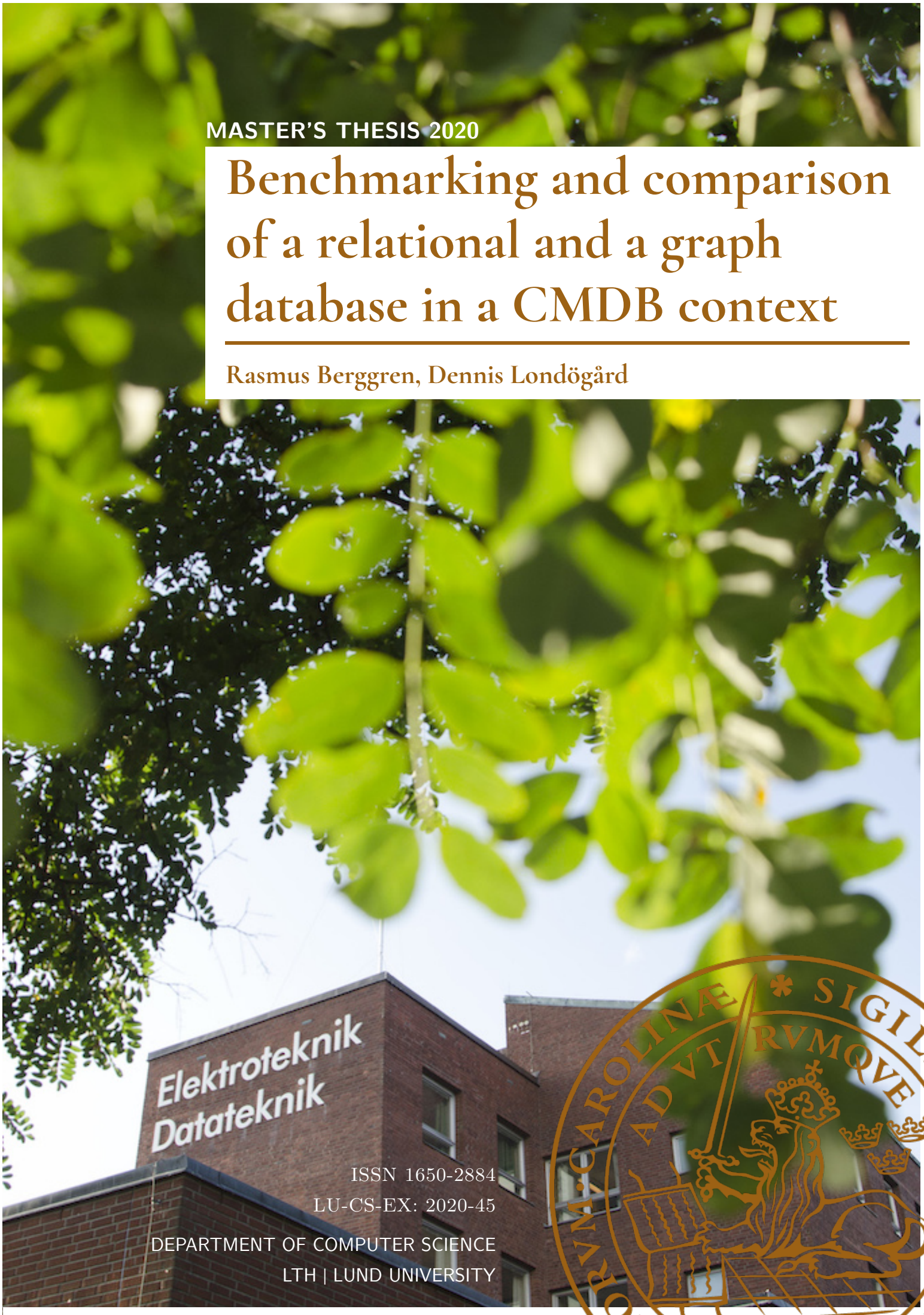
MASTER'S THESIS 2020

Benchmarking and comparison of a relational and a graph database in a CMDB context

Rasmus Berggren, Dennis Londögård



ISSN 1650-2884
LU-CS-EX: 2020-45

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-45

Benchmarking and comparison of a relational and a graph database in a CMDB context

Rasmus Berggren, Dennis Londögård

Benchmarking and comparison of a relational and a graph database in a CMDB context

Rasmus Berggren
dat15rb1@student.lu.se

Dennis Londögård
dat15dlo@student.lu.se

July 8, 2020

Master's thesis work carried out at Axis Communications

Supervisors: Guido Gudios, guido.guidos@axis.com
Lars Bendix, lars.bendix@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

The IT company Axis has grown a lot in the past decade and they are now having trouble keeping track of everything efficiently and hypothesize that they are in need of a CMDB, while also wondering what type of database it would be best implemented as. The goal of this thesis was to investigate what requirements Axis has for a CMDB, and use their requirements to analyse if it would be best implemented as a graph database or relational database.

The work was done in two parts. The first part consisted of making a requirement specification for Axis context, this was done with a combination of literature study, interviews as well as examining an old database used for CM purposes. A minimum viable product was created from the requirements to serve as a proof-of-work as well as a basis for the second part. The second part was a benchmarking based on the parameters of performance, writability, as well as readability. A combination of these three parameters was used to answer which type of database would be best for a CMDB.

By conducting the interviews we were able to create a CMDB requirement specification for Axis. The resulting specification suggests that Axis would benefit from an interconnected CMDB between the departments with a need for traceability. The results from the benchmarking showed that relational databases perform better than graph databases, while graph databases were better in regards to writability and readability.

We conclude that Axis would benefit from a CMDB within the company. Our research shows that both of the database types could be used for a CMDB, however we suggest that it should be implemented as a graph database since it would be easier to maintain.

Keywords: CMDB, Benchmarking, Neo4j, Graph database, MariaDB

Acknowledgements

We would like to thank Lars Bendix, our supervisor from LTH that guided us and gave us valuable feedback every step along the way.

We would also like to thank Axis for giving us the opportunity for doing this thesis, specifically all the people that participated in the interviews. A special thanks to the department of Axis tools and our supervisor Guido Guidos. Lastly we would like to thank Torbjörn Söderberg who went out of his way to help us with our thesis.

Foreword

This report is primarily written for Axis employees for further development, therefore some prior knowledge is assumed. The work uses languages where we presume that the reader has the knowledge of a master in computer science, with basic knowledge within the field of databases and is familiar to the concept of configuration management, but no advanced knowledge in these fields are required.

The work for this thesis was completed in full collaboration. Where Rasmus Berggren took a larger part in conducting the interviews, and Dennis Londögård took a larger part in writing the queries. However we were both heavily involved in every step in the process of the thesis.

Contents

Chapter 1
Introduction

1.1 Background

Axis Communications (Axis) is an IT company widely regarded for their commercial surveillance camera that has grown a lot in the past decade. They are now having trouble keeping track of everything efficiently and hypothesize that they are in need of an interconnected database between the departments. Their hope is that it would improve their productivity by having data that is important for multiple departments stored in one place. It would also benefit new employees by making it easier to search for information from one centralized location. This is because information currently is stored locally within each department by their own means.

Axis is composed of a lot of different departments that vary from everything between development and sales. Each department is as important as the other for Axis to thrive and grow as much as they have the past decades. This report was done with the help from the department of Axis tools who develop and maintain software.

As mentioned Axis has grown a lot the last decades and with this difficulties has arised with keeping knowledge accessible for new and old employees. They currently have trouble fetching important information about certain artefacts such as who is the owner of which system and which system depends on another. This creates dependencies on people that Axis wants to reduce. To do this they want to create a Configuration Management Database(CMDB) which is an essential part in software development to manage artefacts within the company between all relevant parties. Its main focus is to keep track of relevant Configuration Items(CI), which are the important artefacts, and their corresponding relationships to other CIs. The goal is to make it easier to fetch information and have a platform that includes every department at Axis. However as the work is carried out for Axis tools the main focus of the project will be on the needs of the developers.

To implement a CMDB a type of database needs to be chosen. Axis has a hypothesis that it would be best implemented as a graph database using Neo4j, instead of as a relational

database using MariaDB. The hypothesis is based on the fact that a CMDB tends to have a structure that resembles a graph and therefore should benefit from being implemented as a graph database. A benchmarking of three aspects, performance, writability, and readability, will be made. The benchmarking will be done to determine if the CMDB is best suited as a relational database or graph database.

A CMDB is not something that is made in one day, instead it should grow organically with the needs of the company. Therefore it will not be possible to implement the perfect CMDB in this master's thesis. Instead the focus will be on creating a CMDB requirement specification for Axis and implementing a minimum viable product (MVP) of it. This will be done to provide a proof-of-work and as a basis for the benchmarking. The proof-of-work should then be further examined and developed to create a more complete CMDB for Axis.

1.2 Problem formulation

The goal of this project is to create a MVP for Axis as a proof-of-work for a CMDB. To do this CIs had to be determined and use-cases will be created based on interviews, literature and older databases they currently use for CM purposes. The MVP will be designed and specific queries will be implemented based on use-cases. These queries will be evaluated based on performance, readability and writability. By comparing these aspects the intent is to make an informed suggestion whether the CMDB should be implemented with Neo4j or MariaDB. To do this the following research questions will be examined:

1. What are the necessary configuration items for Axis CMDB?
2. What would the CMDB requirement specification be for Axis?
3. Should a CMDB be implemented with a graph database or relational database?
   (a) What can/can not be implemented with a graph database?
   (b) What benefits can graph databases contribute that relational databases can not, and vice versa?

1.3 Outline

This report begins with introducing relevant theory, divided in two parts explaining CM concepts as well as database concepts, which are needed to understand the language used later on in the report. Then the methods that are used to tackle the research questions are discussed and motivated. The following chapter is where the results are presented and analyzed which is followed by a discussion of threats of validity, related works, and future work. At last, the conclusions to the research questions stated in the problem formulation are presented.

Chapter 2
Theory

This chapter contains the theoretical background that is necessary to get a basic understanding of all concepts and terms mentioned within the report. This would be beneficial to read if no prior knowledge is held within either the subject of Configuration Management (CM) or databases, to fully understand the language used within the report. CM and database concepts are two widely different subjects and their theoretical description will therefore be split into individual subchapters. Where one could choose to read the section which is foreign and skim the other one.

2.1 Configuration Management concepts

Research question one and two, mentioned in the problem formulation, requires some knowledge about CM to be answered since they involve CM concepts. The first research question involves Configuration Items which in themselves are identified by Configuration Identification, while the second research question needs knowledge of CM databases(CMDB). A CMDB is based on both Configuration Identification and Configuration Status Accounting. To be able to fully understand these concepts an introduction to CM will be made.

2.1.1 Configuration Management introduction

CM is an old concept that has been around and has not changed a lot for the past few decades. Even though it is an old concept the definition of what it really is can vary a lot depending on who is asked. This creates problems when discussing CM as a concept, since people can have varying opinions of the concepts as well as the name of CM terms. Therefore a generalized definition of CM and who uses it will be presented.

The purpose of CM is to provide consistency to a project throughout its lifecycle where the goal is to maximize productivity and minimize coordination mistakes. To accomplish

this CM helps the team to stay on track. It does this by identifying, organizing and controlling modifications to the project[2]. By keeping track of the history it is possible to assure consistency and provide the ability to retrieve an older version of the project. CM is divided into four main disciplines, Configuration Identification(CI), Configuration Control(CC), Configuration Status Accounting(CSA), Configuration Audit(CA)[7]. From these four principals only CI and CSA is relevant for this thesis, and will thus have one subchapter each explaining the concepts.

CM is a principle that is in widespread use by many companies. Software companies specifically, have two sub groups that take advantage of Software Configuration Management (SCM), managers and developers. Developers may recognize SCM when using a version control tool, such as git. SCM helps the developers coordinate their work by having access to the components of the product and history of these components. Providing a stable working context for changing the product and coordinates for concurrent changes[8]. While the managers use it in a broader perspective where the goal is to identify product components and changes, to control changes, and to record and report the history and status of the product[8].

2.1.2 Configuration Identification

Within the field of CM, artefacts are the components and items that constitute the product and all relevant assets within the company. This could essentially be anything depending on the context or the given company, it could be anything from source code to an informal email within the company. Configuration Identification is the process of identifying which of these artefacts that in some way are important for the company and should be protected with CM principles[6]. Protecting these artefacts means that they are safely stored, version controlled as well as preventing unwanted changes to them. A good guideline for establishing whether an artefacts is important or not is to ask "Would our ability to deliver the right system, on time and within budget, be impacted in any way if a particular document, drawing, piece of software or hardware kit were lost or corrupted, or were used incorrectly or at the wrong version?"[12]. These would then be classified as Configuration Items(CI). Both Configuration Identification and Configuration Items are denoted as CI, from here on in this thesis CI will therefore stand for Configuration Item. The CIs needs to be granulated to an appropriate extent and the appropriate metadata for the items needs to be saved for proper Configuration Identification[6]. This granularity and metadata differ between company and context. One company may for example consider a license or server to be a CI while another company would not.

2.1.3 Configuration Status Accounting

CSA is the ability to record and report the status of a project and its configuration items at any given time. Therefore CSA depends on that the information is of good quality, both in regards to Configuration Identification and that it is safely stored in a CMDB[6][5]. The right CIs have to be identified with their corresponding metadata and relations, and should only be modified within the CMDB if there is a reason for it. The goal of CSA is to provide visibility of the project and does this by providing a way to fetch metadata information about CI and what relationships they have to each other. The relationship between these items is a

central component of CSA and is what is called traceability, which can help when for example wanting to calculate the impact a specific change has on the project[6].

2.1.4 Configuration Management Database

A CMDB is a tool used in CM to safely store CIs, their attributes as well as relationships between the items[24]. This database has the ability to provide CSA and should therefore support all relevant queries so that the data can be fetched in multiple ways[22]. To trust the CSA the data has to be consistent within the CMDB. This means that not everyone should have the ability to modify the database, but only authorized personnel, and that the data is consistent even if something were to happen to the database, such as a system failure. Figure 2.1 shows the process of an authorized personnel inserting a CI into the CMDB as well as a user performing CSA on the same database.

It is difficult to create the perfect CMDB in one go, as it is something that dynamically grows with the needs of the company. There is no universal CMDB that fits everyone since every company has their own needs and therefore identifies their Configuration Items differently. Instead it has to be a flexible database that can be dynamically changed throughout the project's lifecycle to fit the new needs that might be discovered[22]. This would mean that the database would potentially grow to be very large and would therefore need to be able to handle a large amount of data. A CMDB could be used by many different employees and would potentially be under quite a lot of pressure[22][5]. In return it should be able to handle multiple simultaneous requests.

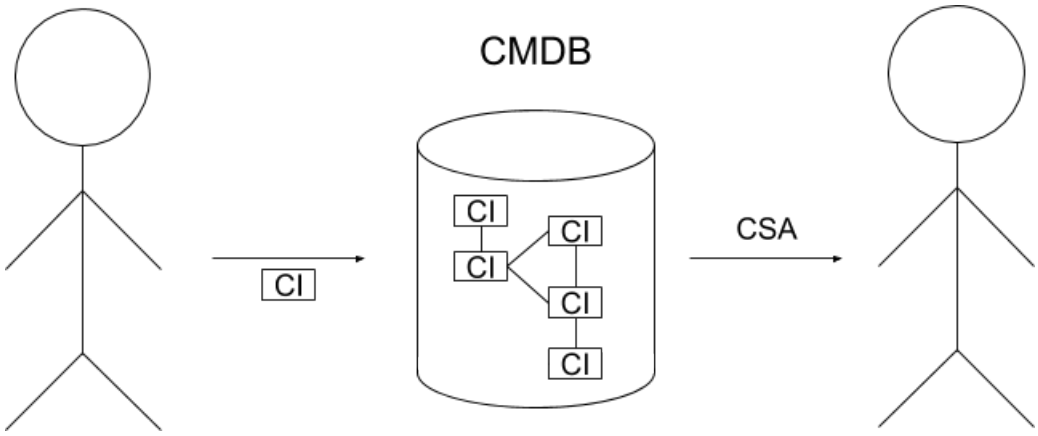


Figure 2.1: Graphical depiction of how the terms of CI, CSA, and CMDB relate to eachother

2.2 Database concepts

The third research question asks whether a CMDB should be implemented as a graph database or a relational database. To answer that question a general description of terms and concepts of what a database is and what benefits they may provide is given. A deeper explanation and theoretical comparison will be made between relational databases as well as graph databases. Which will serve as a base for arguments made later in the thesis.

2.2.1 Database background

Databases is an old concept that started gaining popularity in the 1960s when navigational databases and the network databases were the most common[18]. However these early systems were inflexible therefore in the 1980s the relational databases grew in popularity. More recently with the growth of the internet NoSQL databases have gained a lot of popularity for its ability to perform queries on unstructured data and for its performance[18]. This would potentially be beneficial for a CMDB since its data should be flexible and therefore resembles unstructured data.

The similarities between databases is that they all strive towards storing data in an organized way. The difference between databases comes in how the data is stored and how one accesses it, most databases make use of queries for the user to manipulate the database. Queries are a way to access, manage, modify or update data[18], and could be seen as the means by which a user exercises their use-cases of the database. A specific use-case might be "fetch all people that are less than a certain age" and with the right query, that could be achieved.

There are two important factors to consider for any database, especially for a CM database, to be able to handle a large amount of data without suffering drastically in performance as well as ensuring a high quality of the data. A set of common properties to evaluate data quality of databases is the acronym of ACID, which stands for atomicity, consistency, isolation, and durability[10]. Atomicity guarantees that all transactions that are made in one request is done or that none of them are done. Consistency ensures that all data is consistent with the corresponding constraints such as data type restriction, that a date is in fact a date for example. Isolation ensures that all queries are not affected by other queries, meaning that two people should not be able to update the database at the same time and override each other. Durability means that if a query has been successfully committed to the database it will stay within the database even if a system failure occurs. All databases do not support ACID because of different reasons such as ensuring a high availability[10]. Therefore a developer has to make a choice when choosing a database to pick one that fits the needs of their context.

2.2.2 Relational database

As mentioned, relational databases became popular in the 1980s and have been popular ever since. Relational databases' biggest strength in the 1980s were its standardization of how the database was represented and the way that queries were made. Relational databases store their data in tables which consist of rows and columns where each column is an attribute and each row is a new data point stored in the database, as could be seen in figure 2.2. These tables make it clear what the data point is, which is a strength for relational databases[19].

MariaDB is one relational database that is popular, open source and guaranteed to stay open source. It is a fork of the hugely popular MySQL, a non open source database; both share similar features and both use the ISO standard, Structured Query Language(SQL)[11]. Most relational databases use SQL which makes it easy for the user to switch between different types of relational databases and a large community of developers are familiar with it.

2.2.3 Graph database

As a response to the growing complexity of data and technological advancement came the non-relational databases, also known as NoSQL. Non-relational databases is a relatively new technology that has gained popularity in the last decade[18]. The selling point of these databases is that they, as opposed to relational databases, do not need to be pre-defined to the same extent and can instead be dynamically adapted to the scenario where a static table needs to be defined beforehand. While relational databases are table-based that is not the case for all non-relational databases. These databases vary from document based, key-value pairs or graph databases, among others[14]. These databases are tailor made for specific scenarios and are therefore good at one specific thing but not necessarily useful in all situations.

Graph databases are, as the name suggests, made specifically to efficiently handle data structures that resemble graphs[17]. In other words a structure with a large set of relationships, potentially benefiting the traceability of a CMDB. Instead of tables, representing the data with nodes and edges, as can be seen in figure 2.2. One of the currently leading and mainstream graph databases is Neo4j[16]. Neo4j is a commercial graph database consisting of both a paid enterprise edition as well as a free community edition. As opposed to SQL databases that use similar query languages between the databases, that is not always the case with NoSQL databases where almost every database technology has their own query language. Neo4j has developed their own query language called Cypher which is optimised for graph traversal while keeping to the familiarity of SQL by being a declarative query language[15].

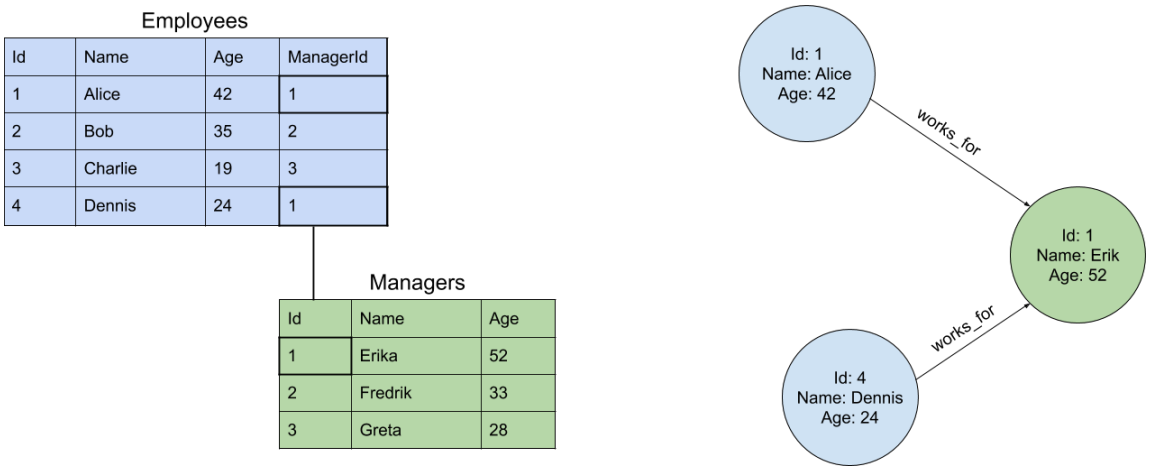


Figure 2.2: Depictions of a relational database to the left and a graph database to the right

Chapter 3
Method

This chapter starts by establishing the context of the thesis, which would be useful to understand why certain choices were made. The chapter continues by discussing the possible methods that could be used to answer the previously mentioned research questions, and a motivation for why we decided to do as we did. The thesis was divided in two parts, the first one focusing on forming a requirement specification for a CMDB, while in the second part consisting of a benchmark comparison of two different types of databases. This chapter will reflect this by being separated in two subchapters representing the two different phases. A description of how each method was carried out will be given, providing the reader with enough knowledge so that they may reproduce the project.

3.1 Context

Axis Communication started out in 1984 with the intent of creating new innovative network solutions for digital devices, initially starting out by providing network printing services. They have kept this aim over the years but have expanded and transferred their focus to new products, today mainly producing and developing network surveillance cameras. Today they develop and manufacture their own cameras that are sold to multiple distributors. In return this means that Axis has grown quickly to a large company with many departments, consisting of project managers, sales team, marketing teams, hardware developers, software developers and more to be able to provide the service they have. All these teams have to coexist and work together towards a common goal for Axis to continue to succeed and prosper. Axis has grown consistently, which in part is thanks to that every department has the possibility to adapt their workflow to what is most efficient for them, not having to follow any strict standards.

Since its beginning in 1984, and especially the last decade, Axis has grown to become a big company with over 3500 employees[1], but with growth comes complications. The company has become quite dependent on their employees' knowledge of where certain information is

stored and how certain systems work. This is a problem when new employees start at Axis since the work is an uphill battle from the start. It is especially problematic if someone leaves the company that holds a lot of information.

This thesis is carried out at Axis tools which is a software department that develops new software as well as maintains old software. Axis tools are divided into minor teams that are owners of their own systems. These teams work in close relation with project managers when maintaining and creating new applications. Therefore developing a CMDB would fit this department. An important factor for a CMDB is that it should be suitable for the whole company to provide some kind of coordination between departments, and therefore multiple departments have to team up to create a suitable and sustainable database. However since this is a thesis and is done at Axis tools, a decision was made to focus primarily on software developers and their needs for a CMDB and have other departments as a secondary goal.

3.2 Methods for phase 1

To answer research question one and two, which are "What are the necessary configuration items for Axis CMDB?" and "What would the CMDB requirement specification be for Axis?", there was a need to combine different methods to reach a conclusion. Both questions have a need of understanding best practices when performing configuration identification as well as forming a requirement specification, and it was therefore essential to explore best theoretical practices as a basis, followed by consulting the employees at Axis to verify our findings and determine what specific needs Axis has.

A complete overview of the process of phase 1 can be seen in figure 3.1. The process starts with a literature study to get a deeper understanding of what a CMDB is and what requirements there should be on a CMDB in general. With background knowledge from a CM course we had learnt the general principles of CM but lacked specific knowledge about CM databases, therefore our literature study was focused on the concept of CMDB. To create a CMDB requirement specification for Axis we had to understand which problems they have. This was done by interviews and an analysis of an old database used for CM purposes. The interviews were conducted to investigate current problems that need to be addressed. To analyse old problems that Axis has faced and how they addressed them, we examined their current database.

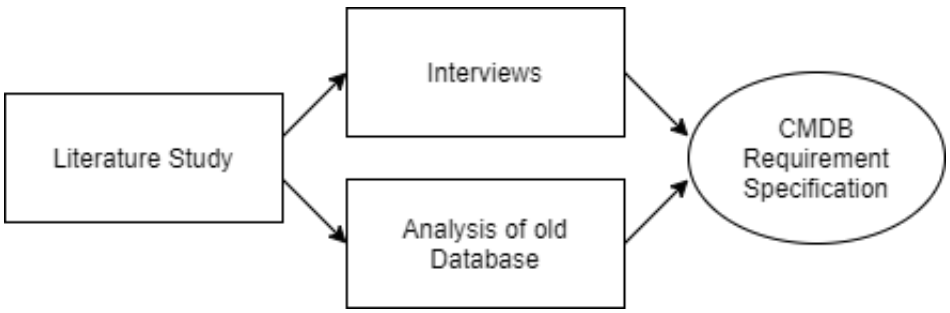


Figure 3.1: The process for phase 1

Other solutions that were considered to reach more people was to interview other companies or use surveys instead of interviews. By interviewing other companies the focus would

shift from being specific for Axis and therefore not generate a fair result for the thesis. Surveys would reach a wider group of people however we wanted to have an open ended discussion with the interviewees which would not be possible with questionnaires. By not having direct contact with the participant it would be difficult to assure that everyone had a common understanding of the concepts of CM and CMDB. Therefore we believed that we would receive more useful information from interviews. The interviews were also held with key people from each department, except software where there were more varied personnel, which were specifically chosen to get as broad a perspective as possible with a limited number of people. Since it was possible to interview key people we reckoned that it would give the best result in the time frame we had. Another reason for doing interviews only at Axis was that it was accessible because the work was carried out on location at Axis.

The literature was mainly found by using different online search engines, namely google, google scholar, and LUBsearch. Using specific keywords to find articles, papers and white papers, trying to mainly base our knowledge on peer reviewed papers. Letting ourselves be inspired by any relevant information we could find, while being more critical of non peer reviewed sources. The filtering of relevant literature was made in three steps where we first search with our keywords and read the titles. If the titles seemed relevant we read the abstract of the paper to get a more thorough understanding of it. If it was still relevant we read the whole paper and discussed it together and decided if it was a paper that could benefit our thesis in any way.

The goal with the literature study was to search for similar works that could provide inspiration of how we could carry out the work and what results other companies suggest that fit their context. It was also made to see what literature suggests that a CMDB should consist of, what their examples of Configuration Items are and what a requirement specification should be. By evaluating the literature some preliminary ideas of requirements and CIs were determined to be essential, and needed to be verified before the interviews began. Therefore we see no way of not including a phase for literature study in this thesis, as it was an essential part of the process.

As mentioned earlier, a CMDB should coordinate a company by storing important data in a central database that is used by multiple departments. To ensure that this would be the case, it was required to talk to different departments within Axis to get a wider view of Axis specific requirements. This was achieved by interviewing different people in multiple contrasting departments, formulating their needs as the users' use-cases for a CMDB, representing the requirements of Axis. Since the work was carried out from the perspective of axis tools, a software department, this was our outset and main point of focus. The departments that were interviewed were; IT, Governance, Project Management and Software Development, to achieve as varied a view of the requirements as possible. In total nine interviews were conducted where four people worked as software developers. These four interviewees had quite different backgrounds from working one year to over twenty years, and worked in different teams. This was done to get a wider perspective of the needs of software developers. While the remaining interviews were one too two key people at their respective departments, representing the whole department.

Before conducting the interviews we needed to decide how they were supposed to be carried out. The goal was to create a dialog when interviewing, but with some specific topics to discuss. Therefore we decided to go with semi-structured interviews which means that we had some predefined open questions, meaning a question that can not be answered with a

static yes or no response[20]. The questions changed throughout the interviews based on how the interviewees answered and if some questions were hard to understand we would then try to formulate it in another way. Important to consider is that the topics for each interview were never changed, just reformulated. Sometimes it is recommended to initially have test interviews to try out the questions before the real interviews start, to ensure the validity and quality of the questions[20]. This was not done since we believed that it would require a larger time investment than we had decided to allocate for this part of the process. Since we wanted to create an open-ended dialog we would be able to formulate the questions during the interviews, adapting to the conversation and still get a viable result.

A problem with the interviews is that they took longer than expected, taking approximately four weeks to carry out in total. This was because the people that we wanted to interview had other more important matters to attend to and that people had to reschedule because of sickness or for other reasons, which is something to always consider when conducting interviews. Another problem that became apparent was that some interviewees had not heard of the term CMDB or CM, and there were others that were not completely sure what a CMDB should do. To deal with this we introduced CM and CMDB for the interviewees at the beginning of the interviews to have a common understanding of the concepts. However there were still some requirements and use-cases that might not be within the scope of a CMDB that were proposed, that needed to be filtered out.

In the down-time between each interview we decided to spend our time as efficiently as possible, deciding to examine one of Axis' current CMDB solutions. A database called "CMDB" which is owned by Axis tools. This was examined to see which use-cases it tried to solve and how useful it was for the company at this point. However early on in the examination it was noticed that no one at Axis currently uses the database, and there was only one person that had, at one point, used it. No one had maintained the database for some time, making it obsolete, which is the main contributing factor to why it is not in use. It also became clear that the database had one specific use-case, to solve a single person's requirement, and would, according to the definition of a CMDB, barely be classified as such. Therefore there was not much focus on this database, just a short informal conversation with the person that had used it and consulting the documentation as well as source code.

A requirement specification for Axis was formed from a collection of the gathered information from literature, interviews as well as the examination of the old database. The literature provided a base of knowledge from which we could examine if common CMDB requirements would be relevant for Axis during the interviews. The interviews were the main source of requirements, being the only source revealing Axis current needs from across the company. Comparing the interview requirements to the theoretical ones together with the examination of the old database gave the CMDB requirement specification for Axis.

By narrowing this specification down to the essentials, a Minimum Viable Product(MVP) was formed with an accompanying UML diagram. The MVP was made to give the best representation of the CMDB and where it was possible to showcase as many use-cases as possible. A lot of use-cases were quite similar in functionality, such as selecting different CIs by the same data type. Then one use-case would be sufficient to cover many, limiting duplicate work. The goal with the MVP is to implement it as a database in phase two for benchmarking, used as a basis for research question three.

It would be possible to implement all use-cases with all requirements. However it was believed to be another waste of time since the goal was simply to create a proof-of-work of

a CMDB for benchmarking purposes more so than a complete product. Creating a complete and functioning CMDB that would be useful for the company would require much effort and would be a waste of time for our thesis if it were not desirable. Instead the goal was to create a MVP that could showcase all important benefits of a CMDB that were gathered throughout phase one and that could be benchmarked. Another reason was that Axis own employees are better than us at implementing databases and therefore would most likely have done a better job than we would when implementing the database. Therefore it would be a waste of time from our part to implement something that could be better made.

3.3 Methods for phase 2

To make a systematic exploration of our third research question, "Should a CMDB be implemented with a graph database or relational database?", a comparison had to be made. As can be seen in figure 3.2 the MVP requirement specification were both implemented in Neo4j free community edition, a graph database, and MariaDB which is a free and open source relational database. To accomplish this we had to learn Neo4j since we held no prior knowledge about the subject, however prior knowledge about MariaDB had been learnt from a prior database course. These two database types were then benchmarked and compared on three parameters, writability, readability, and performance. The parameters were chosen by similar works and the goal is to combine these parameters to answer the research question.

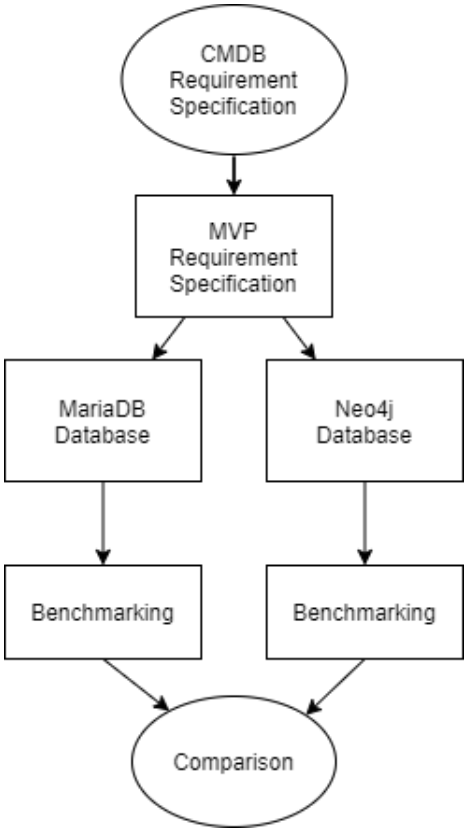


Figure 3.2: The process for phase 2

Other parameters to consider could be how efficiently the database handles the data

storage or how much data the database can manage. The efficiency indicates how much actual memory any given data would occupy on a hard drive, this would not be relevant in many modern contexts where memory is cheap and no longer an issue but could be worth considering how it scales for large amount of data. How much data the database actually can manage before it becomes unusable or unstable is also worth considering when handling large amounts of data, like big data, this is however not the case for this context and was therefore not considered either.

To be able to test and benchmark these two databases and see how they would scale we decided to go with two differently sized datasets. The datasets consisted of randomly generated, but realistic, data points as can be seen in figure 3.3. The small dataset was a realistic size to how large Axis currently is and the other dataset would resemble a larger organisation, to what Axis could grow into in the future. The sizes of these two were made by an approximation of our supervisor. These two sizes would make it clear how the two databases would scale with more data and a comparison would be able to be made. It would be possible to test with a third dataset that would be even bigger. However that would not currently be relevant for Axis to examine, and is therefore not part of this thesis. Another reason to not use a third dataset is the time it would take to prepare data, benchmark it and analyze the results which we did not deem necessary.

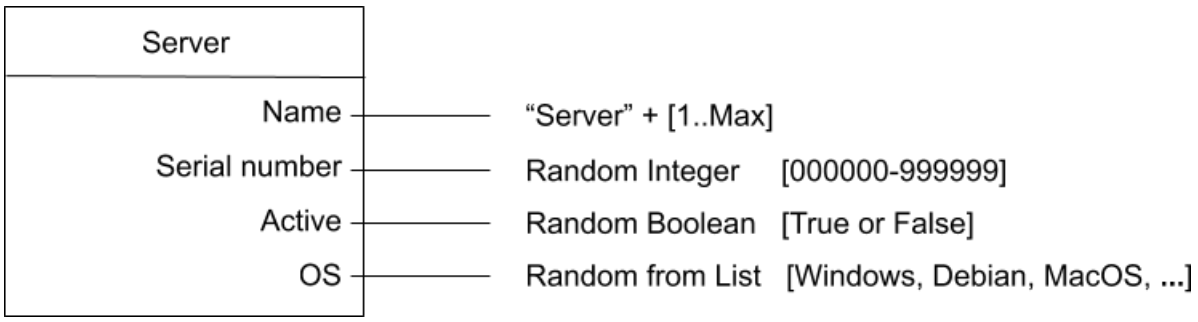


Figure 3.3: Generating random data for a server

The data insertion into MariaDB was easy since the UML of the MVP was converted into a UML that fits MariaDB and then the tables could easily be made. The only difficulty was that the data had to be inserted in a specific order since a relation could not be created before the parent was created. E.g every system has an owner and therefore the employees had to be inserted before systems. Neo4j on the other hand does not have any predefined tables since we use the community edition. This edition does not allow you to create predefined labels. Another issue with Neo4j is that it was incredibly slow to insert data. The big dataset took about 7 hours to insert. There is an admin tool that could be used for inserting data into an empty Neo4j database, but that was not something we deemed worth our time examining how it works and instead used a simple but time consuming insert query, since it was not part of any test. Another problem that arose with Neo4j was that it quickly filled the cache memory with log files, which Linux allowed but it created complications. To fix this modifications were needed in the config file to not save as many log files and instead remove old files more frequently. However this was noticed too late and the computer's cache was filled two times which took one day to solve. Another situation arose when we decided to update to the latest versions of both Neo4j and MariaDB. We did not have the latest version of our operating system, resulting in us not having the latest version of MariaDB. This re-

sulted in us spending one day on letting our computer update. Neo4j released a new update during our thesis, which we decided to update to. However during the updating of Neo4j something went wrong resulting in the visual interface of our operating system disappearing, which made it necessary for us to consult IT.

The benchmarking measured three different aspects, writability, readability and performance. As can be seen in figure 3.4 together they form the result of the benchmarking. Writability is the difficulty of initially writing a query. Readability determines how hard a query is to read and understand, this resembles how it would be to maintain the code. Performance is the execution time of each query. These three parameters fulfill the lifecycle of a system from developing, using and maintaining it. Writability and readability were evaluated by subjectively analyzing what we thought about these two aspects. It was evaluated by our impression of complexity and the lines of code. Therefore the results of these two parameters will not be perfectly objective in an academic scenario, solely being based on our experiences when working with the languages for an extended period of time. However we believed that it was the best we could do within the time frame and our expertise. Another solution to analyze writability and readability would be to use token analysis[23]. This is not something that we know how to do, and would take a considerable amount of time to carry out and was therefore not done.

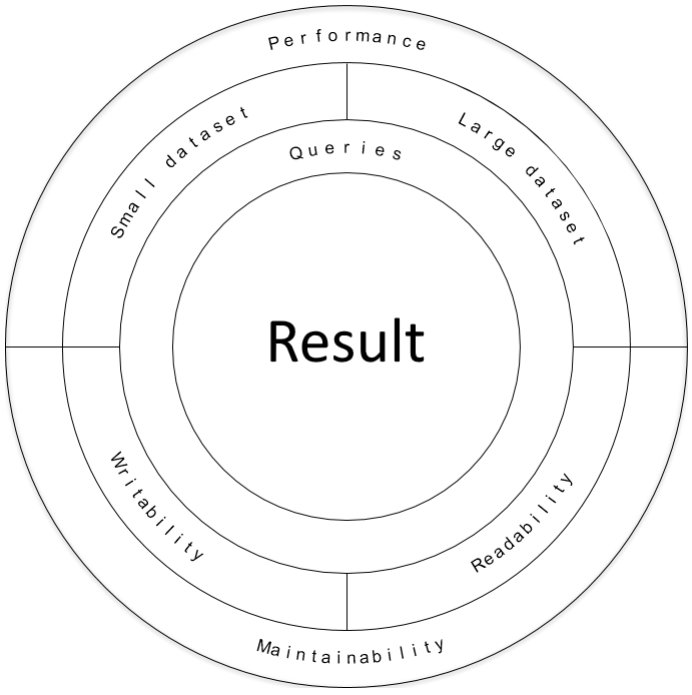


Figure 3.4: The parameters that were measured during the benchmarking

The benchmark was measured on eight queries which were derived from the use-cases discovered in the first phase. Creating the queries in such a way to cover all use-cases with as few queries as possible, covering similar use-cases with a single query. E.g fetching the name of one CI would be identical for all CIs. The same query was implemented in both cypher and SQL to be tested in both databases. The experiment consisted of filling the database with a small amount of data, running the queries 500 times each and saving the time for

each query. Then doing the same for the big data set, each query being executed 50 times, this because the execution times ran remarkably longer. This was implemented using python where we used a library simply called Neo4j for Neo4j, which is officially supported by Neo4j themselves, and used a library called pymysql for MariaDB.

The time was evaluated by calculating the average of both types of databases and by comparing how they scale from the small to the big dataset. The average is calculated by first removing the top 10% and least 10% of the values, this to remove divergent numbers caused by external processes that could negatively affect our results.

Chapter 4
Results and analysis

This chapter evaluates the results and conclusions reached in this thesis. This is done by analyzing the data obtained from applying the previously described methods, and then presenting the results. Split in three parts where the first part answers the first two research questions, “What are the necessary configuration items for Axis CMDB?” and “What would the CMDB requirement specification be for Axis”?. The second part is an intermediate step taking the system requirement from the first part, turning it into a minimum viable product(MVP) used for benchmarking in the following phase. The last part benchmarks the MVP to answer research question three, “Should a CMDB be implemented with a graph database or relational database?”.

4.1 Analysis and results of CMDB requirement specification

To create a CMDB requirement specification with suitable CIs for Axis we have to analyze the results of the three methods used in phase one. First we analyze and present the results from the literature to find general requirements for a CMDB. The interviews are analyzed, presenting a short description of the interview process followed by the resulting requirements found during the interviews. Lastly the old database is examined, finding what use-cases the database was made to fulfill. After analysing these three methods individually a compiled result is presented, showcasing the complete requirement specification.

4.1.1 Analysis of literature study

No universally good CMDB exists that is optimal for any given context[4][7]. This, because no two given companies are identically alike, having different requirements. But there are however some general design principles and concepts that are worth considering regardless

of the context. This is why a literature study was a good base for this project, creating a base from which subjective input from the potential users would then be needed to form it into something that would best fit the context.

An initial step when designing a CMDB would be to define the configuration identification. To define what counts as CIs and what we want to save in our CMDB. As mentioned in the theoretical portion of the text, there are many definitions, that vary slightly, of what constitutes a CI. Kelly describes it as something that would impact our ability to deliver the right system on time within budget, which could essentially include a huge variety of items[12]. According to Daniels, a CI is part of the product or is the product itself[6]. His definition focuses more on the product instead of everything that affects the product. Therefore we chose to follow Kellys definition during this work, because the definition is broad and easy to implement in the interviews. The goal was to have an easy to understand and broad definition for the interviewees for them to keep an open mind.

With such a broad definition of CI it is easy for the CMDB to quickly grow in unmanageable complexity. It is important to focus on the CIs that affects the final product the most and then continue expand if needed. By doing this the risk of letting the CMDB become too complex is reduced since it will not be perfect from the beginning and instead be good enough. About half of all CMDB projects fail and the reason is almost always the complexity and unreasonably high expectations of a CMDB[4]. Therefore it is important to reduce the complexity wherever possible. One way to do this is to analyze the granularity that is needed for each CI[6]. Depending on the context a single laptop could be a CI, however in another context it could be important to keep track of the individual parts in the computer. Having an appropriate granularity may reduce the complexity in the CMDB which in turn could improve the success rate[4].

Another reason for CMDB failure is to not have a clear view of what a CMDB should, and should not do[13]. The goal of the CMDB is to store centralised data and provide the relationships between these, essentially providing CSA[3]. For the database to provide CSA it needs to be able to report and manipulate the current version and the complete version history of all CIs[3].

Sometimes it is unavoidable that some departments, such as governance, have a need for a homegrown database that is relevant for only them[13]. By identifying these databases and understanding that they may not have to be centralised in a CMDB the complexity will be reduced. In the long run it will help produce a better quality reporting of the traceability.

To ensure that the CMDB is easy to search within and is useful for different departments within the whole company the naming of CI is important to consider[6]. Significant naming of CI making it easier to identify what they are and do, specifically should naming of software be descriptive of their functionality. It would be important to be able to change every aspect of a CI if the need for a change arises, being able to change its name or any attribute associated with the CI[22]. This is important for flexibility, if it is revealed that an item needs to change with the evolution of the database and the company. Within this thesis no real attention will be made regarding the names of CIs, since it will not affect the benchmarking or the proof-of-work. However the flexibility will be important to consider to show that it could be a complete CMDB for Axis.

When choosing the type of database to use when implementing a CMDB there are some important aspects to consider, such as performance and that the database supports CSA which requires different kinds of relations[22]. Since the CMDB is supposed to be a cen-

tral database within the company, there will potentially be a lot of requests to the database. Therefore it needs to have good performance for a variety of queries. The kind of queries may vary from simple get requests to mass insert of data points. Axis has no need of good performance regarding mass inserts since these can be executed during the night time and would not occur often. However the performance of other queries, such as updates, get and simple inserts are important since these would continually be run throughout the day by multiple people at the same time. Especially the get requests would be important since fetching data will be done by all users, while not that many will insert or update the data. There is an upper limit to how good performance that is required. As long as the queries take a reasonable amount of time it does not matter how long they take. What counts as "reasonable" would differ from one's perspective, but a query taking 0.002 sec or 0.01 sec would not generally be noticed by someone even though the difference is 5 times. For the CMDB to support CSA it has to be able to support different kinds of relations which are one-to-one, one-to-many, and many-to-many[22]. This to support the flexibility that is required of a CMDB so that it is possible to symbolize different scenarios.

To be able to trust the data in the CMDB we have to assure that the data is inserted correctly and that it is up to date. If we can not assume that the data is up to date, no one will use the database since they can not trust it. As mentioned above the CMDB is supposed to provide CSA, which is the status of the project at any given time, and therefore the data has to be up to date. Other than that the data is the latest version available, it is also important that the data is correct. One way to assure this is to avoid any unwarranted changes to the database by restricting the writing access to only authorized people[5]. This would make the database less accessible since not everyone can use it when they see a need for it, and it would provide a delay of when the data is updated. This would make the database more reliable and would benefit the database in the long run keeping it from growing in an undesirable way. Therefore there would most likely be more benefits than drawbacks by having authorized personnel managing the database and just letting all users fetch the data.

Information may be stored in external databases where they are managed. To assure that the CMDB is up to date with these external databases the CMDB has to support integration with other databases[22]. Since Axis has multiple smaller databases it would be beneficial for them to support an easy integration into the new CMDB. This would also allow for future maintenance in these smaller databases instead of letting everyone work with the CMDB directly. However we do not know how these smaller databases look like and would have to examine how this requirement could be accomplished for Axis. Therefore we decided to not put any effort towards integration with other databases, to save time and resources.

Since the database stores important data it is required that the database can handle system errors. The database has to be able to go down and still keep the data intact in storage. Otherwise it would be useless, e.g if the power went out we still want all information to be there when the power comes back. Therefore it is important to consider this when choosing the type of database, since some databases provide this kind of security, by having the properties of ACID, while others do not. Since the data is important someone might want to try to attack the database to receive this data. However this is not something that is relevant for this thesis since it is not solved within the database, instead being solved externally.

These were the resulting requirements that the literature study concluded in. They are separated into three parts, namely CI, CSA, and database requirements to define what is CM related and what is a requirement on the chosen database.

CI

- Definition of CI
- Appropriate granularity
- Store centralised data

CSA

- Provide relationships
- Store version history of all CI
- Effective CI naming

Database requirements

- Access control
- Data is correct
- Data is up to date
- Performance
- Be able to change every aspect of a CI
- Synchronize with external sources
- Possibility of different kinds of relationships

4.1.2 Analysis of interviews

The knowledge from the literature review served as a base of general needs of a CMDB. The interviews however were used to find specific requirements from Axis. Verifying if the knowledge found during our literature study was relevant for the context and what specific CIs and other use-cases Axis has. This was done to create a less general requirement specification that would be tailored for Axis needs.

The interviews were carried out in a semi-structured fashion, meaning that there was an initial guide used during the interview assuring us that we covered all the subjects that we deemed important. The interview guide was designed to cover the important subjects for a CMDB, namely configuration identification and CSA. Creating a general interview guide that would have relevant questions that would be applicable for all departments was not really possible, because of how different the departments operate. There are therefore some questions that had to be specifically asked for specific departments. There are also some questions and topics that came up without being part of the interview guide as a result of the interviews being semi-structured. To see the complete interview guide refer to appendix C. It is worth noting that this interview guide is writen in swedish and was made for us, and could therefore be hard to understand from an outside perspective.

The interviews were divided between four different departments and eight employees having quite varying views of what a CMDB should do and what needs they have from it. These departments being, Software developers, IT, Governance and Project managers. The resulting requirements from the individual departments will therefore be analysed and discussed individually making a combined requirement in conjunction with the analysis of the old database later, in subchapter 4.1.3.

People had different opinions regarding if an effort should be made creating a unified CMDB for the whole company or in what breadth the database should reach. Some wanted a limited scope only covering their own department while others considered it essential to cover more of the whole company. The reason could be because everyone had varying views of what a CMDB is. However, as mentioned earlier, according to the literature study, the CMDB should be a centralized database and not only for a single department. It is possible that some departments require a specific database that fits one specific requirement, but then it is not a CMDB. These opinions would probably differ even more greatly if more departments were to be interviewed. However because of the timeframe of the thesis and our focus on the software department we deemed this to be an appropriate scope for the project. In some way being a study if there is a wider interest or need of a central CMDB within the company, which from our research would appear to be the case.

A total of four software developers were interviewed, this department being our main focus. The background between each developer varied to a great extent, involving a veteran developer having worked at the company for over 20 years as well as a junior developer having worked little over half a year at the company. As well as consulting developers from different departments, getting as wide a perspective of the needs as possible. As one could imagine most of the needs from the developers were regarding the source code and CIs related to the code, like tests. A need to know what code exists avoiding creating the same product within the company and how this code depended upon each other, as a form of traceability, was apparent. Axis currently has a lot of different databases and three git platforms making it hard to find certain systems or to even know that they exist. There was a need to have an easy way of finding what services that exist based on a description of the system or keywords, which would be achieved by having appropriate CI names and attributes. This was a problem that was apparent specifically with junior developers, being hard to get acquainted with the systems. The time spent by junior developers getting familiar with the available resources would be minimized by getting a better overview of the systems and their dependencies, making the developer more efficient. Tracking and updating all these dependencies and assets would be a huge effort, the general consensus was that this process would need to be done with automation in some way. Doing this manually would take a lot of time since a single CI might be updated multiple times a day.

Two people were consulted within the department of IT. Their needs varied quite a bit from the ones from software. IT cared less for code and more for a variety of assets. Assets vary from licences to individual computers but with somewhat of a focus on server tracking and management. An asset manager was needed to map the servers and to track attributes regarding the servers. IT had a similar problem to software where there were a lot of different locations where information was stored but not collected in one location. Some asset managers simply being excel documents or some relying on information someone knows and has not written down. A central database to collect all information was seen as the highest priority for a CMDB, and was something the department was currently investigating. Another

important factor was role based access management, limiting who can access and edit the assets and attributes within the database. Some needs were brought up during the interviews that IT deemed essential but could be argued to be outside the scope of what a CMDB could or should provide to not get too complex. These features were specific to IT and as such were outside our scope of knowledge with no background in IT. With such specific requirements they did not fit within the scope of a centralized CMDB and should instead be an IT specific tool.

We interviewed one person from governance, working with managing risks and laws, such as GDPR. Keeping track of these risks and to trace where these might occur appeared to be the main advantage of a CMDB. Then needing to track assets from all over the company to know how or when something might be compromised, and to know who is the owner over what system. To rely on that this database is always up to date and that the information within it is trustworthy is important and it would therefore be difficult to ensure this with an automated discovery system which could be a safety risk if not done correctly.

One project manager was interviewed. The points brought up within this interview were similar in many ways to the one from the software developers and in some ways similar to governance. There was a need for a database tracking what projects exist, if and how they depend on each other. Knowing that if a certain function is completed in one project, that another project depending on it could proceed. There was also a need to know when personal information was used, to then tag that project that it contains personal information making it easier to comply with GDPR.

The resulting requirements found during the interviews are divided by the four departments, displaying what use-cases each department has from a CMDB and what they need to track.

Software Developers:
CI

- Systems
- External packages
- Servers
- Clouds
- Tests

CSA

- Dependencies between a system and an imported package
- If the packages are trustworthy or have been compromised
- If the packages are locally stored within Axis
- Dependencies between systems
- If the system contain any known bugs
- Which tests are related to which systems

- Description of the deploy process
- On which server the system is deployed
- What configuration does a server have
- Be able to find a System based on keywords

Governance:
CI

- Employees

CSA

- Be able to track risks and laws that are related to CIs
- Track hardware to software
- Who owns a certain CI

IT:
CI

- Licences
- Contracts
- Computers
- Servers
- Virtual servers
- Employees

CSA

- Who owns a certain CI
- Notification when a licence/contract or contract is about to expire
- What the server configurations are

Database

- Role based access control to assure data quality
- Central database throughout Axis

Project Managers:
CI

- Projects

CSA

- Be able to track risks and laws that are related to CIs
- Dependencies between projects

4.1.3 Analysis of the old database

Axis has an old database called "CMDB" where the original plan was to compare the old requirements that this CMDB tried to solve with the new requirements that appeared throughout the interviews. The old requirements would then support the new requirement specification by either being in both specifications or by providing new requirements.

While examining the database, we looked at the log file to see which queries that were most commonly used. We noticed that the database only received posting queries, which inserted data into the database multiple times automatically every morning. No queries were used to retrieve data from the database, meaning that no one actually used the database. This made it apparent that the database did not fulfill the purpose it was made for. Therefore we had to examine the source code and documentation to understand which requirements and use-cases it tried to solve. It became clear that it only tried to solve one use-case which was to track licences and contracts that were connected to different servers and or other hardware to notify when they were to expire. The database was not a centralised database within Axis, which was one of the requirements discovered from IT during the interviews.

However the database did fulfill one use-case that is important for Axis and had multiple requirements to ensure data quality. The first requirement is that it should have the complete history of all licences and not delete them from the database. This requirement is important for the new CMDB also to assure CSA, since then we have to keep track of the history. Another requirement was that it supported all kinds of relationships between the CIs and that is something that the new CMDB also needs to support. It kept track of who and when someone updated a CI which helped to assure that the data quality were kept high.

By talking to the person that had used the database it became apparent that the use-case was still important and should definitely be considered in the new CMDB. The reason that no one used the database is that no one maintained it and that it needed further work. Even though this is a problem, they had not implemented a flexible structure to the database. A CMDB should be able to flexibly change and adapt according to what is needed and not require a developer to manually change the structure of a CI. Therefore if the database would have a more flexible structure, maybe the database would be more useful and in return easier for the developers to maintain.

The requirements found during the examination of the old database were the following.

Database requirements

- Ensure data quality

CI

- Licences/Contracts
- Servers

CSA

- Notification when a licence/contract or contract is about to expire
- Version history

4.1.4 Resulting Axis CMDB requirement specification

The final CMDB requirement specification is a combination of a literature study, interviews and examination of the old database. General requirements on the database are taken from the literature where some of the requirements are also established through the interviews or examination of the old database. The general requirements on the database are:

- Access control
- Data is correct
- Data is up to date
- Performance
- Be able to change every aspect of a CI
- Synchronize with external sources
- Possibility of different kinds of relationships
- Central database throughout Axis

The CMDB stores CIs and their metadata and it is important to store them with an appropriate granularity. The resulting CIs for Axis were derived from the interviews and the old database to ensure that all CIs within the requirement specification were specific for Axis. These CIs are:

- Systems
- Clouds
- Packages
- Tests
- Licenses
- Contracts
- Servers
- Virtual servers
- Computers
- Projects
- Employees

Apart from storing CIs a CMDB should also provide CSA. To provide CSA that is appropriate for Axis it should fulfill the following requirements to satisfy all interviewed departments.

- Provide relationships
- Store version history of all CI
- Effective CI naming
- Dependencies between a system and an imported package
- If the packages are trustworthy or have been compromised
- If the packages are locally stored within Axis
- Dependencies between systems
- If the system contain any known bugs
- Which tests are related to which systems
- Description of the deploy process
- On which server the system is deployed
- What the server configurations are
- Be able to find a System based on keywords
- Be able to track risks and laws that are related to CIs
- Track hardware to software
- Who owns a certain CI
- Notification when a licence/contract or contract is about to expire
- Which project depends on another project

4.2 Analysis and resulting MVP

The result from phase one is a complete CMDB requirement specification that will be converted into a database in phase two which is benchmarked. However, to perform the benchmarking there is no need for a complete CMDB requirement specification and therefore a minimum viable product was made. The MVP was made to save time and complexity in the second phase, implementing as little as possible while still being able to benchmark in a CMDB context. This was achieved by removing unnecessary redundancy while keeping the coverage of as many requirements as possible. The benchmarking results in sufficient data for answering “Should a CMDB be implemented as a graph database or relational database?”.

4.2.1 Analysis of MVP

As can be seen in the CMDB requirement specification there is a need to store CIs with a suitable granularity. It would be redundant to store multiple similar CIs for a MVP and therefore it was decided to remove any redundant CIs. E.g IT wanted to have a complete asset management database that keeps track of all relevant hardware such as servers and computers among others. However this is excessive since it is sufficient to just keep track of servers since there is not much difference between a server and other hardware for our purposes. The goal is to show that the database can keep track of hardware and store its metadata. The server CI also covers similar needs as a virtual server and clouds, it was then also deemed sufficient to cover these with a server CI. The servers will also be connected to licences to be able to supply the requirement of notifying users when something is about to expire. Licences may be excessive, but we believe that it is good for showing that this database could be a complete CMDB. We also believe that this requirement was the most important requirement in the old database and therefore we kept it in this MVP.

Project managers and software developers requirements are quite similar in regards to their CIs structure and what they want to accomplish with their CIs. Both require some kind of impact analysis, tracking the impact of dependencies. Other than that they require different CIs with different meta data which would be redundant to showcase, not contributing anything original. Since our focus in this thesis is on software developers we decided to remove the project managers requirements to reduce the redundancy in the MVP.

The requirement of synchronizing with external sources is not in the MVP since it is not part of the benchmarking. The reason is that we want to benchmark two types of databases and synchronizing with an external database would not be handled by the database language. Instead it would most likely be handled by an external tool, and that is not part of any of the three research questions and was therefore avoided.

Potentially we could store a lot of attributes for each CI, however we saw that as redundant and decided to keep the most important ones that could resemble a real scenario, so that it would still be a proof-of-work. It was important to have enough attributes to accurately resemble each CI but not more to make it unnecessarily complex.

It was important for us to supply all kinds of relationships to show that the databases support one-to-one, one-to-many, and many-to-many relationships. Therefore we had to keep relevant CIs and relationships that could showcase each kind of relationship at least once.

One requirement is to restrict or track system access to showcase data quality. We decided to not accomplish this since we saw multiple ways of doing this and it would not benefit the benchmarking in any way except showcasing a more real-life CMDB. And since we are not the best developers and did not know the way that Axis would want it implemented, we would need to research how it should be implemented. Instead we decided to skip this part and spend time on other more important parts that would affect our benchmarking.

4.2.2 Resulting MVP

The final MVP requirement specification that cover all aspects to be benchmarked while keeping it simple is displayed in bullet points below:

CI

- Systems
- External Systems (Packages)
- Employees
- Servers
- Licences

CSA

- Notification when a licence is about to expire
- Dependencies between systems
- Dependencies between a system and an imported package
- On which server the system is deployed
- What the server configurations are
- Who owns a certain CI
- Be able to find a System based on keywords

Database

- A central point in Axis
- Dynamic/Adaptable structure
- Possibility of different kinds of relationships

These requirements will be used in the next phase when deciding which queries to benchmark and the structure of the database. An UML of the MVP can be found in appendix B.

4.3 Analysis and results of benchmarking

To reach a conclusion regarding research question three, "Should a CMDB be implemented as a graph database or relational database?". The results from the benchmarking needs to be analyzed and discussed. The chapter starts with establishing the parameters from which the benchmark will be compared on and which were not used but could be considered. The two datasets used will be described and their size will be motivated. The queries used for the benchmarking are discussed and motivated. The queries have to match the requirements that Axis has on its CMDB and they should resemble as much of a real life scenario as possible. This to be a realistic proof-of-work that can be further developed by Axis. Then each query is analysed based on the decided parameters from which a conclusion will be drawn.

4.3.1 Benchmarking parameters

There are many parameters that could be considered while benchmarking the database depending on what aspects are important for the context. The performance of a database is a common parameter to examine, being used in similar works[23][9]. The parameter was also requested by Axis, considering it to be an interesting factor to compare for both this context, but also in similar future projects. It could however be argued to what extent it is important when choosing a database type in a CMDB context. The time difference might be large in comparative difference, one query taking one millisecond and another taking 10 milliseconds. While this is a large percentage difference it is essential to take in consideration how important this time difference is as well. According to Axis, a query is only slow if you noticeable. However if a query is run by hundreds of employees at Axis the smaller numbers would become more relevant than if a query is run once. The performance could also be seen as a measure of how heavy loads the database can handle, by processing many queries at the same time or in fast succession. How the performance scales with the size of the database is also important to consider. When a company grows and more relevant CIs are discovered the CMDB will inevitably grow as well, it is therefore important to consider which impact this change will have on the database.

When choosing a database type for a project there is more than performance that is worth considering. A developer also has to think about the maintainability of the system, meaning how easy it is to learn and write the code and how easy it is to read the code[9]. This is also something Axis considers important, since they want their systems to be maintained for a long time. Writability, meaning how easy and intuitive it is to write specific queries for the respective languages. Readability, how easy it is for someone with appropriate background knowledge to read and understand what a query does. These parameters are important to consider before dedicating oneself to a database partly to know how big an effort it would be to implement the database but mainly how easy it would be to maintain. One database could be slightly faster than another but perhaps that does not count for much if it is a major effort to maintain and update it, which is important for longevity of a database. Both of these parameters could potentially be measured with token analysis[23], but within our limited time and limited knowledge in code analysis we determined this to be too complex. Therefore we chose to base these parameters on our subjective opinion of the two languages after working with them during this thesis, deeming this to be our best option.

Since a CMDB grows day by day it could become quite large in regards to data storage space. It could therefore be beneficial to look at how much data the database type can store or manage. However storage is cheap today and it is not a priority for Axis that the database takes up little memory. To fill up the database enough for it to be unable to perform queries could potentially be a problem. In this context it is not considered a problem since the data does not reach the size where it starts to become a problem, which would be larger data sets, like big data.

The resulting benchmarking parameters are the following:

- Performance
- Writability
- Readability

4.3.2 Benchmark UML and Dataset

To represent the MVP in the databases a structure was designed with the defined CIs and relations, that could fulfill the use-cases of the MVP. An UML of the decided structure can be seen in appendix B. This UML was then converted into a more specific UML that matched MariaDB where intermediate tables were inserted to support many-to-many relationships. Neo4j did not need any modification to the UML, since its structure is not predefined. Each node is its own datapoint with direct relation to other nodes with edges.

An exact size of the dataset used for filling the databases would also be useful for knowing the context for further exploring with different sizes. The small and large dataset used in the benchmarking are presented in table 4.1. Where the small dataset symbolizes the size of Axis and the big dataset symbolizes a larger company that Axis might grow into in the future. The datasets consist of a single version of each CI. However if multiple versions of each CI would be considered, even the large dataset might not be that large. E.g a system might have multiple updates a day, and could easily grow to be far bigger than 2500. These two datasets do show how the databases would scale with the growth of data points and will be sufficient to do a benchmarking on. An even bigger data set would become time consuming to perform benchmarking on since it took up to 7 hours to fill up the database with the large dataset. Below the Licences in the table are the relations that exists within the MVP CMDB. Most of these numbers are odd which is a result of them being generated based on random amount of relationships between CIs.

	Small dataset	Large dataset	Difference (%)
Employees	3000	100000	3333
External systems	1000	10000	1000
Systems	250	2500	1000
Servers	1000	50000	5000
Licences	1000	50000	5000
Dependencies	623	6233	1000
Hosts	1993	99938	5000
IS Owners	621	6257	1000
Server Owners	1000	50000	5000
Server Licences	1000	50000	5000
Packages	3546	35330	1000

Table 4.1: Amount of CIs for the two datasets

4.3.3 Queries

Eight different queries were chosen covering all the use-case derived from the requirements of the MVP. The queries were designed in such a way to cover all use-cases of the MVP.

Most use-cases could be reformulated into covering more requirements from the complete requirement specification from chapter 4.1.4. The code for each query in their respective language can be found in appendix A, a short description for each query will instead be

given in this chapter.

The first query is to add an attribute to an existing CI. E.g a server does have a name, but now you also want to store a server ID. This symbolizes dynamic/adaptable structure in the database, which is a requirement from the MVP, by showcasing how difficult it is to insert new attributes to a CI. This query could be made in two different ways, either we make it possible for all new data points to have the new attribute, but the old ones are not updated or we give all old data points a default value to the attribute. We chose to give all data points a default value, since Neo4j would not require any update to be made if we do not update all nodes with a default value. With the community edition used for Neo4j there is no way to constraint each node of the same type to have the same attributes, instead having to go through each node individually. It is worth noting that this query could potentially be very different with the enterprise edition. Looking back it could be argued that this query do not say a lot in regards of performance since a default value on all CIs do not say anything specific. However it does showcase the flexibility of the databases.

The second query is to modify existing relationships. This query symbolizes a dynamic and adaptable structure in the database to showcase that it is possible to redirect relationships. The use-case that we decided to showcase was if an employee quit their job and is owner of a system, how can we find their ownerships and give the ownerships to someone else. This query could be easily reformulated to a package that is updated and all systems that depend on this package should point to the new version.

The third query is to fetch all CIs that are affected by a certain CI. This use-case symbolizes a relevant query for developers where they can see if one package has a security threat which systems might be affected by this. This query both showcases a relation between one package and systems, but also between systems and systems to support traceability. It could easily be reformulated to project managers that want to do the same thing, but for their projects.

The fourth query is to fetch something by a string instead of by id. This query showcases usability by fetching data by a substring in a longer description text. This was done to test both languages string matching. The query could be made for any CI that has a description or string of any type.

The fifth query is to fetch everything that is directly connected to a certain CI. This symbolizes CSA where you can get the complete status of a certain system, which server it is on, which licence the server has, who is the owner etc. This query could include as many CIs as would be reasonable, however we decided to include everything that is directly connected.

The sixth query is to fetch by date, which symbolizes the requirement on notifying the users when something is about to expire. Fetch data by date could be relevant in other scenarios, but this query is a bit harder to reformulate into a different scenario. It was chosen because it was the specific use-case that the old CMDB tried to solve.

The seventh query was to insert a new CI with the possibility of relationships. This query symbolizes a dynamic/adaptable structure by having the ability to create a new CI on demand instead of having a developer inserting it manually into the database. The use-case that we decided to solve was to create a project with relations to a system, however it could be any CI that was created and inserted with relations.

The eighth query is to insert a CI of an already existing CI type with multiple relations. This query symbolizes that the databases should be able to handle inserts of new data, and it is important that it is easy to insert into the database. The use-case that is showcased is to

insert a system, with dependencies and owners.

Other queries that were initially considered but not selected were queries testing the possibility of mass insertion and finding the shortest path between two data points. Mass insert could be seen as when the company acquired a large number of new resources simultaneously, like if they merge with another company. This query was ultimately chosen to be avoided since this would likely be a very rare occurrence, and because of it being so rare it would not make a large difference how long it would take, or how hard it would be to implement as long as it would be possible. Instead we decided to insert single items, since we believed it was a more relevant query. We initially wanted to try creating a query for finding the shortest path between two data points since this was a feature we knew that Neo4j was good at, however we did not find any specific use for this query and therefore decided against using it.

We believe that these eight selected queries represent all important queries that could be made to a CMDB. Query 1, 2, 7, and 8 symbolizes a dynamic structure of the CMDB where 1 and 2 updates an existing CI and 7 and 8 inserts new CIs. These queries are required for the CMDB to be able to adapt when new requirements or needs occur. The two updates add attributes and redirects relationships to be able to provide an adaptable CSA that might change with the growth of the CMDB. The two insert queries either insert an existing CI type or create a new CI type and insert it. These two shows that it is possible for the CMDB to grow in two ways. Both in adding more of the same CI types and adding new CIs when needed. These four queries symbolizes the possibility to adapt the database with the needs, however to ensure that the database is usable we have to fetch data and provide CSA. Therefore query 3-6 illustrates different fetch methods that symbolizes different use-cases from the interviews. These queries are more important than the other queries since they are the majority of the requests since most people will want to fetch data and not insert new data. The four queries vary from a simple fetch, that fetches data by date or string, to a recursive call that illustrates impact analysis. CSA requires the CMDB to provide the status which means to show relationships and attributes and therefore we provide four different kinds.

4.3.4 Analysis of performance

Defining the setup used when performing the benchmarking would be useful for recreating the tests or could be used as a base of comparison if future work would be done with a different setup. The hardware and software specifications used heavily impact the execution times from the benchmarks. The used setup is a laptop, that might not provide accurate execution times when compared to a real server. The specification used for all tests was the following:

- CPU: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz x 4
- Memory: 8 GiB
- GPU: Intel® HD Graphics 520 (Skylake GT2)
- OS: Debian GNU/Linux 10 (Buster) 64 bit
- Computer name: HP elitebook 840 G3

The eighth queries were analysed in performance by executing them on a small dataset as well as a large dataset. After the queries were run and post processed as described in the method chapter, the average execution time as well as the diverging longest and shortest times were calculated and saved. A table comparing the execution time of every aspect was compiled as can be seen in table 4.2. The difference in execution time, displayed in percentage, between the small and large dataset are also displayed to compare how the database scales for each query.

Query	Language	Small time(ms)	Large time(ms)	Difference (%)
1	MariaDB	4.9	7.3	149
	Neo4j	179.1	6495.8	3627
2	MariaDB	0.6	0.7	117
	Neo4j	9.9	35.2	456
3	MariaDB	6.3	57.2	908
	Neo4j	7.4	32.0	432
4	MariaDB	0.6	2.8	350
	Neo4j	1.9	6.0	316
5	MariaDB	12.7	69.3	546
	Neo4j	39.7	196.0	494
6	MariaDB	7.6	314.7	4145
	Neo4j	28.3	1255.9	4438
7	MariaDB	45.9	44.9	98
	Neo4j	5.5	7.0	127
8	MariaDB	0.7	1.5	214
	Neo4j	14.0	47.7	340

Table 4.2: Execution times for all queries

These values are also presented in the form of bar graphs comparing the execution time difference between the databases for each query, the bar displaying the average time and the error bars indicating maximum and minimum times.

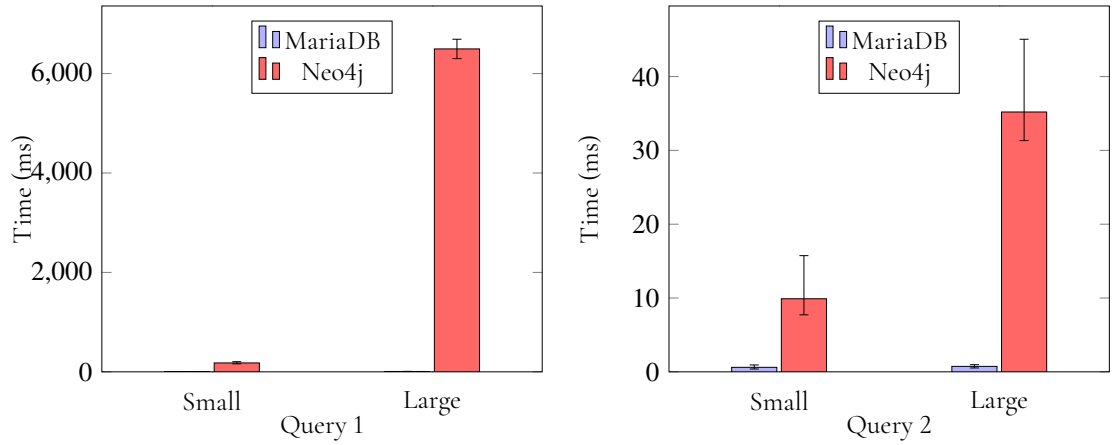


Table 4.3: Execution times with error bars for query 1 and 2

Query 1 and 2 both update attributes in existing data points, not creating new ones. Query 1 adds a new attribute to all existing CIs of a specific CI type and Query 2 updates all systems from one owner to another. A more detailed description can be found in subchapter 4.3.3. As can be seen in table 4.3 Neo4j is far slower for both queries for both data sizes where Query 1 takes over 6 sec for the big data set which is 36 times more than for the small dataset. This would mean that this query would become impractical with the growth of the database, but with Axis current size it is still usable. However, as argued before, this query might not be that useful when choosing database type for a CMDB context. With the growth of the database MariaDB would be the more safe choice for these queries, in regards to performance, since it has a lower execution time and better scaling.

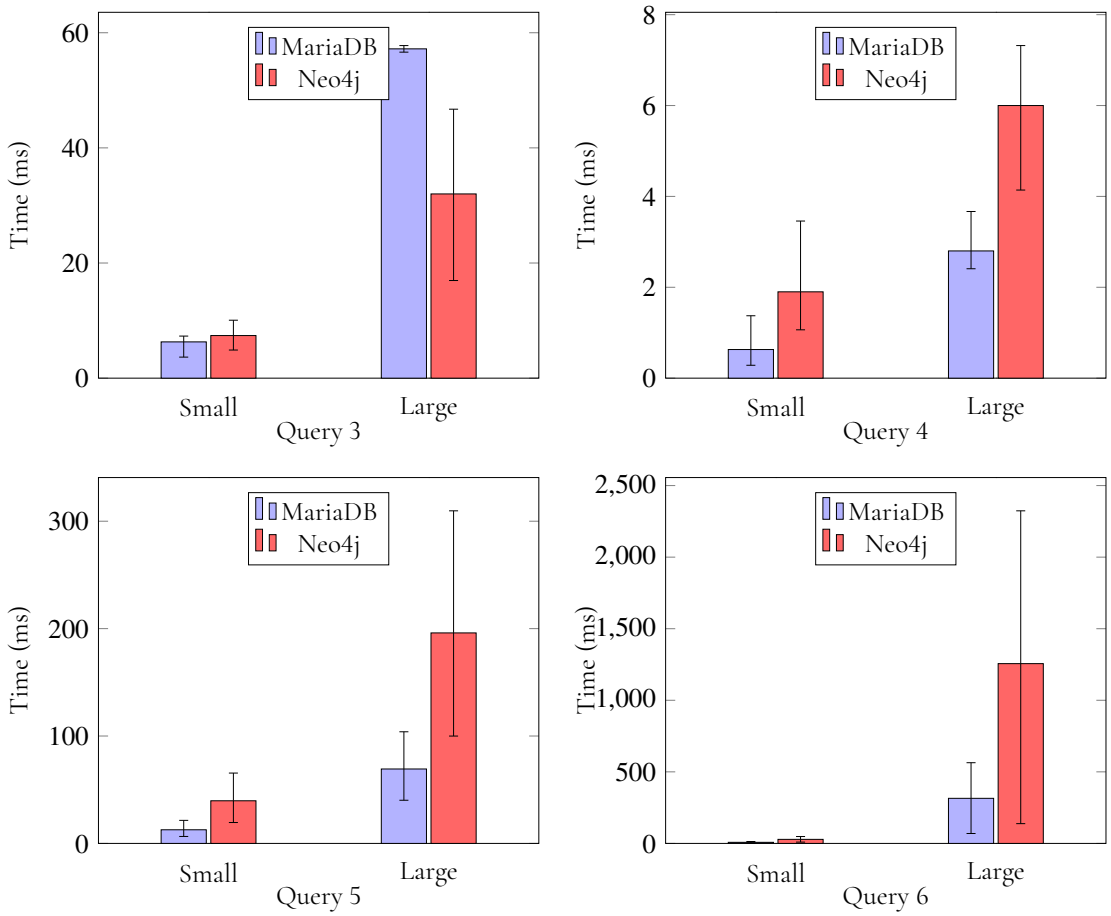


Table 4.4: Execution times with error bars for query 3, 4, 5, and 6

Query 3, 4, 5 and 6 are all queries that fetch information about CIs not changing any existing attributes. Query 4 and 6 are simple queries that fetch data by a string and date respectively. Both of these queries almost scale the same, however the execution time is higher for Neo4j as can be seen in table 4.2. Query 6 could even be argued for being too slow with the big data set. It is worth noting that there is a huge diverging max and min for the large set of query 6, as can be seen in table 4.4, for both languages where Neo4js fastest execution time is close to MariaDBs fastest execution time. This can also be seen in query 5 where Neo4js fastest execution time is faster than MariaDBs slowest execution time. We have not found out why this inconsistency occurs, but it could be because Neo4j requires more memory. This would mean that if Neo4j could run with its fastest execution times, the differences would be reduced significantly.

Query 3 is a recursive function that fetches all systems that have a relation with each other. As can be seen in table 4.2, MariaDB is faster with the small dataset but slower with the large dataset. This is because Neo4j scales twice as good, which means that this difference would diverge even more with the growth of the database. However both databases do perform this query within reasonable time and therefore it should not make a difference when choosing database type. Query 5 joins multiple tables to get an overview of the CIs. Both databases scale similarly, but Neo4j is quite a bit slower as can be seen in table 4.2. 196 ms for the big

dataset could become troublesome when multiple requests are made at the same time, and if the database grew a bit more it could become impractical.

In the choice of database type, it does not matter when looking at the small datasets. However if the database grows, Neo4j would become impractical first since both query 5 and 6 is slow with the big dataset, where only MariaDB is slow for query 6. Therefore MariaDB seems to be the superior choice when fetching data.

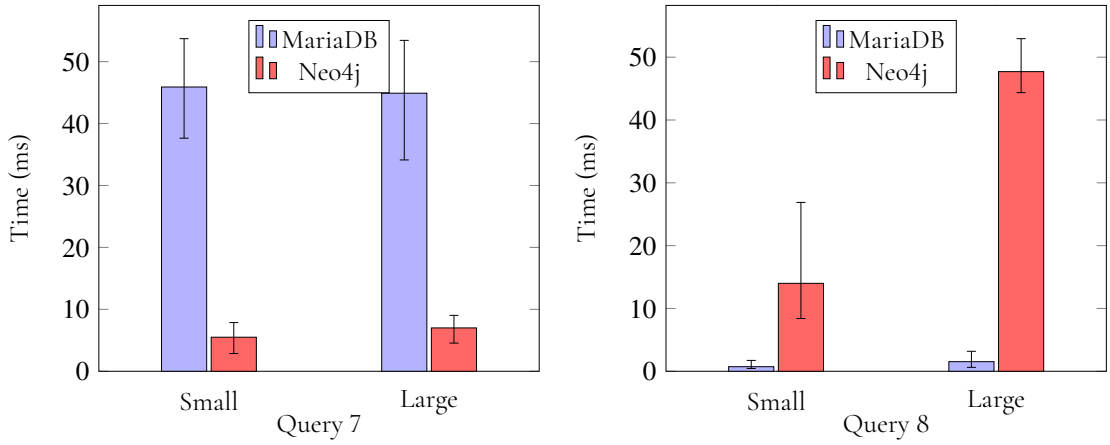


Table 4.5: Execution times with error bars for query 7 and 8

Query 7 and 8 are both queries that post new CI to the database. Query 7 creates a whole new CI type, essentially creating two new tables in SQL for the CI and relationships towards another CI, but only creating a node with relationships in Neo4j. Both languages take almost as long executing for the small dataset as they do for the large, which is expected for this query since it is not affected by the existing data points. Neo4j is faster than MariaDB for this query as can be seen in table 4.5. The reason being that Neo4j only has to create a node as it would for any existing CI type. This is one of the advantages of Neo4j, showcasing its flexibility. Query 8 creates a new CI of an already existing CI type with many relationships. In table 4.2 it can be seen that MariaDB outperforms Neo4j in time as well as scaling in this aspect. When choosing database type for a CMDB, the performance of insert queries should not be the main concern since both queries are quick and do not scale significantly.

4.3.5 Analysis of maintainability

We divided maintainability into two parts, the first part is to be able to write or modify code while the second is to be able to read and understand old code. Each part is important while developing and maintaining code. This analysis is made subjectively by us where our perception of how it was to write and read the code. Our subjective results can be seen in table 4.6. The more difficult queries will also be looked at from how hard it was to find information online and how many lines of code it took to write the queries.

First of all, query 4, 6, and 8 were simple queries, as can be seen in table 4.6, since they only consisted of basic syntax and took a few lines to write. These queries were easy to implement in both Cypher and SQL and there were a lot of similar examples of these in

online documentation. The reason these queries were easy were because they only involved one data point.

The first query had to alter the table in SQL to have one more attribute and give every existing data point a default value. In Cypher it is not required to alter any tables, however we have to set an individual value on every node. These two queries are quite similar and take three and two lines respectively to write and consist of quite basic syntax. Both had easy to access documentation and were both easy in regards to readability.

The second query actually required two separate queries in both SQL and Cypher where we first update all relationships and secondly remove any duplicate relationships. To do this SQL uses a simple update and a delete while Cypher requires the use an external package called APOC. This is the reason that Cypher got a lower score in table 4.6 in regards to both writability and readability. APOC is officially supported by Neo4j, however it has to installed separately into Neo4j. This created difficulties and made it harder to use and understand.

The third query also requires the external package APOC, and it was not clear to us why this was the case. The reason was that if the same query were made without APOC it resulted in an infinity loop because the calls are made recursively and our data has circular recursion. Because this was not immediately apparent why Neo4j acted like this and instead needed APOC to complete the query it was quite difficult to write. However the readability was understandable since the query took four lines of code and there were not any abnormalities except APOC. On the other hand, SQL was tricky to write since we had to use a method called “With Recursive” that was not easy to understand how it worked even with documentation. After writing the query it would be hard to understand what it does since it is a single query in 18 lines that combines a lot of different SQL concepts. This query in SQL was the most difficult query to both write and read out of all the queries.

The fifth query was quite easy in both languages where the major difference is that SQL has to define which tables it wants to combine, and it has to combine all intermediate tables that are between two CI tables. However while Cypher could perform the query in just four lines it took MariaDB 15 lines of code. Important to note is that SQL only requires join statements, just a lot of them. Therefore, writing the query was approximately as easy in both languages, however reading the query was a bit easier in Cypher.

The seventh query was more difficult to write in SQL than Cypher. The main reason being that there are more steps that needs to be done. Neo4j does not use tables like MariaDB, which means that they can insert a new CI however they want, while MariaDB does have to create a new table with corresponding attributes. It is also required to create a table between the new CI and to the other CIs, since it has to be able to have many-to-many relationships. It is not a requirement to have an intermediate table, like we decided to implement, however to increase the flexibility we thought it would be the best to have the ability to have many-to-many relationships from the beginning even if that is not required. Cypher does this query in three lines, while MariaDB requires four independent queries where two creates the tables and two inserts the data. However, reading the queries afterwards both queries are easy to read and understand since they only use simple statements within both languages. The reading is a bit easier in Cypher only because the code is shorter and therefore there is less to understand.

Another factor to consider is the size of the community and available documentation for the query languages. SQL being an older and more mature language currently has a larger community than Cypher. We noticed this when searching for documentation, it was often

easier to find documentation from different sources with different examples for SQL. For Neo4j we had to depend on Neo4js official documentation.

Important to know is that while evaluating the readability and writability we did not have any previous experience with Cypher and Neo4j before this thesis. And therefore since we do believe that almost all queries are as easy or easier to write it would probably indicate that Cypher might be slightly easier than SQL when implementing a database for a CMDB context.

Query	Language	Writability	Readability
1	MariaDB	5/5	5/5
	Neo4j	5/5	5/5
2	MariaDB	4/5	4/5
	Neo4j	3/5	3/5
3	MariaDB	1/5	1/5
	Neo4j	4/5	4/5
4	MariaDB	5/5	5/5
	Neo4j	5/5	5/5
5	MariaDB	4/5	3/5
	Neo4j	4/5	4/5
6	MariaDB	5/5	5/5
	Neo4j	5/5	5/5
7	MariaDB	4/5	3/5
	Neo4j	5/5	5/5
8	MariaDB	5/5	5/5
	Neo4j	5/5	5/5

Table 4.6: Subjective result of writability and maintainability, where a higher value indicating that the query is easier for the given parameter

4.3.6 Results of benchmarking

The benchmarking is based on the three aspects performance, writability, and readability where the two latter is more important than the performance. The performance is only important if it is noticeable when querying the database while writability and readability explains how the maintainability of the database would be. By having a database that is easy to maintain, we ensure that the database will be useful for a long time.

From a performance perspective MariaDB is the better choice in all queries except for query 3 and 7. In both those cases the two languages still have a reasonable execution time that would not be noticed when querying the database. However from a CMDB perspective query 3-6 will be used the most since all those fetch data and everyone is able to fetch data, but not modify it like query 1,2,7, and 8 does. Neo4j also had two queries, 1 and 6, which took over 1 sec to execute. These execution times are barely acceptable and could potentially grow

to be a problem if the database grew larger. However for the current size of Axis it would take a long time for this to become a noticeable problem.

Writing the queries was easier in Neo4j than in MariaDB even though we did not have any prior knowledge within Cypher. It can be seen in table 4.6 that Cypher got a better writability score than every SQL query except for query 2. This is a clear indication that we believed that it was easier. Another indication was that query 1 was reformulated to ensure that code had to be written in both SQL and Cypher. The reason being that Neo4j is more flexible in its structure and does not have to modify tables since each node can have any attributes or relationships even though the node is of the same type.

The readability was based on our subjective opinion reading the queries while also considering the amount of lines required, where a query with remarkably fewer lines more often was easier to read. As can be seen in table 4.6 the readability is easier in Cypher for each query except for query 2. This clearly indicates that Cypher is easier to read in most cases.

Since the performance was not the most important aspect when choosing a database type we have come to the conclusion that Neo4j is the better choice over MariaDB when implementing a CMDB. It would be easier to maintain while having an acceptable performance.

Chapter 5
Discussion and related work

A final discussion is held in this chapter to reflect on the choices made during this thesis and the conclusions reached. The purpose of the chapter is to bring up and discuss factors that would impact further research within this field. Discussing and reflecting on each part of the thesis of what went well and what could have been done differently, bringing up pitfalls that could have been avoided or circumvented. Threats to validity will be discussed. How they could have affected our outcome and to what extent. Four related works are summarized and discussed. Creating a more complete picture of the subject by discussing how our works build upon each other for a greater understanding of the subject. Lastly some ideas for future work are presented which could further solidify the results.

5.1 Discussion and reflection

The two first research questions, "what are the necessary configuration items for Axis" and "what would the CMDB requirement specification look like for Axis", were answered in the first phase which consisted of a literature study, interviews, and analyzing the old database. The most important part was that the results would be unique and fit Axis specific context. This was achieved by first doing a literature study that gave us some general knowledge of CMDB requirements, then the CMDB requirement specification was formed by the more specific knowledge from the interviews. The interviews gave us a lot of information about the needs Axis has from a CM perspective and we then tried to implement these requirements into the CMDB requirement specification. Without the interviews the CMDB would not have been specific to Axis and therefore would not have answered the research questions that were asked. However our results would be relevant for other companies to base their research on, where our requirement specification could serve as an example and inspiration. While Axis has a unique context it is not completely unlike others, and would have requirements in common with other contexts.

A problem we faced in the beginning was that the interviews took longer to perform than

expected. We conducted eight interviews within four different departments where most of the interviewees were with key peoples in their respective department. This was done to try to get as good answers as possible while reducing the time spent on interviews. The interviews took about four weeks, because of peoples busy schedules. In the end we believe that we received good and relevant answers, however we noticed through the interviews with system developers where we interviewed a junior developer that they had another perspective than the senior developers had. This leads us to believe that it would have affected the end results if we were to interview junior personnel within every department that we interviewed, but it was not something we had time for within our limited time frame.

The old database called "CMDB" was not as relevant as we first believed that it would be. Our initial goal was to form a CMDB requirement specification for Axis and compare this to the old database. However when we examined the database we noticed that this was not possible since it was not an central database within Axis. Therefore it differed too much and we instead tried to see if the database would fit as a part of our CMDB requirement specification. In the end we believe that we focused too much on this database and that it did not provide as much new insight as the time spent on examining it.

There were two purposes for creating the MVP, where the most important was to save time and resources, not needing a complete CMDB for the benchmarking. This was assured by creating a MVP where all requirements were covered, removing redundancies. The other purpose of the MVP was to create a proof-of-work for Axis. We believe that our MVP symbolizes the complete CMDB requirement specification. It contained relevant CIs and covering all use-cases either directly or by similar use-cases and CIs. The MVP showcases a small CMDB, and it would be easy to build upon it if wanted. The MVP saved us a lot of time, by not having to implement and insert more data than needed while still providing good results for our benchmarking.

The parameters on which the benchmarking was judged were decided from the suggestions made by Axis as well as related works. The performance parameters were frequently seen in the related works and Axis also deemed it to be important. Axis also had a need of judging the maintainability of the languages. This was not a clearly defined parameter but from reading related works who base their benchmarking on terms such as complexity, readability and maintainability. We combined the knowledge and decided to base our definition of maintainability as a combination of readability and writability. As mentioned earlier, there were other parameters that were discussed if they should be used. These are related to the data size, which Axis did not consider a priority.

Regarding the third research question, "Should a CMDB be implemented with a graph database or relational database?", we did not reach as clear a conclusion as we had initially hoped. Determining that there are certain pros and cons for both languages not making any of the languages clearly superior. Regarding their performance Neo4j was generally slightly slower in most scenarios, but not to an extent that we considered making it unusable or inefficient for implementing a CMDB, except for query 1 and 6. However like we argued earlier query 1 might have been a redundant query looking back and it could be argued how relevant it is. Query 6 however displayed that Neo4j might have problems regarding scaling, where it in the larger dataset reached a time of over 2 seconds. The actual impact of having a response time of two seconds would depend on what kind of database it is. A database which experiences heavy traffic, getting requests multiple times a second, and where it is critical to retrieve the information fast, 2 seconds would make such a database unusable. A CMDB

which is used by a limited number of users within a company would not experience as heavy loads to the same extent. Therefore the impact of a slower query would not affect a CMDB in the same way as other databases. Regarding maintainability the languages were also quite similar, slightly preferring Neo4j for most queries.

We had some unexpected issues with Neo4j that we did not consider when starting out, none of which affected our results but were hindrances. One problem was the exceptionally long time it took to fill the large database, to prepare the database for the benchmarks. This took around 7 hours for our large dataset, and would take exceptionally longer for an even larger dataset. As mentioned earlier it is possible that there are more efficient methods for filling the database that we decided to not examine that could have made the benchmarking process faster. We did however not care about this since filling the database is something that would only happen once. Neo4j was also harder to configure than expected. Having problems with the memory getting full with temporary files and log files causing the computer to behave strangely and Neo4j to stop functioning properly. This was resolved by changing parameters in the configuration file, limiting how large these files were allowed to grow. It could however be argued that this would not be a noteworthy problem if Neo4j would be run on a real server with more memory. Having to download and install the official package APOC from a github repository for using certain functionality in Neo4j was slightly confusing. Where it was unclear why it was not directly included in Neo4j.

5.2 Threats to validity

The goal was to create a CMDB requirement specification for Axis, and we did this by interviewing eight people. These eight people have to symbolize all of Axis which is impossible to completely do with this few people. If more people were to be interviewed the foundation would remain the same however there would possibly be some more specific CIs and requirements added, that would most likely be similar to the current ones. This would especially be the case if we interviewed people with a wider variety of work experience or included more departments. Because of the time frame we decided to focus on software developers, because of our background and since our supervisor at Axis is a software developer, deciding to interview four people from this department. Therefore we are more certain that the software developers requirements are covered by the requirement specification than the other departments. We also believe that since we interviewed key people within the other departments, we did get the most important requirements that symbolizes most of each department's need for a CMDB.

By creating a MVP we reduced the requirement specification to a minimum while still illustrating as many requirements as possible. By doing this it is inevitable that we removed parts of the CMDB that could potentially provide a different result. E.g we are filling it with less data than it otherwise would have, which could affect the results. However the queries that are made would not have been very different than they are in the MVP since we would still want to perform the same kind of queries. Maybe they would include different types of data to illustrate the same query, or just fetch data from different data points. Therefore the results would still be similar since it is important to look at the scaling to understand how the databases perform with a rising amount of data.

When implementing the CMDB in both MariaDB and Neo4j we had no prior experi-

ence with Neo4j and had some experience with MariaDB, having read a course about SQL databases. This could mean that we have not implemented the Neo4j queries in the most efficient way. However we consulted our supervisor when needed, reading documentation and examples online to find the best and most efficient way to write the queries. This reduces the risk that the queries are inefficient and instead is implemented in the best way.

Benchmarking of the two parameters readability and writability was done by giving them a rating of our subjective opinion when working with the languages. This could most likely be done in a better and less subjective way if there were an exact science when comparing syntax for different programming languages. We did not find any reliable way of doing this and opted to simply look at the lines of code and rate our experiences. While this method was solely subjective it was not completely unreliable since it was made by two master students in computer science having worked with both languages for a longer period of time during this thesis.

The benchmarking was performed on a laptop instead of a real server, where the database would be placed in a realistic scenario. This is not something that we considered before the end of the tests and it is possible that the results would have been different. We believe that the resulting differences between the two languages and the scaling between the small and large dataset would be similar. This, because a server with better hardware than our laptop would most likely decrease all execution times with a constant factor. However if the execution times were reduced, it would result in the two database type being capable of handling more data before becoming unreasonably slow. These are just our theories however and it is something that would need to be examined further.

No one could know that the COVID-19 pandemic would occur which meant that we had to work from home during half of the thesis and most of the benchmarking. This resulted in some setbacks in regards to updating our software, since we had to be on Axis local network to be able to update. However this was a setback in regards to time, and does not affect our results. Another problem it provided was that we did not have the same possibility to contact our supervisor or other Axis employees. We could still contact them by means online, it was not however as natural as it would have been otherwise.

5.3 Related work

To look at related work is important to confront and compare our work. The conclusions of these works may strengthen our results, by showcasing that our results are not completely unique. However if our conclusions are different, it is discussed why this could be the case and if it affects our results in a negative way. Their work may provide a more complete picture of the subject by combining our results and conclusions. The works were also used as a basis for inspiration in the beginning of the thesis. Specifically have four papers been selected, their results and conclusions are analysed to relate to our own. The papers are selected to be relevant while covering different aspects of the thesis, focusing on our different research questions.

It is worth noting that the second paper “Graph-Datenbanken als Grundlage des Configuration Managements – Eine Untersuchung am Beispiel von Neo4J” was a paper written completely in german which we have very limited knowledge in. We deemed this paper to be very relevant to our thesis while reading the english abtract. We therefore decided to trans-

late the paper with the help of Google translate as well as our limited knowledge in german, reading it from a critical perspective. We are quite certain that this gave an accurate result since the resulting text was completely readable and understandable. Making sense when put in perspective to prior knowledge within the field.

5.3.1 Requirements and Recommendations for the Realization of a Configuration Management Database [22]

The IT infrastructure Library (ITIL) is a collection of best practices in IT. They provide five management processes, Incident Management, Problem Management, Change Management, Release Management, and Configuration Management. This paper has its focus on Configuration Management which goal is to provide information to the other processes. Therefore it keeps information in a database called Configuration management database (CMDB) which stores Configuration Items (CI), which is something that contributes to the deliverable, and the relations between different CIs. Currently the industry is far away from having a standardized one size fit all version of a CMDB since many companies have different needs. However this paper is a start to standardize requirements to either implement or look for a CMDB. To do this they divide the CMDB into two steps where the first step consists of what the requirements on a CMDB tool is and its underlying information model and the second step is how these requirements can be accomplished.

This paper provides both requirements on the information model and functional requirements on a CMDB tool. These requirements were obtained by analyzing literature from different sources to come up with unique general requirements. Examples of these are that the CMDB should be adaptable and the system has to have visualisation support. Thereafter they suggest realizations solutions such as the CMDB has to support relationship cardinalities. A more comprehensive catalog of recommendations are under development and more requirements will be provided.

This paper provides a general CMDB requirement specification that is not specifically made for any specific context. It is troublesome to create a generic CMDB requirement specification since each company has their specific needs that are based on how the company is built and what kinds of products they create. Our thesis builds on this literature where our literature study is similar to their CMDB requirement specification. In our study we try to gather as many general requirements as possible that we later can specify to fit Axis context. Therefore this related work was a useful part of our literature study, where most requirements were adapted to our thesis, to then be able to provide a more in depth and specific requirement specification. The biggest difference is that our examples of CIs and CSA has concrete examples instead of discussing everything in a more abstract context.

5.3.2 Graph-Datenbanken als Grundlage des Configuration Managements – Eine Untersuchung am Beispiel von Neo4J [21]

The paper is an examination of which database technology would be best suited for a CMDB by comparing Neo4j to Oracle 12c, specifically which performs more efficiently. These are two different database technologies, graph database and relational database respectively. They motivate this research by claiming that no prior research has tested the validity of a graph database for the purposes of CMDB. With their results they would contribute to future research for practitioners when implementing a CMDB by knowing the validity of this technology for such a database. Their results are based on literature study as well as benchmarking. Their research provides them with five general requirements for a CMDB, which are:

1. Support for CI identification
2. Visualization of parts of the CMDB
3. Component Failure Impact Analysis (CFIA)
4. Plausibility checks and support for audits
5. Integration with external databases and system management data stores

From these five requirements they chose to exclude three, only focusing on 2 and 3 claiming that the other three would have similar implications in both of the databases. From these requirements they form use cases which they use to derive database queries used for experimentation. This experimentation is done by creating two separate databases in Oracle 12c as well as Neo4j which would have a similar structure to a real CMDB, as well as creating three queries for both the technologies with the same purpose but with different complexity. By performing these experiments they reached the conclusion that relational databases are faster and more efficient than graph databases. They state that graph databases are an adequate start for implementation of a CMDB but would ultimately recommend relational databases.

This paper is quite similar to ours except that it does not define any specific context. The results reached from this paper would then be a good comparison for how the results would differ in a CMDB context. They however only based their work on three separate queries that were quite similar to each other. Our work can be seen as a future work of theirs. Taking their general requirements in consideration before forming more specific ones from our interviews, for a unique specification to our context. Where we ran our benchmarks on more varied queries that were specifically made for our CMDB context. Our results are similar in regards to performance, however our conclusion differs since we also take maintainability in consideration as well as performance.

5.3.3 Performance of Graph Query Languages Comparison of Cypher, Gremlin and Native Access in Neo4j [9]

The requirements on a database have become far more complex in recent years where a database has to contain significantly more data than ten or twenty years ago, but still has to provide a good performance on each query. The "go to" database has been the relational databases(SQL) because of their high performance together with their ACID capabilities. However they do start to lack performance and the authors want to look at NoSQL databases, especially graph databases, as a substitute for the relational database.

The authors looked at three aspects of the database. The first is the readability of the queries. The second is the maintainability of the database. The third is the performance of the queries. With these three aspects the authors will then conclude if Neo4j is a good substitute for a relational database.

These benchmarks will be done for a specific context, a web application. Each query that is tested is specifically made to match this context, e.g friend of a friend will be tested since it is a normal query for a web application. They implement the opensocial which has a highly interconnected structure and then fill the database with relevant data. This database is implemented using both MySQL and Neo4j, where Neo4j was implemented using Cypher, Neo4js own native java api as well as Gremlin. Each test is then run multiple times to get an average and the readability of the query languages are evaluated by the authors themselves.

The authors' conclusions are that Neo4j has a lot of potential and even outperforms sql in some cases. Cypher was the language that seemed to have the best readability and therefore was the easiest to maintain. However in some queries it was too surpassed by gremlin and Neo4js native java api and it could not be motivated to use cypher instead of the other two options in those cases.

This related work benchmarks Neo4j in a different scenario than us, but a similarity is that both contexts require a lot of relationships between the data points. However we do the benchmarking based on the same three parameters, readability, writability, and performance. A big difference is that they compare Cypher to SQL and other implementations of Neo4j while we only compare Cypher with SQL.

The authors believe that Neo4j has big potential in the future, and even outperforms SQL in some cases, just like we concluded in our research where the queries that focus heavily on relationships were faster. They do also believe that Cypher was quite easy to implement and read for someone with none or little prior knowledge. Therefore they suggest that Cypher would be easy to maintain in the long run. This paper strengthened our results, especially in regards to the maintainability because we both did a subjective analysis and came to the same conclusion.

Another conclusion they reach is that Cypher is too slow and could not be motivated to use even though it is easier to maintain. This paper was done seven years ago when Neo4j and Cypher were very young, however Cypher have received multiple updates since then. We are not sure which language was most popular back in 2013, but now Cypher is the recommended language according to all modern sources and Neo4j themselves. Which is why we solely focused on Cypher and did not consider the other languages mentioned in this paper.

5.3.4 A Comparison of Relational and Graph Databases for CRM Systems [23]

The paper is a master thesis comparing relational and graph databases for a CRM (Customer Relationship Management) system, more specifically MS SQL compared to Neo4j. Regarding these technologies they specifically want to evaluate them by two merits, their efficiency in performance and the complexity of writing queries for the database. They hypothesize that a CRM system would benefit from being represented as a graph database since its structure is comparative to the advantages of a graph database, namely; representing a lot of many-to-many relations and having frequently changing data, which is similar to a CMDB.

The authors want to reach a conclusion whether a graph database would have advantages over a relational database for representing a CRM system. But they would also like to make a more general comparison between the technologies as a wider scientific contribution, basing these comparisons on performance as well as complexity.

The authors analyse the current, in use CRM which is stored in a MS SQL database for the purposes of determining which sort of query transactions are most frequent and would be most relevant to test when comparing the databases. A database was then created in Neo4j which resembled the original one, which was created to simulate a practical example as well as a ground to perform the performance tests. A set of thirteen queries were designed for SQL as well as Neo4j that were then executed and timed for both technologies for benchmarking purposes. It was originally intended for the authors to determine the complexity of these queries by performing a lexical analysis. Though they found out during their work that they did not know how to do this analysis, and found no literature regarding the subject, therefore they decided to solely focus on the performance aspect.

They reach the conclusion that MS SQL is far superior for their CRM database, having better performance for the tested queries. That it is not enough for the data to be highly interconnected for it to benefit from a graph database but that very specific queries would be needed to give graph databases an edge over relational databases. Meaning that graph databases could be more suitable for other purposes than a CRM.

The authors intended to base their conclusions of a comparison based on performance as well as the complexity of the queries. However they noticed partway through that they do not know how to do the complexity comparison, originally planning to do a lexical analysis. We had a similar premise to compare the languages in writability as well as readability, which could be comparable to their complexity parameter. Similarly we also did not find any good source of information about how to do a syntax comparison, instead deciding to do a subjective comparison.

This paper is quite similar to ours, making a comparison of relational databases and graph databases. However their context was for a CRM System while ours was for a CMDB. They reach the conclusion that MS SQL was superior to Neo4j in regards to performance, which our results also indicate. Since we take maintainability into consideration, we have come to a different conclusion than this paper. Therefore it is hard to directly compare the conclusions of these papers, but in regards to performance the paper strengthens our results.

5.4 Future work

There are some future works that could be done within this field to further solidify our results or build upon them. The first two that do not introduce new concepts are testing the database with a bigger dataset and to conduct more interviews to provide an even better CMDB requirement specification for Axis. Testing with a bigger dataset would improve the benchmarking of performance by making sure that the scalability that our results shows are correct. It would also make it possible to test the parameter that we did not try, "how much data the databases can manage". However this is not relevant for Axis at this time, but could be relevant within another context. If a more complete CMDB requirement specification would be desired by Axis they would need to have wider conversations within the company. This could be in the form of more interviews or workshops, involving more people.

It could be worth conducting the performance benchmarking on a server to resemble how the databases would perform in a realistic scenario which would strengthen our result. Another future work that we see is to examine why there is a big difference between the min and max values of execution times for some queries, where the most prominent being query 6. The min and max varies a lot even though we remove top 10% and least 10% of the execution times. This could simply be because the resulting times are quite fast and just some slight interference from some external process could affect the results. We do however not know if this is the case, and it could be worth looking into this further.

Future research could be made by slightly altering the parameters of research question three, "Should a CMDB be implemented with a graph database or relational database?", by changing the context or the type of database. Instead of a CMDB context it could be worth benchmarking the capabilities of graph databases for other contexts, to further compare graph databases to relational databases. There are a lot of different types of NoSQL databases which are made to be good at their respective focus, where Neo4j is good with interconnected data. Therefore in another context, or this context, there could be another type of database that could be better.

Chapter 6
Conclusions

During the interviews a lot of different CI's were brought up where the most common were internal systems, packages, servers, employees, and licences. Each of which are tracked today, but in different ways. Some might be tracked in an excel document while others are tracked in git, and some CI's are even tracked in multiple locations. This way of tracking makes it difficult to easily obtain certain information or update specific data. It could be solved by having a CMDB that stores all information in one location.

Creating a complete CMDB requirement specification within a limited timeframe is not an easy task, neither is it a realistic task. A requirement specification that could be built upon if Axis so chooses was created. From our research we reached the conclusion that Axis would benefit from some sort of interconnected CMDB between the departments. Throughout the interviews a need of configuration status accounting and traceability was often brought up as the main issue.

From our research everything that mattered within this project could, in one way or another, be implemented in MariaDB as well as Neo4j. However in regards to performance MariaDB was generally quicker, however the speed was not a sole deciding factor because both of them were acceptable within the context. The writability was quite similar for both languages where most queries received the same score, but Neo4j was easier for some queries especially those that changed the structure of the database. The readability was generally easier in Neo4j for the more difficult queries. Since both the writability and readability was easier in Neo4j, it suggests that Neo4j is easier to maintain and would be better for longevity.

Axis' initial problem was that they were having problems keeping track of everything efficiently within the company, hypothesizing that there is a need for an interconnected database between the departments. From our interviews within the company we found this to be a recurring problem that employees mentioned, where each department had a their own way of keeping track of their artefacts. This led to the conclusion that their hypothesis was correct and that a central CMDB within the company is desired. With our resulting CMDB requirement specification Axis has a base, which should be further explored, to ensure that all departments requirements are fulfilled. Axis could implement this database either as a

relational database or graph database. Both database types are viable with no clear disadvantages from using Neo4j. We would however recommend using Neo4j, even with no prior knowledge. This is because we believe that Neo4j would benefit them in the long run with better maintainability and flexibility.

References

[1] Axis. About axis. `https://www.axis.com/about-axis`, Accessed on 2020-05-28.

[2] Wayne Babich. *Software Configuration Management - Coordination for Team Productivity*. Addison-Wesley Publishing Company, 1986.

[3] Michael Brenner, Markus Garschhammer, Martin Sailer, and Thomas Schaaf. Cmdb - yet another mib? on reusing management model concepts in itil configuration management. In Radu State, Sven van der Meer, Declan O'Sullivan, and Tom Pfeifer, editors, *DSOM*, volume 4269 of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2006.

[4] Michael Brenner and Markus Gillmeister. Designing cmdb data models with good utility and limited complexity. pages 1–15, 05 2014.

[5] DAVID CONLEY. Configuration status accounting made affordable. *Naval Engineers Journal*, 103:63 – 70, 03 2009.

[6] M. A. Daniels. *Principles of Configuration Management*. Advanced Applications Consultants, Inc, 1985. (Chapter 2, 3, 4, and 5).

[7] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, SCM '91, page 1–18, New York, NY, USA, 1991. Association for Computing Machinery.

[8] Peter H. Feiler. *Configuration Management Models in Commercial Environments*. Technical Report SEI-91-TR-7, Software Engineering Institute, 1991. (Chapter 1, 3, 5, and 6).

[9] Florian Holzschuher and René Peinl. Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, page 195–204, New York, NY, USA, 2013. Association for Computing Machinery.

[10] Ian. What does acid mean in database systems?, 2016. `https://database.guide/what-is-acid-in-databases/`, Accessed on 2020-05-28.

[11] Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). Standard, International Organization for Standardization, December 2016.

[12] Marion Kelly. *Configuration Management - The Changing Image*. McGraw-Hill Book Company, 1996. (Chapter 5 and 7).

[13] Brian Kerr. Mend the divide by creating and maintaining a healthy cmdb. `https://info.axiossystems.com/cmdb-improvement-whitepaper`.

[14] MongoDB. Nosql vs relational databases. `https://www.mongodb.com/scale/nosql-vs-relational-databases`, Accessed on 2020-05-28.

[15] Neo4j. Cypher, the graph query language. `https://neo4j.com/cypher-graph-query-language/`, Accessed on 2020-05-28.

[16] Neo4j. Neo4j – the leader in graph databases. `https://neo4j.com/company/?ref=footer`, Accessed on 2020-05-28.

[17] Neo4j. What is a graph database? `https://neo4j.com/developer/graph-database/`, Accessed on 2020-05-28.

[18] Oracle. Database. `https://www.oracle.com/database/what-is-database.html`, Accessed on 2020-05-28.

[19] Oracle. What a relational database is. `https://www.oracle.com/database/what-is-a-relational-database/`, Accessed on 2020-05-28.

[20] Jennifer Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, Hoboken, NJ, 4 edition, 2015.

[21] Möstl C. Bär F. et al. Stiefel, S. Graph-datenbanken als grundlage des configuration managements – eine untersuchung am beispiel von neo4 j. *HMD 53*, pages 470–485, 2016. https://doi.org/10.1365/s40702-016-0241-x.

[22] Boran Gögetap Thomas Schaaf. Requirements and recommendations for the realization of a configuration management database.

[23] Victor Winberg and Jan Zubac. A comparison of relational and graph databases for crm systems, 2019. Student Paper.

[24] Mathias Salle Yassine Faih, Abdel Boulmakoul. Configuration management database and system. US7926031B2, United States Patent, 2013. Apr. 12, 2011.

Appendices

Appendix A
Queries

Query 1

```
##MariaDB
ALTER TABLE employees
ADD COLUMN IF NOT EXISTS (age int NOT NULL DEFAULT 0);

##Neo4j
MATCH (e:employees)
SET e.age = 0
RETURN e
```

Query 2

```
##MariaDB
UPDATE IGNORE `internal_system_owners`
SET parent_employee_id = str(randomIds[0])
WHERE parent_employee_id = str(randomIds[1]);

DELETE FROM `internal_system_owners`
WHERE parent_employee_id =  str(randomIds[1]);

##Neo4j
MATCH (i:employees {internalId: str(randomIds[0])})-
[rel:internal_system_owners]- (x:internal_systems)
MATCH (p:employees {internalId: str(randomIds[1])})
CALL apoc.refactor.from(rel, p)
YIELD output, input
RETURN output, input
```

```
MATCH (a:employees)-[r:internal_system_owners]-(b:internal_systems)
WITH a, type(r) as type, collect(r) as rels, b
WHERE size(rels) > 1
UNWIND tail(rels) as rel
DELETE rel
```

Query 3

```
##MariaDB
WITH RECURSIVE `rec` AS (
       SELECT *
       FROM `dependencies`
       WHERE parent_system_id IN (
       SELECT (child_internal_system_id)
       FROM `package_systems`
       WHERE parent_package_id = internalId
       )
       UNION
       SELECT f.*
       FROM `dependencies` AS f, `rec` AS c
       WHERE f.parent_system_id = c.child_system_id
       )
       SELECT * FROM (SELECT parent_system_id as id
       FROM `rec`
       UNION
       SELECT child_system_id as id
       FROM `rec`) as T
       JOIN `internal_systems`
       On (T.id=internal_systems.id);

##Neo4j
MATCH (e:external_systems {internalId:internalId})-[]->(i:internal_systems)
CALL apoc.path.subgraphNodes(i, {relationshipFilter:'dependencies>'})
YIELD node
RETURN count(distinct node)
```

Query 4

```
##MariaDB
SELECT *
FROM internal_systems
WHERE description LIKE \% + word + \%;

##Neo4j
MATCH (i:internal_systems)
WHERE i.description CONTAINS word
RETURN i
```

Query 5

```
##MariaDB
SELECT *
FROM internal_systems
JOIN package_systems
ON internal_systems.id=package_systems.child_internal_system_id
JOIN external_systems
ON external_systems.id=package_systems.parent_package_id
JOIN internal_system_owners
ON internal_system_owners.child_system_id=internal_systems.id
JOIN employees
ON employees.id=internal_system_owners.parent_employee_id
JOIN hosts
ON hosts.child_internal_system_id=internal_systems.id
JOIN servers
ON servers.id= hosts.parent_server_id
WHERE internal_systems.id=internalId;

##Neo4j
MATCH (i:internal_systems {internalId: internalId})-[:hosts]-(s:servers),
(i:internal_systems {internalId: internalId })-[:internal_system_owners]
-(e:employees),
(i:internal_systems {internalId: internalId})-[:package_systems]
-(es:external_systems)
RETURN i, s, e, es
```

Query 6

```
##MariaDB
SELECT *
FROM licences
WHERE expiration_date < date;

##Neo4j
MATCH (l: licences)
WHERE l.expiration_date < date( ‘date')
RETURN l
```

Query 7

```
##MariaDB
CREATE TABLE `projects` (
`id` int NOT NULL AUTO_INCREMENT,
`name` varchar(50) NOT NULL,
PRIMARY KEY (`id`));

CREATE TABLE `project_internal_systems`(
`parent_project_id` int NOT NULL,
`child_internal_systems_id` int NOT NULL,
```

```
PRIMARY KEY (`parent_project_id`, `child_internal_systems_id`),
FOREIGN KEY (parent_project_id) REFERENCES projects(id),
FOREIGN KEY (child_internal_systems_id) REFERENCES internal_systems(id));

INSERT INTO `projects` (name)
VALUES('project1');

INSERT INTO `project_internal_systems` (parent_project_id, child_internal_systems_id)
VALUES('1', '1');

##Neo4j
MATCH (i:internal_systems {internalId:1})
MERGE (p:projects {internalId: 1, name: 'project1'})-
[:project_internal_systems]->(i)
RETURN p
```

Query 8

```
##MariaDB
INSERT INTO `internal_systems` (name, version, git_repository, description, status)
VALUES('internal_systemX', 5, 'itis', 'this is a description', 'maintain');

SELECT MAX(id) FROM `internal_systems`;

INSERT INTO `dependencies` (parent_system_id, child_system_id)
VALUES (val, val)

##Neo4j
MATCH (i:internal_systems)
RETURN max(i.internalId)

CREATE (i:internal_systems {name: 'internalSystemsX', internalId: currentId,
description: 'this is a description', git_repository: 'itis', version: 8,
status: 'notInUse'})
RETURN i

MATCH (a:internal_systems {internalId: currentId}),
(b:internal_systems {internalId:str(randomId)})
CREATE (a)-[:dependencies]->(b)
```

Appendix B
UML

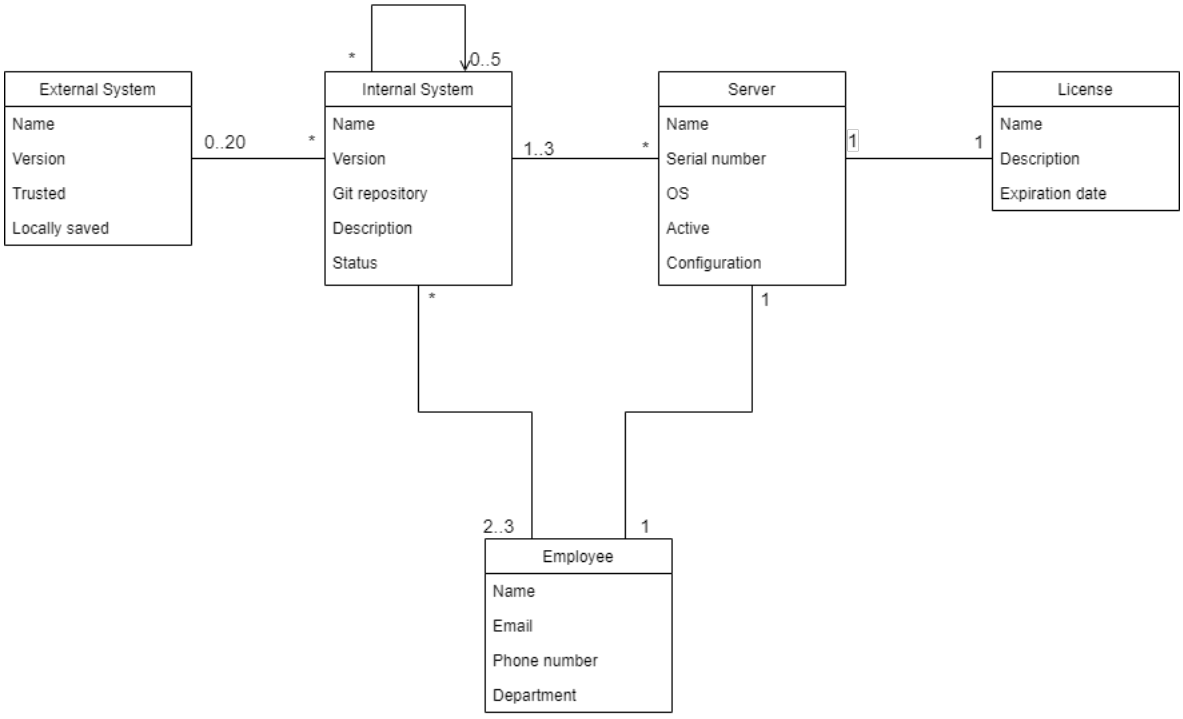


Figure B.1: UML of MVP

Appendix C
Interview guide (in swedish)

1. Vad är din arbetsttitel?
2. Vet du vad CSA(Configuration status accounting) är? Annars bestiver vi konceptet.
3. Utifrån vår beskrivning av CSA, ser du en användning av det, och i så fall hur? Hur löser du detta idag?
4. Specifika områden vi vill fråga om hur de löser i dagsläget och hur de önskar att det eventuellt kunnat lösas bättre:
    - Kommunikation mellan avdelningar
        - Finns där någon kommunikation som hade kunnat förmedlats på ett enklare sätt utan att behöva söka upp någon? Exempelvis
            * Systemägare
            * Statusen på ett system
            * Vem är det som utvecklar på detta systemet just nu?
            * Vem är det som har denna laptop?
    - Audits
        - Har ni något sätt att se ifall alla krav för en release är uppfyllda? Exempelvis att systemet blivit testat innan release?
    - Impact analysis
        - Hur ser ni vilka dependencies ett system har, samt vilken impakt en viss ändring har?
    - Configuration Identification
        - Finns där något du skulle vilja ha mer koll på som du inte har idag? Antingen någon information om ett system som är svår att hitta eller någon koppling.

- Har ni problemet att ni inte vet vad som finns?
- Tycker du detta borde hanteras manuellt eller automatiskt?

5. Något annat du vill tillägga?

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-11

EXAMENSARBETE Benchmarking and comparison of a relational and a graph database in a CMDB context
STUDENTER Rasmus Berggren, Dennis Londögård
HANDLEDARE Lars Bendix (LTH), Guido Guidos (Axis)
EXAMINATOR Per Andersson (LTH)

Grafdatabaser, framtiden för konfigurations hantering?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Rasmus Berggren, Dennis Londögård**

Med internets tillväxt har behoven av databaser ändrats, vilket beror till stor del på att datan har blivit mer flexibel i sin struktur. Detta har medfört att grafdatabaser har vuxit sig stora och har börjat ta striden mot de traditionella relationsdatabasena. Därför har vi valt att jämföra dessa två databas typer i ett CMDB kontext.

Grafdatabaser har vuxit i popularitet på grund av deras flexibilitet och prestanda med väl sammankopplad data. Neo4j är en av de ledande grafdatabaserna som vuxit sig stora de senaste åren med kunder såsom Microsoft och Telia. Den stora anledningen är att de behandlar en stor mängd ostrukturerad data som innehåller många sammankopplingar. Det sägs att relationsdatabaserna är för oflexibla för många moderna behov och inte kan hantera de nya kraven som ställs på databaser.

Många IT företag använder sig utav en konfigurations hanterings databas(CMDB) som är en central punkt i företaget för att spara undan viktig data med dess relationer och attribut. Denna data förändras och växer då nya behov uppstår inom företaget. Förändringen sker i form av ny data, ändring av datastruktur, eller nya relationer. Detta indikerar på att det skulle passa bättre att implementeras i en grafdatabas än en relationsdatabas, då datastrukturen växer till en väl sammankopplad databas. Vi har därför valt att undersöka ifall en grafdatabas är bättre än en relationsdatabas i ett CMDB kontext.

Vårt examensarbete utfördes på Axis Communication AB där vi först skapade en CMDB kravspecifikation åt Axis genom att göra en litteraturstudie, intervjua anställda, samt undersöka en existerande databas som används för konfigurationshantering inom företag. Genom att sammanställa all information så skapades en minimum viable product som symboliserar Axis CMDB. Därefter implementerades denna som en relationsdatabas i MariaDB samt som en grafdatabas i Neo4j. Dessa fylldes sedan med en realistisk mängd data för att benchmarka i en relevant kontext. Denna undersökning utfördes genom att jämföra databaserna på två parametrar, prestanda samt hur lätta den skulle vara att underhålla. En kombination av dessa parametrar användes för att avgöra om en CMDB är bäst implementerad som en grafdatabas eller inte.

Vi skapade en CMDB kravspecifikation för Axis som användes för att utföra jämförelsen. Under vår benchmarking indikerade resultaten på att båda databas typerna var funktionella för kontexten. Dock hade MariaDB bättre prestanda för nästan alla queries, även fast datan var väl sammankopplad vilket egentligen skulle indikera på att Neo4j skulle prestera bättre än MariaDB. Däremot bedömde vi att Neo4j skulle vara enklare att underhålla. Vår slutsats blev att båda databas typerna skulle vara lämpliga för ett CMDB kontext, vi anser dock att Neo4j hade gynnat CMDBn i längden.