the source label.
6. **“Symmetric” -relationship** means that if the relationship exists one way, it exists automatically the other way too. For example, if a node represents Person A, and Person A knows person X, one can assume that Person X knows person A as well.

4 WEB METRICS

Web metrics are a subset of broader software metrics with a special importance in marketing development in most organizations. Website activities are usually tracked to better understand and to improve their business. For many organizations, identifying and observing usability problems are included in the web metrics goals. Usually, these metrics are quite basic such as page views and bounce rate. By having the right metrics, you are one step further on the right way, but you then need to know how to use this data to effectively get insights and do important changes. Web metrics can be divided into two groups, actionable- and vanity metrics. In this chapter, I will go through two different kind of metrics, actionable- and vanity metrics. (Denley 2013)

4.1 Actionable metrics

"An actionable metric is one that ties specific and repeatable actions to observe results" (Maurya 2010)

An actionable metric might be seen as less interesting than a vanity metric, which will be described in the next subchapter. With an actionable metric, one is able to tie a certain activity to a transformation which would be hard to do without digging deeper into the data. (Reid 2019)

For example, you have a X number of visitors on your website, but the time spent on that site is only a few seconds. The number of visitors and the time spent on a site are examples of vanity metrics, but what is considered interesting is knowing why the visitors only spent a few seconds on your site and what is needed to resolve this issue. This is how vanity metrics and actionable metrics separates from each other. The factors that are provided from an actionable metric assists how goals and objectives can be settled. (Stains 2019)

With actionable metrics one can more easily discover changes that should be made to make significant impacts. Actionable metrics can be found inside vanity metrics, and by discovering these actionable metrics you have the possibility to get more insights in what is currently working and what might need some modifications. (Austin 2018)

4.2 Vanity metrics

When considering useful metrics for marketing, vanity metrics are a good way to go. These metrics do not usually show real growth or movement in an organization and making decisions based only on vanity metrics might not be the best decision, because vanity metrics usually measure the marketing activity, but shows seldom what kind of result the activity had on the organization. (Reid 2019)

Examples of vanity metrics are page views, email subscribers, trial users and likes on social media. If these kinds of vanity metrics are left without a secondary metric, they will remain quite useless if you want some more deeper insights. For example, without knowing for how long a person have actively been using a downloaded app, the number of downloads does not give enough information. (Reid 2019)

Page views, traffic and time on a site do show how well a website or an app is doing and there is no harm in tracking vanity metrics, but it is important to remember that these kinds of metrics do not tell us everything. They mostly control your overall performance and describe users' routines on your webpage or app. (Stains 2019)

By going deeper into a vanity metric, there is much actionable data to be found. This kind of information is, for example, from where the webpage traffic is coming and what is driving this traffic. It is relevant to know which vanity metrics that should be un-locked and dug deeper into (Stains 2019)

4.3 Transform vanity metrics into actionable metrics

Examples of vanity metrics and their comparable actionable metrics and why you should turn them into actionable metrics and what insights they might give are dis-cussed in Austin (2018). In this subchapter, these examples are revisited.

The number of pageviews is, as mentioned earlier a vanity metric. This metric's parallel actionable metric is the source of the site's visitors or bounce rate (how many visitors enter a website and then leave the site without exploring other pages on the site). By finding out the source of site visitors you get insights in what channels get more visitors to a website. If a website has a high bounce rate and a high number of pageviews, this might tell you that there are issues with the layout or the content.

The time spent on a website is another example of a vanity metric, and its comparable actionable metric is time spent on the page and scroll depth. By just looking at the time on a site might give some misleading information but looking at the time on a page gives more information if a visitor is engaging with the content presented. When looking at the information that is given by a scroll depth, it shows if a user really reads and scrolls through the entire page, instead of just opening the page and leaving it open without browsing and scrolling through it, if the time on the page is long.

Finally, another vanity metric turned into an actionable metric is opening of emails and email clicks. An email can be opened just to mark it as read, but behind every click is usually an intention. If a link has many clicks, this might tell you that the content is interesting and appealing. Some typical vanity metrics and their corresponding actionable metrics are shown in table 1 below.

Vanity metric	**Actionable metric**
Pageviews	Source of visitors, bounce rate
Time spent on site	Time spent on a page, how long is the scroll depth
Number of followers on social media	Engagement rate
Opening emails	How many clicks on link in email

Table 1 Examples of vanity metrics vs. actionable metrics

5 PLANNING THE DATABASE

When planning and creating a new database, it is important to make sure that each step on the way has been carefully executed. In this case, it was important to figure out what kind of data should be gathered and based on this data figure out how it should be stored in the database. In this chapter there will be a walkthrough on how the data modeling and planning of the graph database was done.

Because this thesis works as a cornerstone for a larger project that will be developed in the future, it was relevant to design a working prototype that will work as a base. With the help of this prototype, one is able to see how a graph database could look like and perform when the purpose is to store user behavior data. The goal was to create a cornerstone and start point for future development in user pattern recognition and product development.

5.1 Data modeling

One of the first steps of planning a database, especially a graph database, is data modeling. In this subchapter a description is provided of the kind of data that should be stored and how it should be stored. The aim was to plan a graph database model where one can store data that in future would help Limber find relevant insights on how users behave in the Limber Projects web application. Limber was interested in figuring out how a standard user moves around in the application and whether or not a user moves in a regular way or if there is any confusion or irregular patterns.

Designing the model for the graph database started with figuring out what data should be nodes (vertices) and what data should be edges (see chapters 2.3 and 5 for more information about nodes and edges). According to Allen (2019) in chapter 3.2.1, during the modeling process it is possible to use sentences for choosing what data should be nodes and what data should be edges. During the data modeling process of the practical part of this thesis, three different sentences were used. These sentences were chosen because they represent questions whose answers are relevant. The answers are the type of insights that Limber would like to get from storing user data in a graph database.

In the sentences used, a session represents a user logging in to Limber Projects. Because it was important to keep users' identities anonymous at this point of the development process, the decision was taken to use sessions to represent user behavior. These sessions will not be connected to a user. Below are the sentences/questions used in the data modeling process:

1. **How many sessions have clicked on the New Profile- button?**
2. **How many sessions have moved from the Document list- view to the New Profile- button?**
3. **From where has a session/user moved to get to the New Profile- button?**

These are very specific sentences that describe standard actions that can be performed in Limber Projects. They were chosen because they can be applied in many different situations while a user is using Limber projects. By using these questions as a base, they would help the data modeling process by making it easier to visualize what the nodes and edges would look like.

5.1.1 Choosing nodes and edges

Let us start with the first sentence *"How many sessions have clicked the New Profile- button."*. As described earlier in chapter 3.2.1, one can use sentences to decide what data should be nodes and what data should be edges. Nouns or objects should be nodes and verbs should be edges. By looking at the sentence, there is one verb: **have clicked**, and two nouns: **sessions** and **button.** This means that session and button could be nodes and click could be an edge, as visualized in figure 6 below.

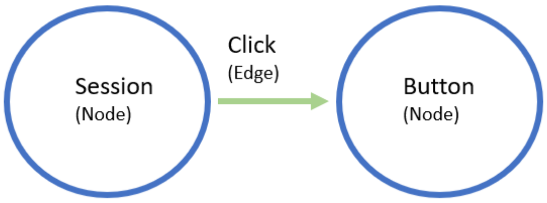


Figure 6 Visualization of sentence "How many sessions have clicked the New Profile- button" as nodes and edges

Now, let us look at the second sentence *"How many sessions have moved from the Document list- view to the New Profile- button."*. By looking at this sentence, there is one verb: **have moved**, and three nouns: **sessions**, **view, and button.** Once again, this could mean that session, view, and button should be nodes and moves should be an edge, as visualized in figure 7 below.

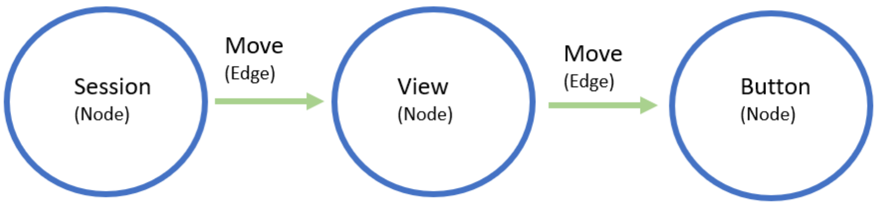


Figure 7 Visualization of sentence " How many sessions have moved from the Document list- view to the New Profile- button " as nodes and edges

Finally, let us look at the third sentence "*From where has a session/user moved to get to the New profile- button?"*. Like the previous two sentences, there are two nouns: **session and button** and one verb: **has moved to**, which means that session and button could be nodes and move could be an edge. This is visualized in figure 8 below.

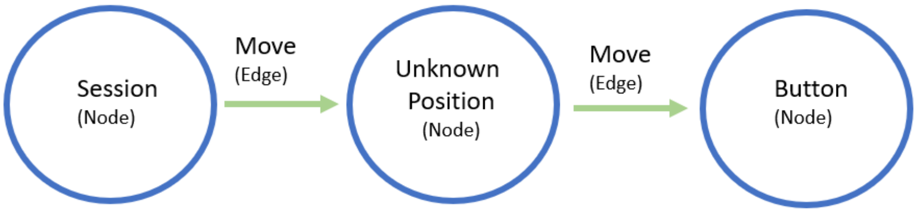


Figure 8 Visualization of sentence "From where has a session/user moved to get to the New Profile- button?" as nodes and edges

Since it is important to find out what has happened during a session, it might not be advisable to have sessions stored as nodes, as tracking how a user has moved during one session would be more confusing this way. What is interesting is knowing where a user has moved from to get to its destination. To get this kind of information, the data object needs properties that provide information on how the session (user) has moved: from where it started and what the destination was.

So, let us look into the possibility of a session being stored as an edge rather than a node. If a session object was stored as an edge, it would contain the important attributes *_to* and *_from*, which are relevant when it comes to finding relationships. In this case it is intriguing to discover the actions taken as the user has moved around in Limber Projects and clicked on different buttons or areas and moved from one view to another.

This edge could have all the same attributes as it would as a node, but what would keep every session in Limber Projects unique is that they would have their own unique session ID. The *_key* attribute that is necessary for every document in ArangoDB is always unique and is either autogenerated or manually created in ArangoDB. Even though it is unique, it would not be optimal to be used as the Session ID, since there are most likely many actions during a session in Limber Projects, and these actions will be stored as a separate edge object. This is why a separate session ID attribute is needed to make it possible to get all the edges that have been created during the session. See figure 9 below for a visualization of this set up.

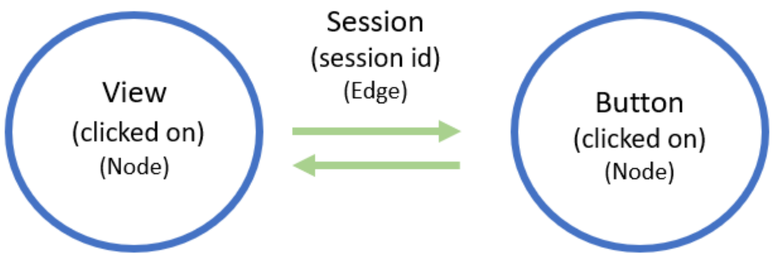


Figure 9 Visualization of the same sentence as given in figure 8, but using "session" as an edge

Since views and buttons are objects that will have relations between each other, it is recommended to store them as nodes, as described earlier in this chapter. Unlike the edge objects where the *_key* attribute is automatically generated, I decided with the node objects to create the *_key* attribute manually in the test environment. Because the nodes' *_key* attributes are used as the values in the edge objects' *_to* and *_from* attributes, they are more readable when running queries if the *_key* attribute is manually in-serted.

5.1.2 Setup for nodes and edges

In the previous subchapter 5.1.1 a decision was taken to store sessions as edges and views and buttons as nodes. To better understand these values, it is important to specify how these values are defined in Limber Projects:

- **Session:** A session represents in this case when a user logs in to the system and moves around by clicking on different areas in the web application.
- **Button:** A button in Limber Projects is a typical clickable button that either opens a small popup view or a completely new view.
- **View:** A view represents in Limber Projects the viewing area that a user looks at and is working in. It can also be a smaller pop-up view that is shown when a particular button is pressed

Because privacy online is important, it is not in this case relevant to track data from a user. Instead, every session will have a unique session ID attribute. This attribute will not be connected to the user itself and it will not be possible to track a session back to a user. In this way it is possible to keep the users' identities anonymous and the sessions will not be used to track users.

An edge object is described in chapter 5.1.2 as holding at least these four meta-attributes: *_id*, *_key*, *_from* and *_to*. The *_key* attribute is unique and either autogenerat-ed or manually defined, while the *_id* attribute is a combination of the *_key* attribute and the name of the collection. The *_from* variable contains information about the starting point, i.e., which node the edge started from. This means that the *_to* variable provides information about the end point. Below in figure 10 is an example of what a simple "session" object could look like stored as an edge.

```
{
        "_key": "482412",
        "_id": "sessions/482412",
        "_from": "views/projectList",
        "_to": "button/createProject",
        "_rev": "_bTOq2QO---"
}
```

Figure 10 An example of a session object with test data saved as an edge

A node object always contains two meta-attributes, *_key* and *_id.* As with the edge object, the *_key* object is unique and either autogenerated or manually defined. The *_id* attribute is a combination of the *_key* value and the name of the collection where the node is stored. Additional attributes can be added according to interest and need. See figure 11 below for an example of how a “button” object can be stored as a node with some additional attributes.

```
{
        "_key" :  "createProject",
        "_id" :  "button/createProject",
        "_rev" :  "_bTOq2QO---",
        "name" : "Create Project",
        "type" : "button"
}
```

Figure 11 Example of how a button object can be stored as a node

6 IMPLEMENTATION

In the practical part of this thesis, ArangoDB Community Edition was used for the database prototype. At this point, it was unnecessary to use any other versions of ArangoDB since all data that was used for the data modeling process was test data. As the test data was stored locally, there was no need for a cloud version. All the collections and documents were created manually on one computer.

6.1 Installation

ArangoDB Community Edition was downloaded from ArangoDB's website (Download Community Edition 2020) and installed locally on a computer using Microsoft Windows as operating system. The local server run at http://127.0.0.1:8529/. After installing ArangoDB Community Edition database working with the database becomes possible either through the WebUI or ArangoDB Shell, which is a synchronous shell designed to interact with the server. The WebUI, or the graphical web interface (GUI) launches in your browser.

6.2 Add collections

As mentioned in chapter 5.1.1, there are two possible types of collections in ArangoDB's graph databse. The first one is a document collection, which contains nodes and the second one is an edge collection containing edges. In the previous subchapter, it was stated that sessions would be edges, and views and buttons would be nodes. Therefore, it was necessary to create three collections:

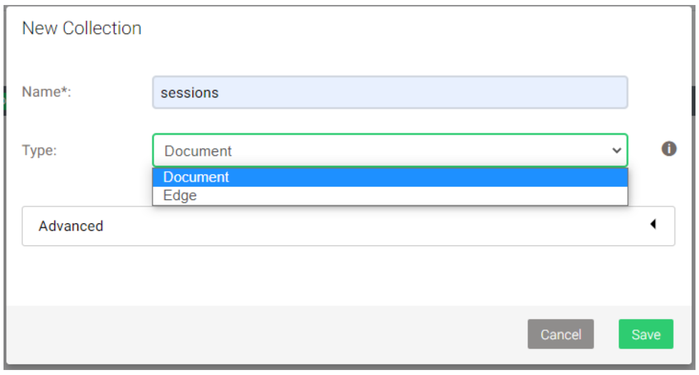
1. A **document collection** where nodes containing information about buttons will be stored
2. A **document collection** where nodes containing information about views will be stored¨
3. An **edge collection** where edges with information about sessions will be stored.

Creating a collection in ArangoDB can be done in two ways. A collection can be created by using Arango Shell, which is a command-line client tool. After selecting the correct database in the Arango Shell a new collection can be created using the command shown in figure 12. This command can be used to create both edge and document collections.

```
db_create(name_of_your_collection)
```

Figure 12 Command to create a new collection with ArangoDB Shell

It is also possible to create new collections in the WebUI. You have to log in to the database where the collection should be stored by launching the GUI in a browser. Once logged in clicking the “Add collection”- button will open the window shown in figure 13 below. The information needed when creating a collection is a name and type of collection.


New Collection

Name*: sessions

Type: Document

Figure 13 A screenshot taken from ArangoDB Community Edition showing how a collection can be created inside the webUI

6.3 Creating nodes and edges

As mentioned earlier, test data was used to design and test this prototype for a working graph database. This test data was created by the author and was manually inserted in the database for modeling purposes. One of the main purposes of inserting test data in the collections mentioned in the previous subchapter was to have the ability to run AQL queries to check whether or not the data was stored in the right way and that the queries return desired results.

To insert data manually in ArangoDB one can once again use either Arango Shell or the WebUI. To insert data with Arango Shell, the first step is to choose which collection the data object should be inserted to. As mentioned earlier, depending on whether one's intention is to create an edge or a node, there are some meta-attributes that must be included when creating a new data document. In a node, these are *_id* and *_key,* and in an edge there are the additional start and end attributes: *_to* and *_from.* Shown in figures 14 and 15 are the commands to be used in Arango Shell when creating new data documents.

```
db.node_collection_name.save(
    {
      _key : ”createProfile”,
      name : ”New Profile”
      type : “button”
    })
```

Figure 14 Example of command in Arango Shell when creating a new node object

```
db.edge_collection_name.save(
    {
      _key : “5673735”,
      _to : ”buttons/createProfile”,
      _from: “views/documentList”
    })
```

Figure 15 Example of command in Arango Shell to create a new edge object

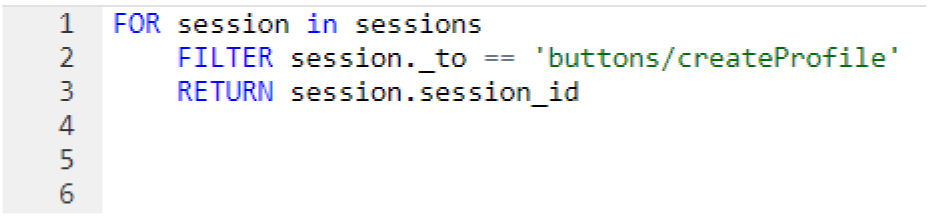
6.4 Running AQL queries to get results

In the previous chapter it was concluded that the database would be built using sessions as edges while using buttons and views as nodes. Without running any queries in the database, the sufficiency of this setup cannot be verified. After all, the data is stored because we want interesting findings, and to get these results we will have to run queries.

One part of the data modeling process was to store some test data in document- and edge collections, then try different queries that will retrieve answers to the three sentences/questions used in the modeling process where nodes and edges were chosen (see chapter 5.1 for more information about the data modeling process).

As described earlier, the database consists of three collections of data documents: one edge collection containing information about sessions and two document collections where one contains information about buttons and the other containing information about views. It is necessary to be familiar with the setup of these collections and the data they contain to be able to run queries that will return valid results.

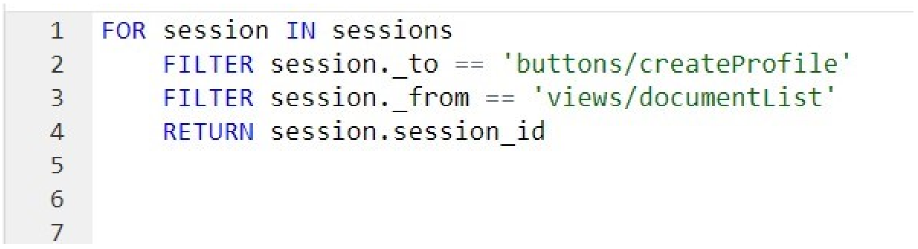
When running a query to retrieve an answer to the question *"How many sessions have clicked on the New Profile- button",* the query should loop through the edge collection that contains session objects. Since it is interesting to find out how many have clicked on the New Profile- button, the query should filter all session objects by their end point, the *_to* attribute. A screenshot of the query is shown in figure 16. The return values are always shown in an array. The return values in the query displayed in figure 16 are only the session ID of the session object. If one wants to get the entire object, the return attribute should be just "session" in this example. The same goes for the two other queries that are shown in figure 17 and 18.



```
FOR session in sessions
    FILTER session._to == 'buttons/createProfile'
    RETURN session.session_id
```

Figure 16 Screenshot of a query that will return all the session IDs that have at some point clicked on the button New Profile. The screenshot is taken from ArangoDB Community Edition

The second sentence/question that was used in the data modeling process was *"How many sessions have moved from the Document list- view to the New Profile- button?"*. The query was used to retrieve data that would give an answer to the question is shown in figure 17 below. Once again, the query loops through all data objects that are stored in the Session collection, and filters theses session objects by their *_to* and *_from* attributes. The query in this case only returns the session id attribute of the session object.



```
FOR session IN sessions
    FILTER session._to == 'buttons/createProfile'
    FILTER session._from == 'views/documentList'
    RETURN session.session_id
```

Figure 17 Screenshot of a query that will return all the session IDs that have at some point moved from the Document list- view to the New Profile- button

Finally, the third sentence used in the data modeling process was *"From where has a session/user moved to get to the New Profile- button?"*. Like in the two previous queries, we once again loop through all objects in the Session collection and filter the session objects by their *_to* attribute. Since we are interested in knowing from where this session has started to get to the New Profile- button, the query returns the *_from* attribute of the session object. In figure 18 below there is a screenshot of the query.

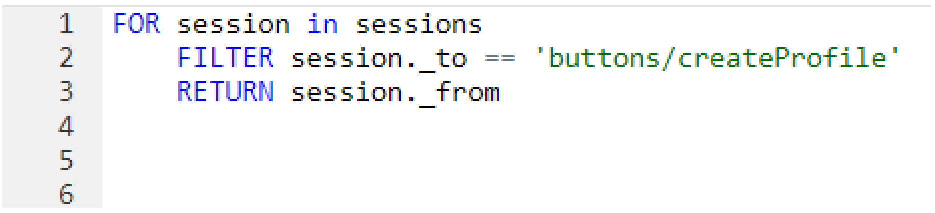

```
FOR session in sessions
    FILTER session._to == 'buttons/createProfile'
    RETURN session._from
```

Figure 18 Screenshot of a query that returns all the edges that have the New Profile- button as end point. The return value in this query is the session object's _from attribute. Screenshot taken from ArangoDB Community edition

6.5 Performance

To test the performance of the graph database, it was relevant to run queries with a large amount of data. The test data was a large JSON dataset (list) with session objects. This test data was created with Python, where every session object was generated with random button- or view objects as the *_to* and *_from* attributes. The *session_id* was also randomly generated by choosing a number between 1-200 (see the example of data objects that were imported in a JSON list in figure 19 below). Finally, the test data was imported to the database by using the Import JSON option in the WebUI in ArangoDB. All objects in the button- and view collections were added manually to the database since they are specific and limited in amount. Once the test data was imported to the database, it was time to test the database's performance by first running four similar queries as in the previous chapter.

```
[
    {
        "_from": "buttons/projectName",
        "_to": "views/documentList",
        "session_id": "100"
    },
    {
        "_from": "views/profile",
        "_to": "button/createProfile",
        "session_id": "110"
    }
]
```

Figure 19 Example of a JSON list that was imported to the session collection

When testing the performance of the graph database, it is important to keep indexes in mind. ArangoDB automatically generates two indexes in every edge collection:

1. **Primary index** - is added to the *_key* attribute
2. **Edge index** - is added to the *_to* and *_from* attributes

The first performance tests were done by using only the primary index and edge index i.e., no user-defined indexes were considered. All four queries shown below were executed with a different amount of session objects in the session edge collection. The amount started with 5000 session objects and ended with 50 000 session objects. The queries that were used to test the execution time are shown below in figures 20, 21, 22 and 23. These queries are later represented as Query 1, 2, 3 and 4.

```
Query 1

FOR session IN sessions

    FILTER session._to == "buttons/createProfile"
    RETURN session
```

Figure 20 Query 1 - Similar query as shown in figure 16, except this query returns the entire session object

```
Query 2

FOR session IN sessions

    FILTER session._to == "buttons/createProfile"
    RETURN session._from
```

Figure 21 Query 2 - Similar query as shown in figure 17

```
Query 3

FOR session IN sessions
    FILTER session._to == "buttons/createProfile"
    FILTER session._from =="views/documentList"
    RETURN session
```

Figure 22 Query 3 - Similar query as shown in query 18, except this query returns the entire session object

```
Query 4

FOR session IN sessions
    FILTER session.session_id == '100'
    RETURN session
```

Figure 23 Query 4 – This is a query that looks for session objects with a specific session ID and returns the entire session object

In figure 24 below there is a visualization of example execution times (in milliseconds) for the four queries described in figures 20-23. These queries were executed without any additional user-defined indexes, which means that the edge index and primary index were active because they are automatically created by ArangoDB.

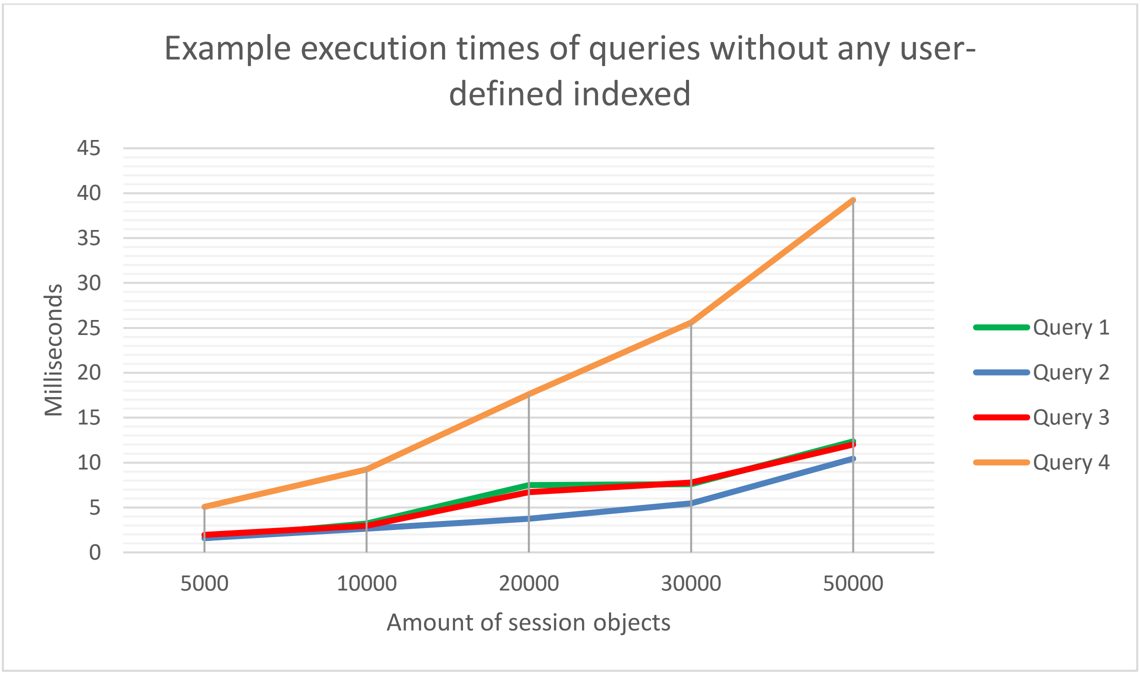


Figure 24 Example of execution times in milliseconds for four different queries without any user-defined indexes added

Queries 1-3 have quite similar execution times, and they increase continuously as the number of sessions objects increase. Query 4 differs a lot form the rest of the queries when it comes to the execution times. All queries execution times increases as the amount of session objects increase, but with query 4 the execution times are significantly higher compared to the other queries.

What is important to keep in mind is that when queries 1, 2 and 3 were executed, the edge index was used to decrease the execution time, since it is automatically applied to the *_to* and *_from* attributes in edge objects by ArangoDB, and in these queries they perform a filtering based on the *_to* or *_from* attributes. Query 4 uses no index when executed, which could explain the much larger execution time.

To speed up the execution time for query 4, a user defined index was added. To the *session_id* attribute in the session objects, a persistent index was added (see chapter 2.4.4 for more details about index types in ArangoDB). Once this index was added, the query was executed again. In figure 25 below the different execution times for query 4 is shown.

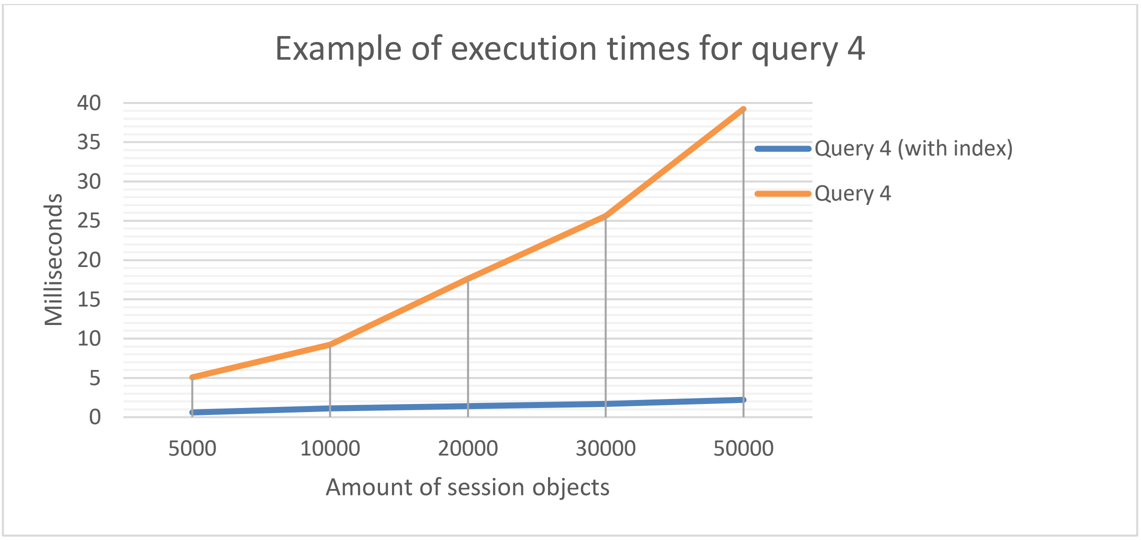


Figure 25 Example execution times in milliseconds for query 4, when the query is executed with or without an additional user defined index

The difference between the execution times is big. When the persistent index was added to the *session_id*, the execution time decreased significantly. With the index, the execution time does not increase much as the amount of session objects increase, as in this case to the maximum amount of 50 000 session objects. Because this index was tested on only 50 000 data objects, it is not possible to conclude that this query (with the index) would perform in the same way with a much larger data amount.

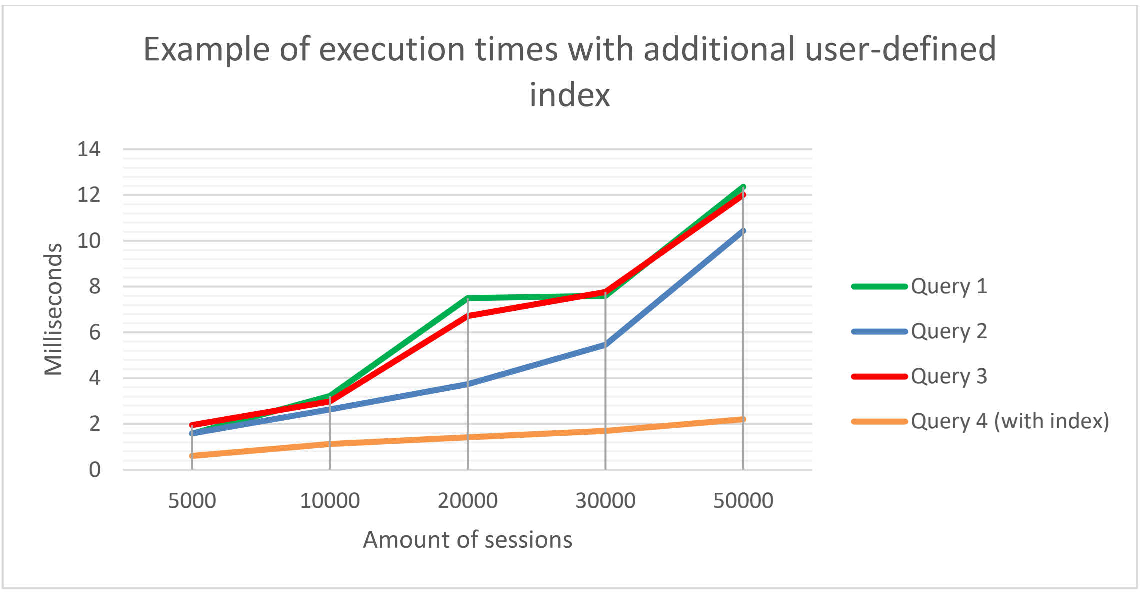


Figure 26 Example execution times for the queries, when query 4 has an additional user-defined index applied

In figure 26 above is an updated version of figure 24. This figure shows examples of what the queries' execution times look like with ArangoDB's automatically generated indexes and the user-defined index applied. As mentioned in chapter 2.4.4, several indexes can be used when a query contains multiple FILTER conditions combined with an OR condition, but in queries 1-4 there is only on FILTER condition used, which means that only one index will be used.

In general, these are good execution times and there is no irregular behavior when the amount of session objects increases, instead the execution time increases constantly as the data amount increases. It is not a straight line since the time does not increase proportionally.

7 CONCLUSION

This thesis was done in collaboration with the Norwegian company Limber AS, who was interested in improving its understanding of how their cloud-based web application Limber Projects is used by their users by collecting user behavior data to improve further development of the web application. Limber was also interested in discovering whether or not Limber Projects is used as designed. This could be done by gathering and storing user behavior data in a graph database and compare it to pre-defined user behavior patterns. This thesis worked as a foundation for a future system.

The purpose of this thesis was to provide an overview of how data can be stored in a graph database and describe how a data modeling process can be done when the aim is to store specific user behavior data in a graph database. The goal for the practical part of this thesis was to design and create a prototype of a working graph database where user behavior data could be stored.

According to Atzeni et al. (2020), the data modeling process works as a helpful tool for creating a structure and provides documentation. This thesis can be seen as a documentation for a data model since the goal for the practical part of this thesis was to create a prototype of a graph database. A data modeling process can be time consuming and requires a clear picture of what the goal is. In this case, the prototype was supposed to represent a graph database model where user data can be stored, which means that this prototype is not based on any existing database and therefore this data modeling process started from scratch.

Without having any existing data model of how user data could be stored in a graph database, the first step was to gather as much information as possible about data modeling before starting the process. According to Disney (2020) one way to visualize a graph database model is to figure out what questions the relationships should answer and also figure out what data is important enough to become their own node and edge objects. In this data model process, it was useful to use this set up, i.e., using questions when deciding what data should be nodes and what data should be edges.

Another important part of the data modeling process in this case was testing the data model by importing test data to ArangoDB and run AQL queries designed to provide answers to the questions used in the data modeling process. Without testing the data model in this prototype with queries, it would have been more challenging knowing if the setup was right.

During the data modeling process, the modeling idea presented by Allen (2019) where one should use sentences for choosing what data should be nodes and what should be edges worked as a base. Allen explains that nouns and objects in the sentences chosen can be seen as nodes and the verbs are the relationships (edges). In this thesis the same principal was used, i.e., the sentences/questions found in chapter 5 were first used with the same goal as Allen presents, nouns are nodes and verbs are edges. By having a closer look at this setup, it became clearer that this setup might not give the desired results. How a user has moved during a session was the main interest in this data modeling process, which ended in the solution that all sessions would be stored as edges, even though they are nouns in the sentences used in the data modeling process. If sessions are stored as edges, they will contain the important attributes that provides information of start and end points, for example from where has a user moved to get to button X. Below in figure 27 is an example of how movements during a session could look like as a grap when sessions are stored as edges and buttons and views are stored as nodes.

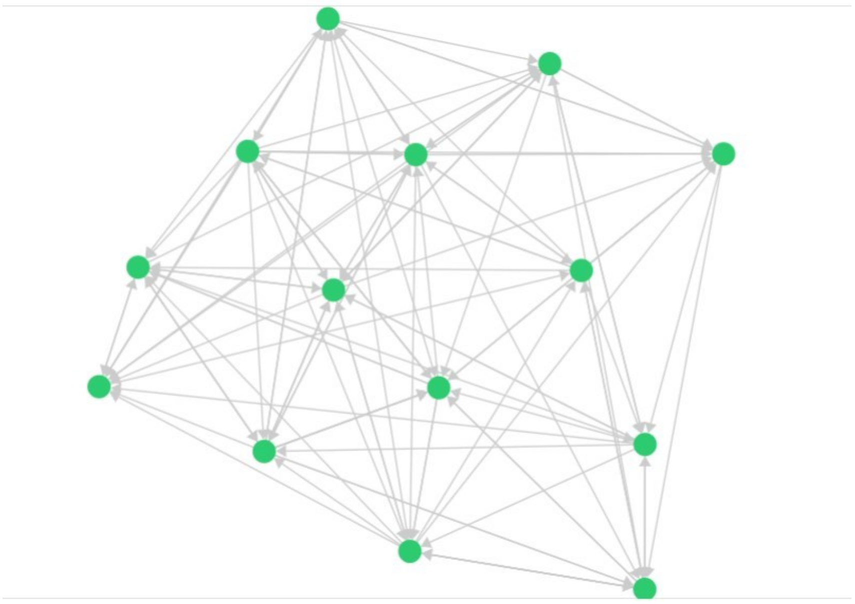


Figure 27 Example of movements during a session visualized as a graph, Graph created in ArangoDB when executing query 4 from chapter 6.5

With the help of the test data that was created for this prototype, it was possible to both evaluate the setup of the node- and edge objects and at the same time evaluate the performance of the database. By running carefully designed queries, it was possible to check if the attributes in the nodes and edges were correct and if the outcome provided the desired information. By running the queries shown in figures 16, 17 and 18 in chapter 6.4, it was possible to establish that the data model was designed correctly with a desired outcome.

The goal with this thesis was to create a prototype for a working graph database model where user behavior data can be stored, and after completing the practical part of this thesis this task has been completed. By doing this thesis Limber AS has now a data model of how user data can be stored and what data should be nodes or edges. The prototype's performance was also tested by running four different kinds of queries (see figures 20-23 in the previous chapter 6.5). The execution time (in milliseconds) for the queries increased constantly as the amount of data increased in the database. The execution time was tested with data amount starting from 5000 objects and ending in 50 000 objects. Once indexes were applied to edge attributes, the execution time was speeded up significantly.

A shortage in this study is that this is only a prototype and it has not been tested live in a real situation with real data, which means there might occur some unforeseen shortcomings in the setup and later need modifications. The performance of the database was good when tested with the test data, but it has only been tested on one computer, which means that the execution times cannot be seen as final. Also, the amount of test data in this thesis was only 50 000 data objects, which means that it is not possible in this case to conclude that the database would perform in the same way if the data amount, for example, would be several hundreds of thousands or millions.

7.1 Future studies

Since the practical part of this thesis was to create a prototype of a working graph database, an interesting future study would be to test this prototype in a real-life production environment. Another possible future study would be to use this graph database model to study user behaviour and develop a system for user pattern recognition.

REFERENCES

Abdullahi, A.I., Basri, S., Ahmad, R., Watada, J. & González-Aparicio, M.T., 2018, Automatic schema suggestion model for NoSQL document-stores databases. *Journal of Big Data*, 5(1), p. 1-17. Available from: ABI/INFORM Global. Accessed: 28.8.2020

Advantages of the native multi-model database – ArangoDB, 2020, ArangoDB. Available from: https://www.arangodb.com/why-arangodb/native-multi-model-database-advantages/ Accessed: 25.8.2020

AQL Introduction, 2020, ArangoDB. Available from: https://www.arangodb.com/docs/stable/aql/ . Accessed: 02.11.2020

Allen, David, 2019, *Graph data modeling: All about relationships.* Available from: https://medium.com/neo4j/graph-data-modeling-all-about-relationships-5060e46820ce Accessed: 26.7.2020

Atzeni, Paolo. Bugiotti, Francesca. Cabibbo, Luca. & Torlone, Riccardo, 2020, Data modeling in the NoSQL world. *Computer Standards & Interfaces,* 67. Available from: ScienceDirect. Accessed: 30.8.2020

Austin, Karissa, 2018, *How to turn vanity metrics into actionable metrics.* Available from: https://www.callrail.com/blog/vanity-metrics-vs-actionable-metrics/ Accessed: 19.8.2020

Basics and terminology, 2020, ArangoDB, Available from: https://www.arangodb.com/docs/stable/data-modeling-documents-document-address.html Accessed: 8.12.2020

Benymol, Jose & Sajimon, Abraham, 2020, Performance analysys of NoSQL and relational databases with MongoDB and MySQL. *Materials today: PROCEEDINGS,* 24(3), p. 2036-2043. Available from: ScienceDirect. Accessed: 30.8.2020

Best Practices for AQL Graph Queries, 2020, ArangoDB.
Available from: https://www.arangodb.com/2020/05/best-practices-for-aql-graph-queries/ Accessed: 25.8.2020

Bush, J., 2020, *Learn SQL Database Programming: Query and manipulate databases from popular relational database servers using SQL*. [ebook], Packt Publishing.
Available at: https://www.perlego.com/book/1484874/learn-sql-database-programming-query-and-manipulate-databases-from-popular-relational-database-servers-using-sql-pdf Accessed: 06.12.2020

Concepts, 2020, ArangoDB. Available from:
https://www.arangodb.com/docs/stable/data-modeling-concepts.html. Accessed: 5.8.2020

Data queries, 2020, ArangoDB. Available from:
https://www.arangodb.com/docs/stable/aql/data-queries.html. Accessed: 02.11.2020

Denley, Norah, 2013, MISSING the MESSAGE in the METRICS., *LIMRA's MarketFacts Quarterly,* (2), p. 42-45. Available from ABI/INFORM Global. Accessed: 28.8.2020

Disney, Andrew, 2020, *The ultimate guide to creating graph data models.*
Available at: https://cambridge-intelligence.com/graph-data-modeling-101/ Accessed: 24.7.2020

Document Revision, 2020, ArangoDB. Available from:
https://www.arangodb.com/docs/stable/appendix-glossary.html#document-revision. Accessed 19.11.2020

Download Community Edition, 2020, ArangoDB, Available from: https://www.arangodb.com/download-major/ Accessed 20.10.2020

Elmasri, R. & Navathe, S.B., 2016, *Fundamentals of Database Systems, Global Edition*. 7th edition, Pearson. Available from: https://www.perlego.com/book/812305/fundamentals-of-database-systems-global-edition-pdf Accessed: 26.10.2020

Franklin, Anna Grace, 2018. *Data modeling explained in 10 minutes or less*. Credera. Available from: https://www.credera.com/insights/data-modeling-explained-in-10-minutes-or-less/ Accessed: 27.8.2020

Fowler, A., 2015, *NoSQL For Dummies*. Wiley. Available from: https://www.perlego.com/book/1003103/nosql-for-dummies Accessed: 12.7.2020

Graphs in AQL, 2020, ArangoDB. Available from: https://www.arangodb.com/docs/3.7/aql/graphs.html. Accessed: 8.12.2020

Hurlburt, G.F., Thiruvathukal, G.K., & Lee, M.R., 2017, The graph database: Jack of all trades or just not SQL?, *IT Professional Magazine,* 19(6), p. 21-25. Available from: ABI/INFORM Global. Accessed: 27.8.2020

Index basics, 2020. ArangoDB. Available from: https://www.arangodb.com/docs/stable/indexing-index-basics.html Accessed: 30.11.2020

Index utilization, 2020, ArangoDB. Available from: https://www.arangodb.com/docs/stable/indexing-index-utilization.html Accessed: 7.12.2020

Kelly, A. & McCreary, D., 2013, *Making Sense of NoSQL*. [ebook]

Manning Publications. Available at: https://www.perlego.com/book/1469465/making-sense-of-nosql-pdf Accessed: 06.12.2020

Kuldeep, L. & Singh, S. P., 2019, Modeling big data enablers for operations and supply chain management., *International Journal of Logistics Management,* 29(2), p. 629-658. Available from: ABI/INFORM Global. Accessed: 28.8.2020

Lal, M., 2015. *Neo4j Graph Data Modeling*. [ebook] Packt Publishing. Available at: https://www.perlego.com/book/4005/neo4j-graph-data-modeling Accessed: 23.7.2020

Maurya, Ash., 2010. *3 rules to actionable metrics in a lean startup.* Available at: https://blog.leanstack.com/3-rules-to-actionable-metrics-in-a-lean-startup-7cf483b0a762 Accessed: 19.8.2020

Melendez, Steven., 2019, *What is relational database schema?.* Available from: https://www.techwalla.com/articles/what-is-relational-database-schema. Accessed: 6.12.2020

Reid, Kyle, 2019, *Vanity metrics and their actionable metric counterparts.* Available from: https://medium.com/@mobileontap/vanity-metrics-and-their-actionable-metric-counterparts-4a877a74ddd5 Accessed: 19.8.2020

Stains, Jonathan, 2019, *Turning vanity metrics into actionable marketing metrics*. Available from: https://www.weidert.com/blog/vanity-metrics-vs-useful-marketing-metrics. Accessed: 19.8.2020

Vázques, Favio., 2019, *Graph databases. What's the big deal?* Available from: https://towardsdatascience.com/graph-databases-whats-the-big-deal-ec310b1bc0ed Accessed: 11.06.2020

What is a database in under 4 minutes, 2019, Linux Academy [YouTube]

Available from: https://www.youtube.com/watch?v=Tk1t3WKK-ZY Accessed: 27.8.2020

What is a graph database?, 2019, Simple [A]. Available from: https://simplea.com/Articles/what-is-a-graph-database Accessed: 20.8.2020

What is a Graph database?, 2020, AWS. Available from: https://aws.amazon.com/nosql/graph/ Accessed 23.7.2020

Which index to use when, 2020, ArangoDB, Available from: https://www.arangodb.com/docs/stable/indexing-which-index.html Accessed: 30.11.2020

APPENDIX 1. SUMMARY IN SWEDISH

Detta examensarbete har gjorts i samarbete med det norska företaget Limber AS. Limber har utvecklat sin egen molnbaserade webbapplikation Limber Projects som huvudsakligen användas av företag som arbetar med komplexa och leverantörbaserade projekt. För tillfället ligger fokus på olje- och gasbranschen. Eftersom Limber AS är ett relativt nytt företag finns det ett intresse att undersöka hur företagets kunder använder Limber Projects. Används applikationen enligt syfte, eller framkommer udda beteendemönster eller andra avvikelser.

För att kunna undersöka detta önskade uppdragsgivaren att en modell och prototyp skulle utvecklas som beskriver hur man kan spara användardata i en grafdatabas. Tanken var att man senare skulle kunna använda denna användardata för att undersöka användarnas beteende i Limber Projects. Limber och författaren beslöt tillsammans att en grafdatabas skulle användas för detta ändamål. För att kunna skapa och designa prototypen fattades beslut att använda ArangoDB som utvecklingsmiljö, eftersom ArangoDB är en grafdatabas.

Idag kan det vara svårt att undvika databaser i det vardagliga livet eftersom de är en viktig del av datoranvändningen i dagens moderna samhälle. Databaser brukar innehålla data som är sammankopplade till varandra. Ordet data brukar används för att beskriva fakta, och oftast samlar man in sådan data som anses betydelsefull. (Elmasri & Navathe 2016:33-35)

Databaser brukar ofta delas in i två olika typer, relationsdatabaser och NoSQL-databaser. Relationsdatabaser använder programmeringsspråket SQL för att modifiera data, medan NoSQL-databaser i större utsträckning är avsedda för dokument och använder sig av varken tabellformat eller SQL, något beteckningen även hänvisar till. (What is a database in under 4 minutes 2019)

I en relationsdatabas är data sparad i olika tabeller, och i varje tabell finns det kolumner och rader. Varje kolumn representerar ett visst ämne, som exempelvis ålder eller namn, medan varje rad kan ses som ett objekt. Exempelvis kan en tabell representera användare där kolumnerna beskriver relevant information kopplat till användare och slutligen representerar en rad en användare. (Vázques 2019)

NoSQL-databaser blev aktuella i samband med ökat intresse för big data. I jämförelse med relationsdatabaser så använder NoSQL-databaser inga tabeller för att lagra data och det är därför möjligt att lagra data mer flexibelt samt behövs ingen information om hur datan är lagrad för att kunna interagera med databasen. (Fowler 2015)

Grafdatabaser är en form av NoSQL-databaser och de är skapade för att hantera samband mellan dataobjekt som innehåller mycket information. Dessa samband kan ses som nätverk och graferna i grafdatabaserna är uppbyggda av noder (*node*) och av bågar (*edge*). Dessa bygger upp tripletter bestående av kombinationen ”nod-båge-nod”, vilket innebär att två noder är kopplade samman med hjälp av en båge. (Hurlburt et al. 2017)

ArangoDBs Community Edition användes som utvecklingsmiljö för att skapa prototypen. ArangoDB kan beskrivas som en multi modelldatabas. Beträffande NoSQL- databaser är detta relativt ovanligt. Vad detta innebär är att den är utrustad med grafdatabas, nyckelvärdedatabas samt dokumentdatabas, och alla dessa tre kan kombineras då man använder ArangoDB. (Advantages of the native multi-model database - ArangoDB 2020)

Dataobjekten som sparas i ArangoDB kan delas in i olika kollektioner och varje dataobjekt kan innehålla flera attribut. Då dataobjekten eller dokumenten är definierade lagras de i sina respektive kollektioner. Dessa kollektioner består av dokumentkollektioner som innehåller noder samt ”edge” kollektioner, som innehåller bågar. (Concepts 2020)

Datamodellering brukar användas då man planerar en databas. I en NoSQL-databas är data oftast sparad som aggregat, vilket gör att datamodellerna hos en NoSQL-databas kan se olika ut beroende på databassystem. (Atzeni et at. 2020)

Då en datamodelleringsprocess för en grafdatabas genomförs kan följande princip användas för att avgöra vilken data som skall vara sparad i noder och vilken som skall vara sparad som bågar. Det väsentliga är att formulera vilka frågor den lagrade datan skall kunna ge svar på. Detta innebär att man skall omformulera relationerna till meningar i vilka varje substantiv eller objekt är en nod, och varje verb en båge. (Allen 2019)

Webbmätvärden brukar oftast samlas in vid utveckling av marknadsföring, men kan även samlas in i syfte att förbättra företagsverksamhet. Oftast identifieras två olika sorters webbmätvärde, åtgärdbara mätvärde (*actionable metric*, författarens översättning från engelska) och glädjemätvärden (*vanity metric*, författarens översättning från engelska). Glädjemätvärden ger oftast mindre information även om värdena kan se positiva ut. Exempel på ett glädjemätvärde är antal nedladdningar eller antal besökare på en webbsida. (Reid 2019)

Genom en noggrannare analys av ett glädjemätvärde kan åtgärdbara mätvärden upptäckas. Med hjälp av åtgärdbara mätvärden kan en viss aktivitet kopplas ihop med en förändring vilket underlättar lösningar och därmed har en betydande inverkan på utvecklingen. (Austin 2018)

Den praktiska delen av arbetet gick ut på att skapa en modell (prototyp) för en fungerande grafdatabas i ArangoDB. För utvecklingen av denna prototyp användes ArangoDB Community Edition och testdata som skapades av författaren för att representera användardata. Själva processen började med datamodellering där tre olika meningar/frågor användes som bas, eftersom de insikter Limber var intresserade av kunde fås genom att svara på dessa frågor. Nedan är frågorna som användes översatta till svenska (originalen är på engelska):

1. Hur många sessioner har klickat på ”New Profile¨ - knappen?
2. Hur många sessioner har flyttat sig från ”Documentlist” -vyn till ”New Profile” - knappen?
3. Varifrån har en session/användare förflyttat sig för att komma till ”New Profile” - knappen?

Dessa frågor representerar väldigt specifika beteenden, men samtidigt standard beteende i Limber Projects som kan tillämpas i många olika situationer. Målsättningen i detta skede är endast att hitta en lösning till hur man kan lagra användardata, vilket betyder att sessioner istället används för användare i syfte att hålla användarnas identitet anonym. Eftersom integritet är viktigt, så är det i detta fall relevant att det inte skall vara möjligt att kunna spåra tillbaka till användaren. Därför användes sessioner istället för användare, och varje session blev tilldelad en unik sessionsID som inte skall vara kopplad ihop med användaren.

Frågorna/meningarna nämnda ovan användes inom datamodelleringsprocessen, och i början testades en uppsättning där substantiven och objekten i meningarna blev noder, och verben blev bågar. Detta innebär att vyer, knappar och sessioner var noder och verben som ”klicka” var bågar. Eftersom bågar i ArangoDB innehåller de viktiga attributen *_to* och *_from,* som hänvisar till startnoden och slutnoden, visade sig denna uppsättning inte vara den bästa lösningen.

Det slutliga valet och uppbyggnaden av noder och bågar resulterade i följande:

1. **Sessioner** – Dessa valdes till bågar och kommer att sparas i en ”edge collection” i ArangoDB. Det intressanta med dessa sessioner är att se hur en användare har rört sig under en session, vilket gör att det är viktigt att det framkommer i sessionen var man börjat och slutat. Förutom detta kan varje sessionsobjekt innehålla ett unikt sessionID- attribut, som genereras då sessionen börjar.
2. **Vyer –** Dessa valdes att lagras som noder i databasen och sparas därmed i sin egen ”document collection” i ArangoDB.
3. **Knappar –** Dessa lagras också som noder i databasen och kommer att lagras i sin egen ”Document collection” i ArangoDB

Efter att datamodelleringen var gjord återstod själva implementeringen. Som tidigare nämndes användes ArangoDB Community Edition i utvecklingen av prototypen, samt installerades på en dator med Microsoft Windows som operationssystem. För att kunna arbeta med databasen kan man använda antingen Arango Shell för att integrera med servern eller WebUI:n som är ett grafiskt användargränssnitt.

All data som importerades till databasen var skapad av författaren. Varje vy- och knappobjekt var manuellt insatta i sina respektive kollektioner. En större mängd data behövdes för att kunna testa tillgången till information gällande sessionerna, vilket innebar att det skapades en betydligt större mängda (ca. 50 000st) av dessa dataobjekt för att kunna testa AQL queries samt databasens prestation då dessa kördes. All information var slumpmässigt utvald i varje sessionsobjekt som skapades med hjälp av Python.

Körningen av AQL queries var en viktigt för att kunna testa och evaluera om datauppsättningen som skapades tidigare uppfyllde de önskemål som krävdes. Detta testades genom att köra queries som i teorin ställde samma frågor som användes i datamodellerings processen. Då detta resulterade i det önskade svaret kunde konstateras att datauppsättningen för detta ändamål var gjort på rätt sätt.

Ett annat sätt att testa uppsättningen i databasen var att undersöka hur den presterar då man ber databasen att exempelvis hämta data genom att köra en query med varierande mängda dataobjekt sparade och samtidigt se hur exekveringstiden ser ut. Testen kördes först på 5000 sessionsobjekt och slutade vid 50 000 objekt. Detta testades genom att först köra dessa utan att lägga till något extra index, förutom de som ArangoDB automatiskt skapar. Index används i databaser för att snabba upp exekveringstiden. Noterbart var att exekveringstiden blev snabbare då man hade lagt till ett index vid vissa attribut, i detta fall lades det av författaren manuellt till ett index på *session_id* attributet i session objekten. Samtidigt som dessa test utfördes kunde konstateras att exekveringstiden ökar samtidigt som datamängden ökar, dock är det inte frågan om en linjär proportionell ökning, men en kontinuerligt ökande linje.

Möjliga brister i detta examensarbete utgörs av att prototypen inte har testats in en verklig produktion med riktigt data. Därmed är det inte entydigt att denna prototyp kommer att fungera optimalt samt kan det framkomma oförutsedda brister i datamodellen som kräver modifikation. Dock skulle dessa brister kunna utvecklas till en intressant framtida undersökning, det vill säga prototypen kunde implementeras i en verklig produktion och därmed påvisa hur datamodellen fungerar i praktiken.