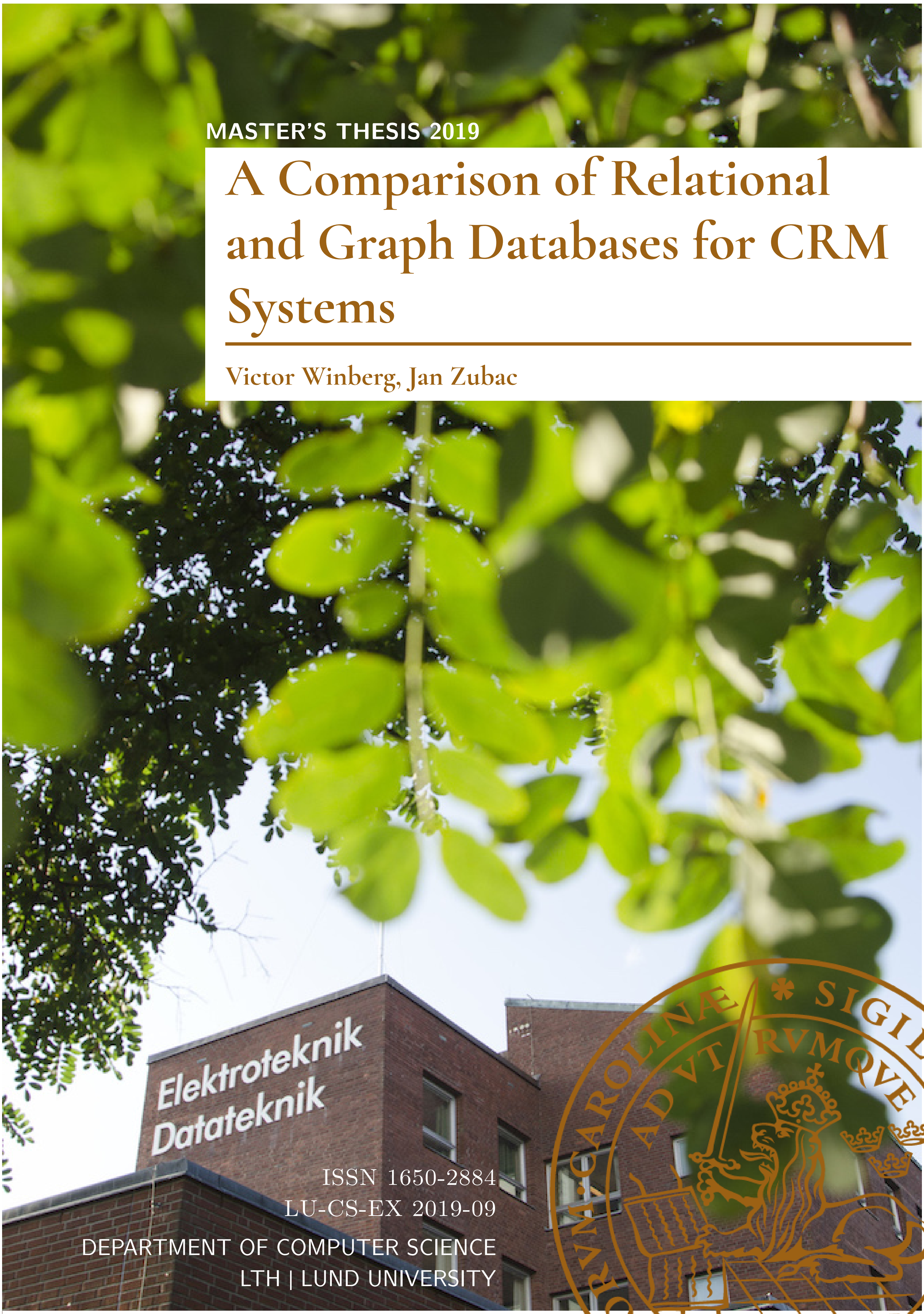
MASTER'S THESIS 2019

A Comparison of Relational and Graph Databases for CRM Systems

Victor Winberg, Jan Zubac



ISSN 1650-2884
LU-CS-EX 2019-09

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-09

A Comparison of Relational and Graph Databases for CRM Systems

Victor Winberg, Jan Zubac

A Comparison of Relational and Graph Databases for CRM Systems

Victor Winberg
`victor.m.winberg@gmail.com`

Jan Zubac
`jan.zubac@gmail.com`

June 11, 2019

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Johan Groth, `johan.groth@lime.tech`
Per Andersson, `per.andersson@cs.lth.se`

Examiner: Flavius Gruian, `flavius.gruian@cs.lth.se`

Abstract

In an age of increasing usage of heavily interconnected platforms, graph databases have increased in popularity due to their ease of modelling these systems. This thesis will evaluate if the graph database Neo4j can be used to model Lime's CRM System efficiently by comparing the performance of Neo4j and MS SQL on queries similar to those existing in Lime CRM.

To benchmark the databases, they were created with the core entities of the Lime CRM system. The databases are created with the same structure and schema as in Lime CRM. Thereafter, the databases were populated with randomly generated data. These databases were then queried a large amount of times with random values in every query to prevent caching.

Our results show that MS SQL was 4-10 times faster for insertion queries. Furthermore, MS SQL was also 2-5 times more efficient at handling queries that return many rows/nodes of data.

Keywords: SQL, Neo4j, graph databases, benchmark, CRM

Acknowledgements

We would like to thank our supervisor at Lime Technologies, Johan Groth, for the help during this thesis project with Lime CRM specific questions. We would also like to thank Jens Gustafsson at Lime Technologies for initiating the contact with us. Lastly, we would like to thank our examiner, Flavius Gruian, for thoroughly evaluating the report to make sure that it is of the best quality before being published.

Contents

Chapter 1
Introduction

In recent years, the popularity of graph databases has steadily increased into becoming a mainstream storage alternative available to use by most database vendors. This is most likely due to an increased need to represent enormous amounts of data with many relationships between the data, such as social networks, e-commerce and customer relationship management systems. The structural benefits of using graphs for modeling objects and interactions in complex systems is one of the fundamental data abstractions in computer science [33]. Hence, applying graph theory on databases should apprehend the same structural benefits as in graphs in graph theory.

Customer-relationship management (CRM) conducts the management, organization and administration process of customers and customer relations in a company [5]. Moreover, it includes the analysis about customers' history and potential customers to improve business. Thus, companies often use IT solutions to better synchronize and store data to a central CRM system, such as the Lime Technologies CRM-systems [5].

In a large CRM-system it might be unsuitable to use traditional relational databases as graph databases outperform them when dealing with large tables where many columns have null values, also called sparse tables [33]. Furthermore, relational databases get outperformed by graph databases when the relational database is using attribute tables, when there are a lot of many-to-many relationships in the database, when the database has tree-like characteristics and when the schema of the database frequently changes [33].

Graph databases use the mathematical definition of graphs as an internal representation. This ensures that the graph theory defined in mathematics can be applied to these types of databases. In graph theory context is made up with nodes which are connected by edges [31]. A node represents an entity or instance of a given label, such as users or accounts, and in a CRM-system, it could be customers, invoices, orders, etc. An edge describes the connection between two nodes, which represents the relationships, and could describe knowledge, friendships, occupation, etc.

Graph databases allow database designers to represent objects and relationships in a coherent manner due to its semantic properties with nodes, edges and attributes [2]. Due to the graph structure of graph databases, the query time performance scales much better than for relational databases. The reason for this is that the execution time of graph database queries is proportional to the nodes traversed in the query. Thus, querying large graph databases is far more effective than querying relational databases as relational databases have to traverse the complete tables to find the query result [29]. This thesis will focus on evaluating the graph database Neo4j, which mainly uses the query language Cypher.

Relational databases are based on the relational model, introduced by E. F. Codd, where data is stored in a tabular form and due to its simplicity gained worldwide popularity, which is a huge benefit itself [2] [21]. The pros of using relational databases is also their support for the ACID properties (Atomicity, Consistency, Isolation, and Durability), which many non-relation databases does not have [18]. However, a con with relational databases is that when the database grows in size and complexity it does not perform as well as graph databases, since it has scalability issues [14]. This thesis will focus on the relational database Microsoft SQL Server and the query language Transact-SQL, which is used in the Microsoft SQL Server database.

This thesis provides a comparison and a benchmark study of relational and graph databases in performance and complexity on storing and processing a large-scale customer relationship management system. The three main areas of focus, that will be concluded in this thesis, are the following:

1. A benchmark study comparing the performance differences between querying with relational databases and with graph databases.
2. A token analysis study comparing the complexity of querying with the relational database syntax and with the graph database syntax.
3. An implementation of a simple CRM-system prototype with both a relational database and a graph database as the storage back-end, simulating a practical example showing a real-world application.

1.1 Problem Statement

This thesis will focus on evaluating and comparing relational and graph databases regarding performance and complexity. The research will conduct a benchmark comparing the querying performance differences on a CRM-system developed by Lime Technologies. Furthermore, the research will also provide the complexity differences of the query language Transact-SQL that is used in Microsoft SQL Server and Cypher, the query language used by the graph database Neo4j, by analyzing tokens, which in some sense relates to the readability of the queries. A prototype including these main focus areas will conclude this research by clearly visualizing a simple CRM-system in use by simulating a real-world practical example using the different measured queries with both relational and graph databases.

The research questions that should be answered in this thesis:

1. How does the performance of graph databases compared to relational databases differ, when used in a large complex CRM database system, and when?
    1.1. What kind of structures affect the query performances?

    1.2. What kind of queries provides the extreme cases?

2. How does the lexical complexity of the queries differ for different databases in a CRM-system?

    2.1. Which readability differences exist in Cypher and Transact-SQL?

    2.2. How does the readability differ for queries when query-complexity changes?

1.2 Contribution Statement

This master thesis will contribute to the knowledge of the difference between relational and graph databases in regards to complexity (readability of the queries) as well as performance. Specifically, the thesis will provide information in regards to whether non-relational or relational databases are best suited for CRM-systems and why or why not.

In the thesis, the work was divided in the following way: Victor mostly worked on the benchmarking prototype and Jan mostly worked on evaluating the Lime CRM and writing queries thereafter. However, most of the programming was done in pairs so we have both participated on all parts of the development.

When it comes to report writing, Victor wrote the introduction and the theory about the relational databases as well as the theory about regular expressions and lexical analysis. Jan wrote the rest of the theory section as well as the majority of the method and the design of experiments section, except for the parts about execution plans and big O notation. Victor wrote the result section and Jan wrote most of the discussion.

The abstract and the conclusion was done in unison.

1.3 Outline

Chapter 2 describes the theoretical background needed to understand the thesis as well as how the given problem was solved.

Chapter 3 consists of a description of the methodology that was used to conduct the study.

Chapter 4 describes why and how the given methodology was chosen, e.g. why the specific queries that were benchmarked were chosen. It also describes how the complexity analysis was set up and why it was done in that way.

Chapter 5 contains the results obtained from conducting the database comparison such as graphs with performance data. It also contains a discussion of the obtained results.

Chapter 6 concludes the finding from the experiment and presents possibilities for future work in regards to the difference between relational databases and graph databases.

Chapter 2
Background and Related Work

This section contains the theoretical background that the work conducted in this thesis is based on. It also contains a short description of relevant studies conducted. The section will mostly focus on describing the different databases, moreover, theory regarding what CRM systems are and how they are used will also be provided.

2.1 Databases

Databases are essential in the activities of our modern day society by being involved in everything from bank withdrawals and online purchases to storing images and videos [7]. A database is, in the most simple form, a collection of data, where data is facts that can be recorded, such as names and account numbers. These records could be stored in an address book, with indexing, or on a hard drive, using a software such as Excel. Complying to the property of being logically coherent and representing some aspect of the real world a database can be used by an intended group for a specific purpose.

A computerized database can contain records of any size, amount and complexity. Thus, the database is required to be managed so that users can search, retrieve and update the data. This is done either by application programs with specific tasks for the database or by a database management system (DBMS). An application program uses specific queries or requests for retrieving or updating the data. A database management system is a *general-purpose software system* that handles all operations and maintenance on the database [7].

A DBMS also provides a conceptual representation of data by excluding details of how the data is stored. The abstraction that organizes records and standardizes their relationships to one another is the *data model*. A data model explicitly defines the structure of data using logical concepts, such as objects, properties and relationships. There are many data models that provides data abstraction to the users with different pros and cons, depending on each specific use case.

Section 2.2 describes the relational database model and gives an overview of relational databases. Section 2.3 describes the different types of graph database models and gives an overview of graph databases.

2.2 Relational Databases

A relational database is a computerized database, no different from the computerized database described in Section 2.1, based on a relational database model. A relational model, in the most simple form, defines the database as a collection of relations. Characterized as a simplistic and mathematical approach of managing data using *first-order predicate logic*, proposed by E. F. Codd in 1970 [6], it acquired world-wide popularity. Codd introduced a representation of all data to be arrays of *n*-tuples grouped into relations. A relation has a *degree n* which responds to how many different parts that is grouped, also represented as a *n*-tuple. The example in figure 2.1 illustrates a relation of degree 4, by E. F. Codd.

supply	*(supplier*	*part*	*project*	*quantity)*
	1	2	5	17
	1	3	5	23
	2	3	7	9
	2	7	5	4
	4	1	1	12

Figure 2.1: A relation of degree 4 by E. F. Codd [6]

Informally a relation is seen as a table of values with a simple linear or flat structure [7]. A *tuple*, in the relational model terminology, corresponds to a row in a table. An attribute defining some part of a tuple correlates to a column header. The values in a column has a defined data type represented by a domain. A domain is a set of atomic values with a specified format. An atomic value is an indivisible value, such as a name, a phone number or an age. A column with the value "*Victor Winberg, Jan Zubac*" is not an atomic value.

2.2.1 ACID properties

One of the key concepts of relational databases is that each sequence of operations with the database is seen as a transaction. This can be seen as "Either the transaction is successful, or not". A set of properties that ensures validity, even when errors occur when performing this sequence of operations, is called the ACID properties [12]. The transaction property ascertains that relational databases fulfill the ACID properties, which are the following:

Atomicity: The atomicity property states that all operations on data, e.g. inserts, updates must be handled by the DBMS as one operation. Meaning that if several queries are written after each other, they all must succeed or they are all considered as failed and are therefore abandoned.

Consistency: The consistency property ensures that a transaction can only be completed if all constraints and actions posed on the data involved are fulfilled. This ensures that the database is always in a valid state.

Isolation: The isolation property states that every transaction must be isolated from another transaction. An example of this would be if a transaction is done where person A sends money to person B. During that transaction, another transaction shall not be able to see that money has been withdrawn from person A's account before person B has received the money, that is, until the first transaction is completed. This ensures that concurrent access to the database behaves in the same way as sequential access.

Durability: The durability property ensures that all changes to data are permanent. This implies that the results of a database transaction shall remain the same even if there is a power outage or a system failure.

2.2.2 Constraints

In Relational DBMS (RDBMS), there are several different types of constraints that affect when a transaction is deemed to be valid. Some of the most common constraint types will be discussed below.

Domain constraints

Domain constraints are constraints that enforce the values stored in a database to follow the domain specified [7]. In figure 2.2, the name data field is constrained by a domain constraint to be a varchar (string) with a length of at most 20 characters. If a transaction tries to save a data type that is not a varchar, or if a varchar with a length greater than 20 is attempting to be inserted into the database, the domain constraint for the variable *name* is violated and an error occurs. Furthermore, in figure 2.2, `CHECK(zipcode > 0)` is also a domain constraint as it constrains the domain of the *zipcode* data entries to be integers between 1000 and the maximum size of an integer.

```
CREATE TABLE customers {
    id INT,
    name VARCHAR(20),
    address VARCHAR(25),
    zipcode INT,
    country VARCHAR(30),
    telephone VARCHAR(20),
    CHECK (zipcode > 999)
}
```

Figure 2.2: A table creation schema for the customer relational model in figure 2.4

Key constraints

Key constraints force primary keys and foreign keys to have non-NULL values. Furthermore, key constraints also ascertain that each primary key is unique. For example, if the id of the customer in the relational the relational model in figure 2.4 is not unique, it would not be possible to distinguish between customers that have all other fields except the id-field in common. The primary key constraint solves this problem by ensuring that primary keys must have unique values [7].

Moreover, key constraints check relations between tables by making sure that tables that have foreign keys from other tables contain some of the values from the table that they are referencing [7]. An example of this using the relational model in figure 2.4 is the fact that `customer_id` in the Orders entity is a foreign key that references the id in the Customers entity. The foreign key integrity constraint ensures that all `customer_id`:s inserted into Orders must be present in the Customers table, otherwise, orders without valid customers could be created.

Null value constraints

Null value constraints are imposed on the database to ensure that some specific values in the database are not stored as `NULL`. This constraint is used to ensure that some key data components must have a value [7]. For example, for the customer table in figure 2.4 it would not be appropriate to store Customers without them providing a name. This can be ensured by constraining the `name` column to not contain `NULL` values.

2.2.3 Structured Query Language

The Structured Query Language (SQL) is a sub-language of the many data query languages, where a query language is a computer language used for querying or requesting data from a database. The differences between query languages is their syntax and appliance while their similarities is to give factual answers to factual questions. SQL is based on *relational algebra* and *tuple relational calculus* created by E. F. Codd [32]. The relational algebra is built with five primitive operators: the selection, the projection, the cross product, the set union, and the set difference. The tuple relational calculus is a non-procedural or declarative language which is to filter tuples based on a given condition. Together these two definitions form the foundation of SQL that became, and is still today, the most world-wide used query language.

```
SELECT field1 [,"field2",etc]
FROM table
[WHERE "condition"]
[GROUP BY "field"]
[ORDER BY "field"]
```

Figure 2.3: A basic SQL query [4]

A basic SQL is structured using a valid combination of tokens each preceded by a keyword. The example in figure 2.3 shows a basic query with two mandatory keywords *SELECT* and *FROM*, and some optional keywords indicated by the containing brackets [4]. Keywords are words that are reserved with an acquired meaning, such as; *SELECT*, which is used to specify fields to be retrieved, and *FROM*, that specifies the table (or relation) to fetch data from.
A relational model describes a general scenario of a system often practical to use when managing queries. The relational model example in figure 2.4 defines the relation between *Customers* and *Orders*, where a certain relationship between the models are identified by the arrow "->".

```
Customers (id, name, address, zip code, country, telephone)
Orders(id, date_order, date_delivery, customer_id -> Customers)
```

Figure 2.4: A simple relational model

id	name	country
1	Alice Andersson	Sweden
2	Benjamin Bauer	Germany
3	Chloé Chevalier	France

Table 2.1: Result of a *SELECT* query on customers.

id	date_order	date_delivery	name
1	1996-07-04	1996-08-02	Chloé Chevalier
2	1996-08-26	1996-08-28	Chloé Chevalier
3	1997-02-12	1997-02-27	Alice Andersson

Table 2.2: Result of a *SELECT* and *JOIN*-query on orders and customers.

The most essential SQL query is the "show all contents of a table"-query, that would, for example in figure 2.4, translate into `SELECT * FROM customers;`. The asterisk symbol "*" is used to select all fields. The query for "show all contents of a table combined with another table using common values" is used for combining tuples from other relations, or rows from other tables. This could, using the *JOIN* keyword, translate into the query below:

```
SELECT * FROM orders
JOIN customers
ON orders.customer_id = customers.id;
```

This instruction shows all orders' date order and date delivery, including the name, address, zip code, country and telephone of the related customer.
An example of a result retrieved from the two queries, "show all contents of a table" (left) and "show all contents of a table combined with another table using common values" (right), could be seen in Table 2.1 and 2.2.

2.3 Graph Databases

A graph database is a database where data is stored structured as a graph. This means that the database is built using nodes and edges. The nodes in the graph typically represent data entities and the edges represent relations between nodes. For example, a graph database that contains data about pet owners might have humans and pets as nodes and ownership as edges. This thesis will focus on describing labeled property graphs, as the Neo4j graphs are built as label property graphs. Section 2.3.1 describes the property graphs that is used in Neo4j.

2.3.1 Property Graphs

Labeled property graphs contains nodes and relationships for the data stored in the model as many other graph models. However, in a labeled property graph, the edges are directed, meaning that they only go in one direction. An example of this is if person A and person B are friends on a social media platform. It then has to be explicitly stated that person A is friends with person B and that person B is friends with person A using two edges. Furthermore, as the name suggest, the nodes in a labeled node graph can be labeled with one or multiple labels. The nodes in a labeled node graph contain key-value pairs, meaning that they can contain more than one data type at the same time, similar to objects used in object oriented programming. The relationships (edges) of a labeled graphs also must have labels that further describe the relationship between the nodes connected. These edges can also have multiple data fields which means that they can be interpreted as objects as well. Every node in the Neo4j graph database has a unique identifier. Finally, it is also important to mention that edges in a labeled node graph always have a start node and a finish node, e.g. there cannot be an outgoing edge from a node without another node that is connected to that edge [29].

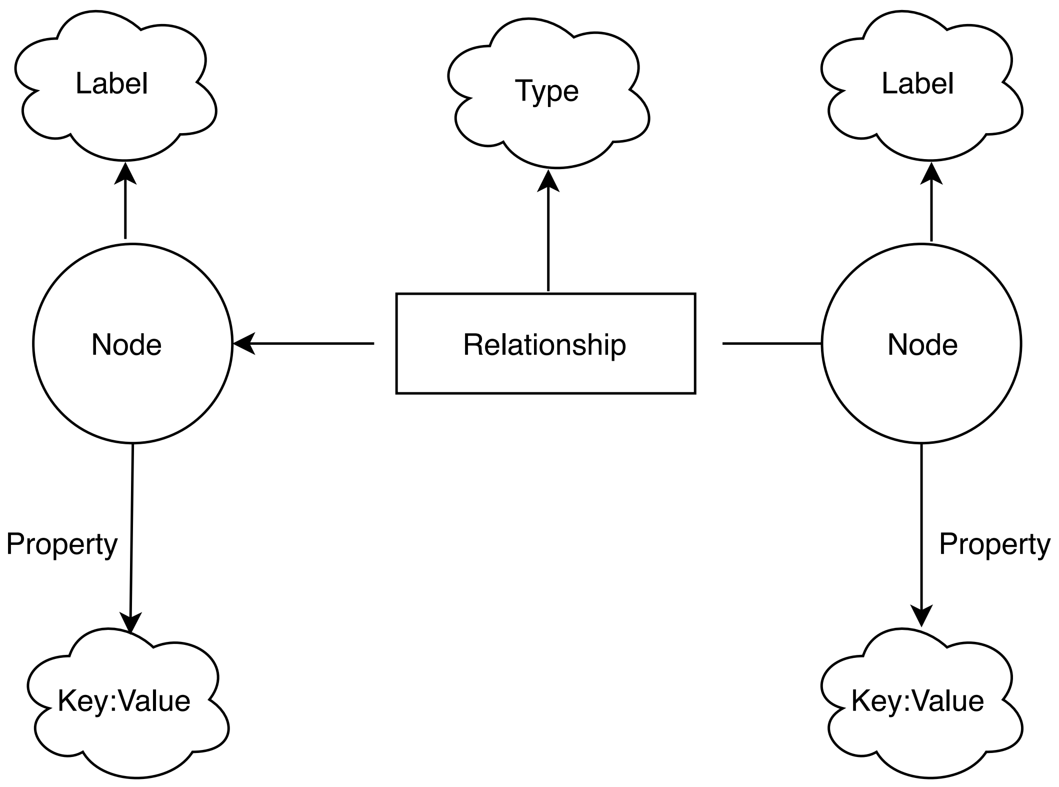


Figure 2.5: A general overview of the different data entities in the Neo4j labeled property graph

2.3.2 Graph Processing

Graph processing and graph storage are closely related. In a way, graph processing is also about how the data is stored in the memory of the database engine. However, it is instead the definition of how relationships between data are encoded and processed in the database instead of the raw storage information such as which byte of a relationship data entry that represents what. Graph databases can be processed in many different ways and still be perceived as a graph storage by the user due to them not seeing the underlying structure. There are index-free adjacency graph databases where each node has references to all its neighbours, e.g. all nodes that have relationships tied to that given node. The databases are said to have native graph processing. On the other hand, there are graph databases that store the index of nodes in a table instead of storing the connections between them. These graph databases are usually interacted with in a similar manner to how interactions are made with SQL-databases. [29].

Native Graph Processing

As mentioned in section 2.3.2, native graph processing is used in index-free adjacency graphs. The definition of an index-free adjacency graph database is that the database stores an explicit reference to all nodes adjacent to it in the graph. This increased efficiency significantly as all node indexes are stored "locally" by each node instead of having a very large, global table containing the indexes of all nodes. If one wants to find the neighbours of a node, it is much easier to query the nodes with indexes adjacent to the node in focus instead of searching through the whole graph for a specific index [29]. This processing property is what makes graph query times proportional to the amount of nodes traversed in the query instead of the graph being proportional to the total amount of nodes and edges. Neo4j uses native graph processing which ensures the efficiency benefits mentioned above.

Non-Native Graph Processing

As mentioned in section 2.3.2, graphs that use non-native graph processing use indexes to connect nodes. The fact that indexes have to be looked up in a table in graph databases with non-native graph processing adds further computational moments in most queries and therefore the execution time is generally longer than for graphs with native graph processing. An example of this can be illustrated with the help of figure 2.6 which represents a table that stores the information about friendships in a graph database with non-native graphs processing. To be able to find the Alice's friends, and index lookup with the cost of $O(\log n)$ has to be performed to be able to find Alice and her related entries. However, if one was to find everyone who is friends with Alice instead, several index lookup with the cost of $O(\log n)$ would have to be done to evaluate every possible entry that might be friends with Alice. So finding Alice's friends is $O(\log n)$, however, finding who is friends with Alice has a cost of $O(m \log n)$ where m is the number of nodes that needs to be evaluated. In a graph database with native graph processing on the other hand, the cost would always be $O(m)$, due to the indexes for each neighbour being stored locally and therefore only needing $O(1)$ to fetch every neighbour index. This clearly shows that native graph processing is more efficient in most cases [29].

To sum it up in short: Native Graph Processing evaluates Node relationships in querying, leading to the execution time being proportial to the amount of nodes and relationships traversed whereas Non-Native Graph Processing uses indexing in a table-like manner to evaluate relationships and therefore the execution time is propertional to the table sizes.

2.3.3 Graph Storage

As with graph processing mentioned in section 2.3.2, graph storage can be native or non-native. Native graph storage is when the structure of the storage (database) is specifically designed to handle graphs. This makes the storage very optimal, as it is specialized to handle and store graph data. Neo4j uses native graph storage, therefore, the focus will be on explaining native graph storage as it is what will be used as a base for this thesis. Due to Neo4j having

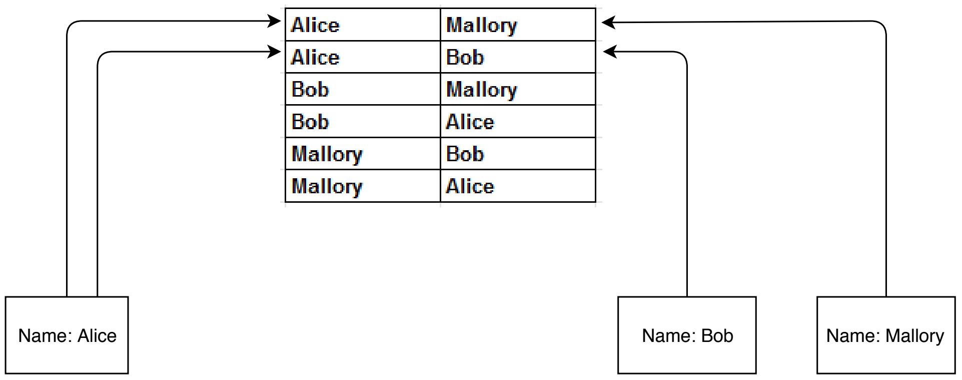


Figure 2.6: Non-Native Graph Processing for a friendship database where SQL-like indexing is used for node traversal

native data storage and native data processing, it is a graph database with complete native support. This means that every aspect of Neo4j is specifically designed to handle and store graphs, which is one of the key factors to Neo4j's efficiency [29].

Neo4j stores different data in separate files to ensure that the data is easier to find. This means that relationships are stored in one data store, nodes are stored in another data store etc. For all data stores that Neo4j uses, each data record is of a fixed size. For example, in version 2.2 of Neo4j, the node records had the size of 15 bytes as seen in figure 2.7. A fixed size for each record makes it very easy to find a specific record. Let's say that we want to find the record with id 300, due to the fixed size, it is known that this record is located at byte $300 * 15 = 4500$ which makes it a $O(1)$ operation to fetch that data record [29].

The data that is stored in separate files are nodes and relationships as mentioned above, as well as labels and properties. Labels represent what type of Node it is, e.g. "Person" or "Movie", properties represent the internal data of a Node or a Relationship and are built using key-values such as name: "Victor Winberg" which represents the name of a Person. Nodes are entities that hold properties and labels and represent connection points in the graph, e.g. a Node can have a Person label and a name property as in the previous sentence [29]. The connection between the different data entities can be observed in figure 2.5. It should be noted that relationships can contain properties, however, it was omitted in the figure due to it being optional.

As seen in figure 2.7, relationships are stored in a more complex way as they are harder to represent. The first parts in a relationship record represent the id:s of the nodes that are connected through the relationship. Thereafter, a pointer is stored that indicates what type of relationship it is. Moreover, a pointer to the last and the next relationships for both the start node and the finish node are stored.

2.3.4 Constraints

Graph databases, and Neo4j in particular do not have access to the wide range of constraints that relational databases have. The constraints available for Neo4j were not used in the database to be benchmarked, but it is worth noting possible constraints as the existence and lack of constraints is worth discussing. Neo4j does not have any constraints that can enforce a node to have outgoing or incoming edges, it can only enforce uniqueness and existence of

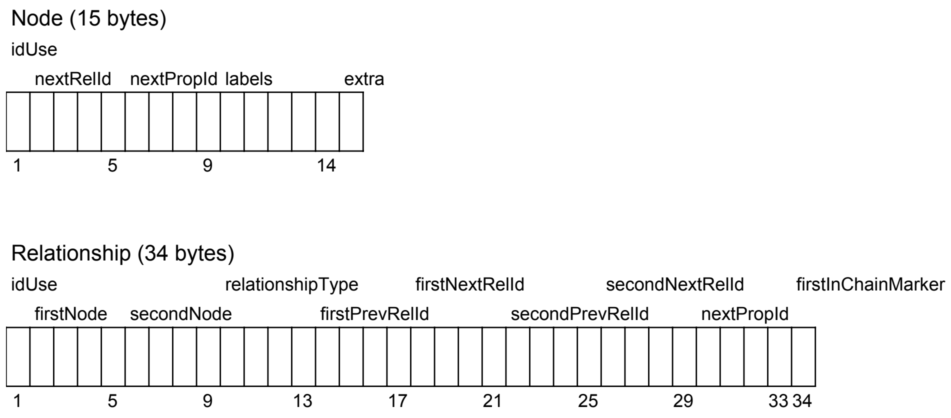


Figure 2.7: Neo4j node and relationship store file record structure, as described in [23]

properties. There are three different constraint types available for Neo4j as mentioned in [25], namely:

Unique property constraints

The unique property constraint corresponds to the UNIQUE constraint in SQL which enforces one specific property to be unique in every node of with the given label. However, the difference between the constraint in Neo4j and in SQL is the fact that using the unique property constraint in Neo4j always creates an index on the property that is desired to be unique [25].

Property existence constraints

The property existence constraint ensures that all nodes of a given label type have to have given properties. This means that one can not remove the property from a node nor create nodes of that specific label without the property that is constrained. This constraint therefore corresponds to the rule in SQL is that one can not insert into a table without specifying a value for all columns [25].

Node key constraints

The node key constraint property is a combination of the other constraints mentioned above. It enforces a node of a given label to one or more properties for every node of that type that is created. It also enforce uniqueness of one or several properties [25].

2.3.5 Graph Query Languages

A graph query language, similar to the Structured Query Language described in section 2.2.3, is a collection of operators used to query data from a database. Since SQL was introduced many years before any graph query language, many general concepts and approaches in SQL is directly transferred or similarly made in the graph query languages [23]. The lack of standardization has also led to several immensely different implementations and frameworks. However, some essential common key functionality is provided by a typical graph query language [35]. Then, grouped into the categories; adjacency, reachability, pattern matching and summarization, the core functionality can be described as follows [1] [15]:

Adjacency queries: The node/edge adjacency queries determines that two nodes are adjacent when there is an edge between them. Similarly, two edges are adjacent when they share a common node.

Reachability queries: The reachability queries determines whether two nodes are connected by a path. Several studies and research exists regarding the computational complexity for reachability queries.

Pattern matching queries: The pattern matching queries finds all subgraphs within a graph. As an attractive field of study in database theory, it deals with two problems: the graph isomorphism problem and the subgraph isomorphism problem.

Summarization queries: The summarization queries determines special functions that allows for summarizing and assembling on the query results. For example, the aggregate functions is included, such as: average, count, sum, minimum, maximum, etc.

Subgraph matching is the task of finding subgraphs within a graph. Supported by all graph query languages it is, in a sense the simplest form of graph query [35]. However, belonging to the pattern matching type of queries, the subgraph isomorphism problem is of a special interest. Finding efficient algorithms or approaches for the most essential graph query task is of great interest. Thus, much study has been done and the task is known to be NP-complete [35].

Cypher

Cypher is a graph querying language that is used for querying Neo4j databases. It is a declarative language developed mostly by Neo4j through the openCypher Project [27]. The fact that Cypher is declarative means that the data to be fetched is specified, instead of specifying how to actually fetch the data. The Cypher Query Language has many similarities with SQL, such as having the same abilities to manipulate and fetch data, such as changing data, fetching data and creating ”data schemas” [9]. The language is built up using the keywords. Some of the most common keywords are specified in table 2.3, however, there are numerous more keywords in Cypher that are not as common.

The Cypher Query Language has different patterns to represent different data entities in queries. Nodes are represented using parentheses, this means that in a Cypher query `(n)` represents a node called *n*. Node labels can also be specified in a query, to specify that *n* has the label Person, it is instead specified as `(n:Person)` [13]. Relationships are described using brackets, where the relationship type can be specified inside the brackets, e.g. `[:KNOWS]`. Furthermore, directions of relationships are specified with an arrow pointing in the direction in which the relationship goes [13]. Using the terminology above, we can specify a pattern that describes that Person *a* knows Person *b* as follows:

```
(a:Person)-[:KNOWS]->(b:Person)
```

MATCH	Corresponding to the SELECT statement in SQL, specifies what nodes to traverse in the query
CREATE	Used when inserting data into the graph
WHERE	Specifies conditions on the queried data, e.g. only fetch names that start with an A
ORDER BY	Sorts the output in ascending or descending order
RETURN	Specifies what is desired as an output from the query
LIMIT	Limits the data returned from the query to a given number of entries

Table 2.3: Some of the most common Cypher keywords

Furthermore, properties can also be specified when querying data. When it is desirable to specify properties for a Node, it is done within curly brackets inside the Node brackets, e.g. `(a:Person {name: 'Jan Zubac'})` represents all person nodes where the name is equal to ”Jan Zubac”.

An example using the syntax introduced above as well as using some of the keywords in table 2.3 can be written as follows:

```
MATCH (
   (m:Movie)-[:RATED_BY]->(:Person {name: 'Victor Winberg'})
)
WHERE m.releaseYear > 2000
ORDER BY m.movieName
RETURN m
```

The query above returns all movies released after the year 2000 that have been rated by the `Person` ”Victor Winberg” and sorts them by movie name is ascending order, given a database containing users, movies and user ratings.

2.3.6 NoSQL

NoSQL is a term that describes a type of databases that have increased in popularity lately. It stands for "Not only SQL" and is a collective term to describe databases that are not built entirely on the regular relational database tables. NoSQL databases are designed for storing large amounts of data and are used when the data to be stored cannot effectively be represented using relational models. These types of databases generally do not use Structured Query Language (SQL) to query the data. Instead, many NoSQL vendors have developed query languages of their own that are used to query the unstructured data in the databases [24].

Several NoSQL databases have been developed by companies that process enormous amounts of unstructured data, such as Facebook's database Cassandra and Google's Bigtable which are both non-relational databases. Other popular NoSQL databases are MongoDB, Neo4j and Voldemort [24]. One of the major differences between relational and non-relational databases is the fact that relational databases are designed to conform to the ACID properties described in section 2.2.1. NoSQL databases are limited by the CAP theorem which states that network shared data only can fulfill two out of three properties consistency, availability and partition tolerance [11].

2.4 Database Benchmark

It is very hard to conduct proper benchmarking as there are many factors to take into consideration. Karl Huppler mentions five properties that one should strive to achieve to some degree to have a good benchmarking process. The properties are the following [16]:

Relevant: The data presented in the benchmark provides information relevant to the comparison.

Repeatable: If the benchmark is conducted again in the same conditions as the previous time it was conducted, the results will be very similar to each other.

Fair: All entities being compared in the benchmark should be able to partake in the benchmarking with equal prerequisites

Verifiable: The results are presented in a way in which the validity of the results can be assessed.

Economical: The sponsor/stakeholders of the benchmark can run it without problems.

To fully achieve all properties at once is nearly impossible. However, one should strive to achieve all properties at least to some degree [16]. This thesis will not emphasize fulfilling the economical property as it is not relevant in this study. The study will strive to fulfill all other properties as much as possible.

2.5 Customer Relationship Management

Customer Relationship Management (CRM) is very hard to define. It is a very broad term that can represent many different elements. Generally, CRM is defined as customer centered technology-based tools. Adrian Payne and Pennie Frow [28] define CRM in the following manner: "CRM is a strategic approach that is concerned with creating improved shareholder value through the development of appropriate relationships with key customers and customer segments. CRM unites the potential of relationship marketing strategies and IT to create profitable, long-term relationships with customers and other key stakeholders." [28]. There are various types of CRM systems, namely Strategic CRM, Operational CRM and Analytical CRM [5].

Operational CRM focuses on automation of customer-centric operations such as sales, marketing and services. This means that services such as help desk interaction is integrated in a common CRM system to make it easier for customers to get in contact with services that can help them with various problems. Operational CRM also focuses on sales force automation (SFA) by utilizing technology to keep track of sale trends etc. This data is also used by an Operational CRM system to automate marketing by providing campaigns to its customers by analyzing the sales data [5].

Analytical CRM, on the other hand, focuses on handling data to determine which customers are the most valuable, which customers are more prone to accepting certain deals and so on. This enables users of the CRM to easily detect which customers to focus their attention on for upcoming sales, and which customers need another approach [5].

Lastly, there is Strategic CRM which is mainly used to get an understanding of the needs of the customers to be able to conform to as many needs from the customers as possible. As a result of this, strategic CRM systems also aim to help to create and deliver value to their customers by defining the business and customer [17].

2.6 Related Work

This section will list other work that has been made about the difference between graph databases and relational databases. It will describe how this information was relevant to our work.

In [33], Vicknair et al. make a comparison between a Neo4j database and a MySQL database. The study was conducted to benchmark how long it takes for a set of queries to be executed for the different databases. Several different databases were used, such as databases containing only integers, databases containing 8kB random strings and databases containing 32kB random strings. Different database sizes containing the mentioned data types were created to see how the number of nodes affected the execution time of the different queries. The work of Vicknair et al. is useful in this report due to the fact that it gives a great insight on how the size of the database impacts the execution time of a Neo4j database, which is similar to what is to be studied in this master thesis. However, the data used was randomly generated artificial data which implies that it was not structured in a way in which a database would normally be used, e.g. as a social network so the results might differ from what is observed when a CRM system is evaluated. Their conclusion was that graph databases performed better at both structural directed acyclic graph traversal queries and in full-text searches.

Another piece of work that is related to our work is the research conducted by Batra and Tyagi in [3]. This paper also compares the execution time of queries for a Neo4j database and a MySQL database. Similar results to [33] are presented, however, the differences between the different databases are far more distinguishable due to the results being represented as graphs. This research paper is interesting to us for the same reason as the research conducted by Vicknair et al., namely that it gives us more data on execution time difference for the different database types. Results from Batra and Tyagi's work concluded a similar conclusion as the results attained in the research conducted by Vicknair et al., namely that graph databases performed better on their traversal and nested queries.

Webber et al. [29] have written a book that describes graph databases and Neo4j in detail. In the book, the basic properties of graphs such as the formal definitions of nodes and edges are very well explained. Furthermore, the book contains in depth explanations about the underlying structure of the Neo4j graph database, such as how the data is processed and stored in the Neo4j graph database which was very useful to us when evaluating why or why not Neo4j was useful for storing and querying a CRM system. The insight in storage and processing mentioned in this book also facilitate an understanding of the differences between Neo4j and MS SQL. Moreover, the book also contains comparisons between SQL and Cypher queries with explanations to why some queries are hard to write in regards to both database types (relational and graph). These comparisons are a useful as a base for both the performance and the complexity analysis as they give insight into the difficulties of both databases.

Chapter 3
Method

The work was conducted by performing an empirical study on two databases built with the database structure and schema that exist in Lime's CRM system. The relationships between each data entity was modelled to be structured as in the real system. This means that all relationship between entities are correct, however, we modelled the amount of relationships from each node to be constant to an average of the amount of relationships we observed in the real system. Therefore, instead of some person owning 3 documents and some owning 15, every person owns the same amount of documents and so on. The databases were built with randomized data values. In practice, this consisted of us comparing the relational database that Lime currently uses, namely Microsoft SQL Server, against the graph database, we have chosen, Neo4j with our data values. It is worth noting that the version of Neo4j used was the Community Edition, meaning that the comparison in this thesis is between a free-to-use software (Neo4j) and a paid software (Microsoft SQL Server). However, the major differences between the Enterprise and Community Edition of the Neo4j client are features that are very useful in a database that is to be used in a real system such as additional constraints and further customization options. Therefore, execution times presented in this thesis are not that affected by the fact that a free and paid software are being compared. Following is the steps needed to make the comparison:

1. Analyzing Lime's current CRM system to determine what type of queries are called and which operations are common in the system.

2. The queries that are to be benchmarked were written so that the Microsoft SQL Server and Neo4j queries return the same data. These queries were then compared from a performance and complexity standpoint for the different databases.

3. The performance was compared by comparing CPU time used by the process when performing the Microsoft SQL Server and Neo4j queries. Furthermore, we analyzed why the queries are slow, fast or perform equally for the different databases. Thus, answering problem statement 1 - "How does the performance of graph databases compared to

relational databases differ, when used in a large complex CRM database system, and when?”

4. The lexical complexity of the queries was to be compared by performing a lexical analysis to split the queries into tokens. Thereafter, we intended to evaluate how a similar comparison had been done in other scientific papers. However, during our project we realized that this was a subject that was not touched upon in scientific literature. Furthermore, there was not enough time to conduct a user study to determine the perceived complexity from many user’s perspective. Therefore, it was decided to not include this study in the report as the results can not be backed by any scientific research. As a consequence of this, we were not able to answer research question 2 in this thesis.

5. A working prototype, with front-end, back-end and configurable database systems was built to be able to run the queries many times and quickly visualize the results. The purpose of the prototype is to demonstrate noticeable differences for the end users live.

3.1 Benchmarking

During the thesis work period, several different methods were tried to achieve a benchmarking that is as good and fair as possible. One of the most important aspects of the benchmarking process conducted in this thesis is the fact that all queries, for both databases, are called with the same random sequence of value parameters to avoid query caching. The reason for this is to see how well the two databases perform without caching, as they both perform extremely well and are close to indistinguishable when the queries are cached. To be able to do this, a function was written in Python that randomly selects an existing entry for a specified column in a specified table so that the queries actually always return data and do not try to query entities not existing in the database.

However, some of the more complex queries were benchmarked with caching as well to see how the performance of the databases differs when the queries are cached. The reason for only analyzing and documenting the caching effects on the complex queries is, as mentioned above, that the simple queries become close to instant in both databases when they are cached.

3.1.1 Mathematical time measurement functions

To get a broad spectrum of numbers to analyze later on, the mean values and standard deviation values will be calculated for the execution times of the queries. Each query will be benchmarked and executed numerous times. The queries will then be executed with randomized parameters to emulate reality more, in terms of less caching of the queries. The top 5% largest and smallest values will be removed from the data so that the extreme case execution times will be ignored in the calculation. Removing the extreme values will be done so that the results do not get distorted if there are execution times that differ very much from the average, e.g. if the operating system schedules a heavy process when one query is being executed and to not be as affected by a query being cached for the lower execution times. This approach was done in a similar way by Vicknair et al. [33]. They removed 10% of the largest and smallest execution times, however, their sample size was much smaller than ours, so we deemed it fitting to only remove 5%.

3.1.2 Measuring the query execution time

To measure the execution time of the queries specified in section 4.4, the queries will be called through a programming language for efficient calculation of the time measurements. Cypher will be queried through the Neo4j Bolt Driver 1.7 for Python [30]. This python package has functions to execute queries as well as measure the execution time of the queries. In the Neo4j Bolt Driver 1.7 for Python, when a query is executed using the execute() method, several attributes are set to the object that is returned. Some of these attributes will be used for the benchmarking purposes, namely the attributes t_first and t_last which represent the time it took for the server to have the first result ready and the time it took for the server to consume the rest of the results. This implies that t_first + t_last represent the total execution time of the query and this will therefore be used as a measurement of the execution time of Neo4j. Neo4j's official client also operates through the bolt driver. In this client, one can execute individual queries and get the results, as well as the execution time and the time it takes to consume the data. The values of the execution times in Neo4j's built in client tool were evaluated to ascertain that they present approximately the same execution time as the attributes t_first and t_last to ascertain that they are a trustworthy measurement of the execution time. The evaluation showed that the execution time for the queries in Neo4j's official client were the same as the execution times retrieved from the python bolt driver. This indicates that the results from the python driver are accurate if one trusts Neo4j to show the accurate execution times in their own client.

The execution time of the queries on the Microsoft SQL Server were measured in a different way. The queries were executed through the Python library pyodbc [19]. Pyodbc has no built in functions for measuring the execution time. Instead, the method that was used to measure execution time of the Microsoft SQL Server queries was to record the UNIX timestamp [20] before and after each query has been executed and subtracting the execution times to get the time difference. This approach can be considered risky, as it is hard to know how much of the time returned is CPU time and how much is the actual retrieval time of the data. However, there is a possibility to query a Microsoft SQL Server database through the Microsoft sqlcmd utility [22]. This command line tool enables execution of queries through the command prompt and it displays the execution time of the queries executed. A comparison between the execution time given by Microsoft's official tool and the UNIX timestamp method was conducted to determine if there was any notable difference between the methods. The comparison showed that the execution times were very similar, however, one has to take into consideration that the same query being called twice does not always take the exact same time no matter which measurement method is used. Therefore, it is impossible to say if the execution times are completely identical or not.

Chapter 4
Design of Experiments

This section describes more in detail how and why the work was conducted the way it was. For an even more detailed insight in the benchmarking prototype, one can visit a release of the code used to conduct the benchmarking done in this thesis by following the DOI specified in [34]. The GitHub repository also contains instructions on how to reproduce the experiments conducted in this thesis to be able to validate the results.

4.1 Analyzing the Lime CRM System

One of the first tasks in the process of benchmarking Lime's CRM system was to analyze the system. This was done both to get a general understanding of what CRM Systems are and how they are utilized, as well as to get specifics of how Lime CRM works and what some common operations are. This section will describe the process of investigating the Lime CRM to determine how to model a smaller version of the large scale CRM-system that Lime distributes.

4.1.1 Data Relationship Analysis

A key component of understanding a software system that is heavily reliant on data storage is to study what is being displayed in the system and envisioning how the displayed data is stored in a database. In our case, a demo database containing a few hundred data entries was provided to us. This database was a Microsoft SQL Server database, and was therefore imported into Microsoft SQL Management Server where all tables, relationships, keys, constraints and so on were visible to us. The analysis started with identifying all tables present in the database and subsequently noting what columns each table had. Section 4.2 describes how the data acquired in this step was formed into an ER-diagram.

After we got an idea of what data entities were stored in the database, the database was imported into the Lime CRM System to see how the data is displayed in the CRM System that was to be modelled. It was then quickly realized that many of the table and columns stored in the database were there to facilitate e.g. front end development and maintenance, which means that many of the tables were rarely queried for data. These tables were omitted in our model, as the model to perform the benchmark on had to be constrained to not reveal too much information about exactly how the Lime CRM System was implemented. Instead focus on the core entities, which technically could be derived from the Lime CRM Desktop Client as well as the Lime CRM Web Client without knowing the database structure.

To study how the different tables are connected, SQL Server Profiler was used. This program logs the operations conducted on the database that is connected to the profiler. In this case, the demo database was connected to the profiler, so all queries performed on the database when we interacted with the Lime CRM System were logged to later be viewed. Parts of the Lime CRM System containing many data fields from different tables were explored and interacted with to systematically locate queries of relevance to the study, e.g. advanced queries with many joins. The profiling also led to an increased understanding of the database, which was helpful later on.

When the core entities and relationships have been singled out, we created an ER-diagram for us to envision how all entities are connected as a whole and what columns are necessary and which ones can be omitted in the benchmark model. The ER-diagram of the core entities that are used in the databases that were benchmarked can be seen in figure 4.1.

4.1.2 Common Operations

In the benchmark, several queries were analyzed to see how well Cypher performs in comparison to Transact-SQL in a CRM system. As it was not realistic to cover all the queries that are called in the Lime CRM System, a small subset of queries needed to be singled out. The vision was to create realistic queries on our model that represent queries that are often called in the real Lime CRM System. By analyzing the system as well as talking to our supervisor at Lime, it was deemed that the most common operations to perform in the system is to display various data. This means that the most common queries are SELECT queries. As a result of this, a majority of the benchmarking queries consist of retrieving data from the database. When it comes to the data retrieval queries, some of the most common operations is to filter data using some or several of Lime CRM's available filters. Therefore, many of the queries used in the benchmarking will contain filtering in one way or another.

The other very common database operation was to add data to the database, which is why several insertion queries will be benchmarked to see how the different databases perform when inserting data. Lastly, updates are also used somewhat regular in the Lime CRM so there are also some queries that benchmark the efficiency of data updates for the different databases.

4.1.3 Reasonable size for Lime's CRM System

It is of great importance to ensure that the system on which the benchmark is to be performed on resembles the Lime CRM System as closely as possible to guarantee that the results are actually relevant to Lime's CRM system. For the benchmark to be realistic, the size of actual databases that customers of Lime CRM use were estimated. Furthermore, the quantity of each data entity stored in the database was estimated to assure that the proportions in the benchmark database model are close to the ones in the real system. It would e.g. not be a realistic depiction of the real CRM System if our model had 100 documents in it and the real system has 10000 documents. Moreover, it is important to mention that many of Lime's customers have tailored solutions, which means that the Lime CRM is not the same for all customers. Moreover, some companies that are users of Lime CRM are naturally larger than others. As a result of this, the study will focus on emulating the Lime CRM system for the larger companies, as they are more relevant to the study. This means that the result might be applicable for all companies that use Lime CRM.

For us to estimate the number of data entities stored in Lime's CRM System, a database that Lime uses in-house was provided to us. In this database, the number of companies, deals etc was observed to determine an estimate of what a reasonable size for the largest database to be benchmarked is. The entities that had most entries in the database were histories, which were a factor of 10 more than any other entity, this naturally led us to having history entries make up a large portion of the database.

To evaluate how the databases perform for different database sizes, three different databases were built. The smallest graph database had 35,960 nodes and 129,100 relationships, which corresponds to 35,960 row entries in the Microsoft SQL Server database. The medium graph database had 342,510 nodes and 1,272,500 relationships which corresponds to 342,510 row entries in the Microsoft SQL Server database. Lastly, the largest graph database had 3,460,510 nodes and 12,800,500 relationships which corresponds to 3,460,510 row entries in the Microsoft SQL Server database.

4.2 Database Schema

The database schema was designed based on a basic Lime-CRM setup, with some of the essential CRM-entities included. This means that many CRM-entities that would not affect the benchmarking will be omitted, e.g. due to them not introducing any further complexity and depth to the benchmarked queries. Furthermore, some attributes were also omitted in the ER-diagram as the data properties themselves are not the focus of the benchmarking, due to the benchmarking not being affected by many columns that are not queried anyway. As a result of this, the relationships in the ER-diagram are the most important. The ER-diagram of all the data relationships in our miniature CRM System can be seen in figure 4.1.

The `office` and `coworker` tables store the information about `coworkers` at "our" company and at what `office` they are stationed at. Whereas, the `company` and `person` tables instead store information about the companies that are Lime's customers and their employees. Then we have the `deal`, `document` and `history` tables which store the information that a CRM-system organizes with the previous given tables. A `deal` could be a pending project that could deliver a `value` with a given `probability`. These `deals` then have a lot of information between the different two parties, `coworker` and `person`, from different events. These events are stored as a `history` usually with a `document` of some sort. A `history` could be a call, meeting, email, etc.

When it comes to the relationships presented in figure 4.1, the relationship between a `history` and `deal` entity conveys information about a specific event that was part of the `deal`, for example a phone call. The relationships between the `document` entity and the `history` document represents that a `document` was part of the given `history` e.g. that

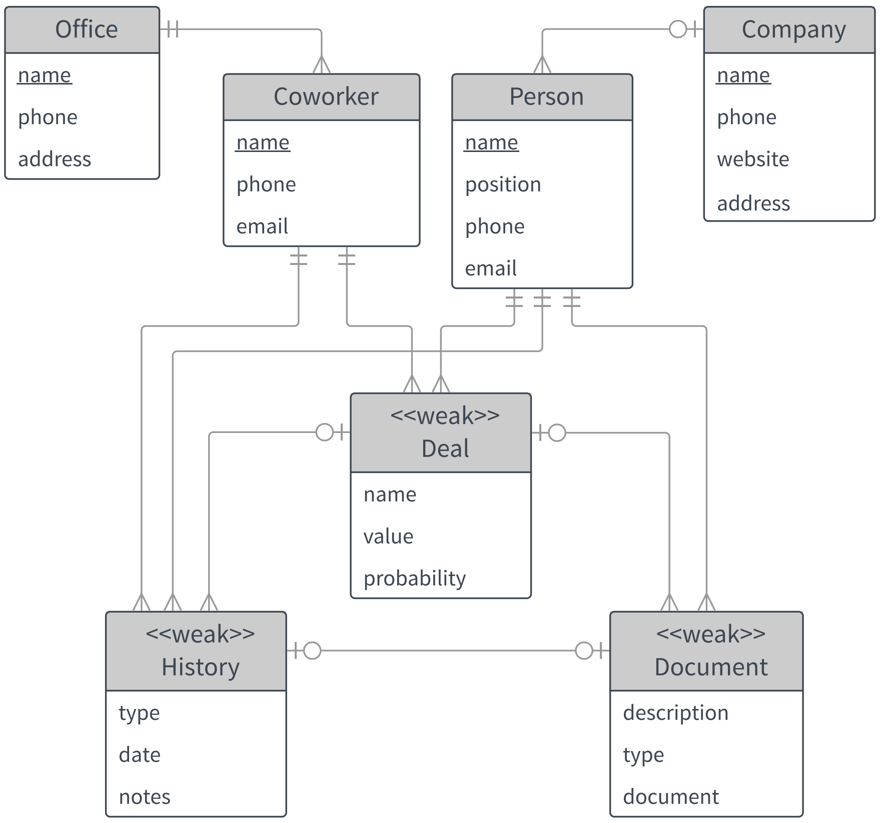


Figure 4.1: An ER-diagram overview of the core CRM-entities

an excel document was written after a visit to a customer. The relationships between a `coworker` and a `deal` corresponds to the fact that the given `coworker` is the salesperson for that `deal`. The relationship between a `person` and a `deal` is very similar. It corresponds to the fact that a `person` is responsible for a `deal`. The connection between `coworker`, `person` and `history` represent that the person/coworker took part in that history entry, e.g. coworker A spoke to person B on the phone regarding a deal, forming a history entry.

4.3 Building the databases

To be able to build a database similar to the real databases used by Lime's customers in the Lime CRM, large amounts of fake data had to be generated. The data generation was done by utilizing the python package faker [8]. With faker, it was possible to randomly generate all data fields needed, such as addresses, zip codes, first and last names and so forth, using the same amount of relationships and structure as in Lime CRM. Therefore, providing us with a real-life scenario with anonymized data. With the use of setting specific seeds, it was ensured that the randomly generated data was the same every time. Hence, both the graph database and the SQL database was populated with exactly the same data.

4.3.1 Microsoft SQL Server

A very important part of building an efficient SQL database was constructing indices on heavily queried data. However, in Lime CRM system, it is highly unpredictable which data will be queried, meaning that it is hard to decide what data to construct indices on. As a result of this, it was decided that for the SQL database, indices would only be created on foreign keys, to make joins faster, and on primary keys. A reason for this constrained usage of indices is also the fact that insert operations are common in CRM systems, and many indices make inserts slower as the index needs to be moved and rearranged whenever additional data is inserted in the database.

As a result of this we created seven tables from the seven given entities in figure 4.1. In SQL, a table describes their relationship to another table by using foreign key constraints. I.e. describing the relationship between a row in the persons table to a row in the companies table is done by a foreign key constraint in the persons table, named company id.

4.3.2 Neo4j

As explained in section 2.3.1, graphs have different properties than relational databases. To construct a graph database from a relational database, each table in the relational database corresponds to a node label. The properties of each node consist of the columns of that table. So to convert the relational database into a Neo4j graph database, node labels for each table will be created, e.g. Person, Company etc. Furthermore, these nodes will be given properties corresponding to each column in the relational database, except for the columns that represent foreign keys that reference other tables. In Neo4j, this will instead be represented as relationships.

When it comes to indices in graph databases, it is not possible to create an index on relationships, which would be the equivalent of creating indices on foreign keys in SQL. As a result of this, indices were only created on the primary keys, namely the id of every node.

Instead of e.g. storing a company id in the Person node, this relationship was instead labeled as `WORKS_FOR`. Hence, the relationship entity also describes in text what it represents, whereas in SQL one has to analyze the two tables that are connected with a foreign key to figure out what the relationship means. The relationships that were used to construct all the type of edges corresponding to the relational database foreign keys are the following:

WORKS_AT: Represents that a Coworker works at a Company

RESPONSIBLE_FOR: Represents that a Person is responsible for a Deal

OWNS: Represents ownership of a Document

ATTENDED: Represents that a Coworker and/or Person is part of a History entry

ATTACHED_TO: Represents that a Document is attached to a Deal

SALESPERSON_FOR: Represents that a Coworker is in charge of a Deal

PART_OF: Represents that a Deal is part of the History for a given entity

4.4 Database Queries

During the benchmark analysis, several different operations were performed on the Neo4j and Microsoft SQL databases. As delete queries are rarely executed in CRM systems, it was decided that delete queries would not be a part of this study. Instead, the thesis focused on studying insertion, retrieval and updates on data. These different queries used will be described further below. The queries that were called during the benchmarking were sought out to be diverse in many ways. The operations used in Lime CRM that we will evaluate when benchmarking are the following:

- Joining few and several tables
- Filter using indexed and non-indexed columns/properties
- Filter on close and distant relationship-endpoints
- Filter using floats, integers, strings and dates
- Filter using few and several operators
- Nested filtering
- Querying small, medium and large size tables
- Insertion into small, medium and large size tables
- Updating a single property
- Updating a property in many rows/nodes

The reason for this is that these operations cover a lot of the functionality of Lime CRM and they also cover many operations of the databases.

4.4.1 Data retrieval

Data retrieval queries are very important in any system that heavily relies on data storage. The reason for this is that whenever some data is to be displayed in the system, the data has to be retrieved from the database. As a consequence of this, a large part of the benchmarking consisted of data retrieval queries. To get a good overall understanding of the differences between Neo4j and Microsoft SQL Server, several different types of queries had to be called in order to see the performance for a specific query.

It is important to mention that our databases contains no null values, which means that all tables/nodes have non null values for all columns/properties. As a result of this, the Microsoft SQL query optimizer creates the same query execution plan regardless of the join type. This means that inner joins and outer joins are treated in the exact same way by the SQL server. Therefore, it was decided that left joins were to be used in all queries, even if it in most databases would not be fitting to construct the queries using left joins.

For the data retrieval queries, most tables should be queried to see how well Microsoft SQL handles larger and smaller tables as well as if there is any difference in Neo4j when retrieving node-types that there are a lot of. Furthermore, different depth of queries was used to see when the databases start to differ in performance. In one of the queries, namely, query A.12, all persons working at a given company are retrieved. This query has distance one, meaning that it is not sufficient to just go through all persons to find the query result, instead companies and persons have to be analyzed together by either joining tables (Transact-SQL) or by analyzing relationships (Cypher). Finding everyone working at a specific company is a natural key component of a CRM system, therefore we deemed it important to include this query in the benchmark.

Moreover, it was tested how Neo4j and MS SQL handle queries that are dependant on retrieving data from many tables combined, or in Cypher, data that can not be retrieved without analyzing many relationships. The reason for this is that the data entities in Lime CRM are heavily connected to each other, so most operations that are true to the functionality of the real Lime CRM System cannot be made without analyzing the performance when several tables have been joined together. Query A.5 where 5 tables are joined to be able to retrieve all entities attached to history data entities with specified type represents this type of query. Furthermore, this query also returns large amounts of data, which is an interesting point of analysis in regards to performance of the databases.

A very common feature in Lime CRM is to utilize the so called quick filter, according to several developers at Lime CRM. This filter allows the user to enter an arbitrary input, thereafter, when the data is queried, it is checked whether the given input exists in any of the columns shown in the current window of Lime CRM. Two queries with different complexity were tested. Firstly, query A.13 which only tests how the system handles the Lime CRM quick filter. Secondly, query A.4 tests the quick filter as well as evaluating how the databases perform when combining the quick filter with a feature called ”transfer”. For transfers, specified columns in one tab, e.g. deals can be transferred to e.g. coworkers to see all coworkers that are part of the selected deal(s).

Lime CRM also has another type of filtering, where only one specified column is filtered after a given criterion. This feature is evaluated using the queries A.2 where only histories before a given date are selected as well as only histories with a specific type to see how well the databases handle basic filtering. Additionally, the queries A.1 also evaluated how well the databases handle filtering in Lime CRM, more specifically, how the databases handle the filter option 'Begins with', where one can specify to only select those data entries in a given column that begin with a given combination.

To further analyze the filtering function mention in the previous paragraph, query A.3 was chosen to evaluate how the databases handle filtering of floats. This operation corresponds to filtering the deals table to only display deals above a given probability.

Lastly, query A.6 was chosen to see how the databases handle heavy filtering of data, in this case choosing the deal with highest probability in the system. Thereafter, this data entry is used to query deals that are related to the coworker that is responsible for the first deal.

4.4.2 Data insertion

To evaluate the performance of insertion for the different databases, we wanted to create data insertion queries with different depths to see how much indexes affected the execution times in the different databases as well as to get an understanding of the performance of insertion queries for MS SQL and Neo4j. One of the reasons that we chose to evaluate insertion is this way is the fact that indexing negatively impacts insertion. Therefore, queries with different amount of indices were benchmarked to see how affected the two databases were by this. However, as inserts only operate on one table, except for creating references to foreign keys/creating relationships, data was inserted in tables/nodes with varying relationship complexity. In query A.9, data is inserted into the histories table, which has five indices, and therefore the insertion is more complex as more indices have to be relocated when new data is inserted. Query A.7, inserts data into a table with two foreign keys, and query A.8 inserts data into a table with only one foreign key, and therefore only index on one row.

When it comes to the insertion queries in Cypher however, they are a lot more complex by nature. The queries update the graph with the same data as the SQL queries. The major difference between Cypher and Transact-SQL is that relationships are automatically created when inserting data that is constrained by foreign keys, due to the fact that the schema enforces the data that is being inserted to conform to the data existing in the table that is being referenced. In Cypher, there is no easy way of creating a node and its relationships using one query, therefore, the MERGE statement has been used, as seen in query A.9, A.8 and A.7. The MERGE-statement creates a node and/or relationship if it does not exist. As seen in query A.9, using MERGE, nodes that we want to create relationships to are found, and the node that is to be inserted is created (the history node is this case), thereafter, all relationships between the new node and the existing nodes are created. This is naturally a more complex process than insertion in Transact-SQL, as we enforce Cypher to find existing nodes before it can create relationships between them and the newly created node.

4.4.3 Data updates

As mentioned in section 4.1.2, update queries are not used as much as insertion and data retrieval in Lime CRM. Therefore, we decided to only test two update queries as there are is not that much update functionality in Lime CRM. Query A.11 was benchmarked to evaluate how the databases perform for the most basic update operation, namely updating a single property in a single node/row. The performance of update queries can be impacted by having multiple indices, as they essentially both delete data, which enforces updates of indices and thereafter inserts the new data which also enforces updates of indices. To test how the databases perform in this regard, query A.10 was chosen to represent an update query that updates a non-indexed property in many nodes.

4.5 Execution Plans

To further analyze the benchmark execution times we used the built in query profiler in Neo4j and the built in query execution plans that are provided in Microsoft SQL Management Server for MS SQL. These query planners were used to distinguish how the different databases actually execute the queries, e.g. what is going on in detail to be able to determine why the queries in one of the databases is more time consuming than the other for particular queries. Therefore, the use of execution plans is good to analyze and compare where and when it occur. Here we will give a brief overview of how the execution plans work.

Neo4j uses statistical information regarding number of nodes with certain labels, relationships by type, etc. to determine how the query should be executed. Then the execution

$$\begin{tikzcd}
ProduceResults \\
Projection \arrow[u] \\
Limit(10000) \arrow[u] \\
Filter(Person) \arrow[u] \\
ExpandAll(WORKS\_AT) \arrow[u] \\
NodeIndexSeek(Company) \arrow[u]
\end{tikzcd}$$

(a) Neo4j

$$\begin{tikzcd}
Select & \\
Top(10000) \arrow[u] & \\
NestedLoops(Join) \arrow[u] & \\
NestedLoops(Join) \arrow[u] & IndexSeek(Company) \arrow[ul] \\
IndexSeek(Person) \arrow[u] & KeyLookup(Person) \arrow[ul]
\end{tikzcd}$$

(b) MS SQL

Figure 4.2: Execution Plan for Query A.12.

planner uses a number of operators to perform the execution of the query [26]. For example in figure 4.2a it uses:

NodeIndexSeek - finds the `company` node with the specific *given* index

ExpandAll - traverses all incoming `WORKS_AT` relations from the `company` node

Filter - filters each row to only contain `person` nodes

Limit - returns the first *10000* rows

Projection - evaluates expressions and produces a row with the results thereof

Produce Results - prepares the result to be consumable by the user

MS SQL Server uses a so called Cost-based Execution Plan that tries multiple different execution methods until it ends up with the most optimal execution plan with the least possible cost [10]. For example in figure 4.2b it uses:

Select - contains a lot of useful information about the plan as a whole

Top - passes the first *10000* rows

NestedLoops - combines the result of the two operations below:

- **IndexSeek** - finds the `company` name with the specific *given* index
- **NestedLoops** - combines the result of the two operations below:
    * **IndexSeek** - finds the `person` ids with the given `company_id`

    * **KeyLookup** - pulls additional data, not held in the nonclustered index

What we want to receive in this query is the `persons` working for a specific `company`.

For this example, MS SQL would store the `persons` and `companies` in two separate tables with a clustered index on their primary keys, similar to a phone book each table is sorted and indexed page by page. Furthermore, the `company_ids` in the `persons` table is separately stored in a non-clustered index, similar to an index in the back of a book referencing to a certain page. From a data-centric point of view MS SQL first seeks using the non-clustered `company_ids` index for finding all `person` ids. Then, by looking them all up in the clustered index we receive the additional data. Lastly, it will combine the results from these operations with the clustered index seek of the given `company` id.

4.5.1 Big O notation

Calculating the Big O notation of a query in Neo4j and MS SQL can be done by looking at the execution plan. Given the example in figure 4.2, we are given two tables; persons and companies where the persons table is 10 times larger than companies. Assigning n as person rows and m as company rows, we can calculate the big O notation of the two queries.

MS SQL uses the b-tree structure of the index to seek directly to matching records where `company_id` is the given company id. This operation is executed one time and therefore takes $O(\log(n))$. Then we do a key lookup for each of these values k, giving us $O(k \log(n))$. Then we join these values by performing an index seek on the company with given `company_id`, taking $O(\log(m))$ time. The total big O notation is therefore:

$$O(\log n) + O(k \log n) + O(\log m)$$

Neo4j uses the b-tree structure of the index to find the `company` node with given index, taking $O(\log(n))$ time. Then, we perform expand and filter each evaluating a small amount of relations/nodes k, that with enough relationships should continuously be a low value, with the complexity of $O(2 * k)$. Then, we limit and perform a projection that evaluates each incoming row for each returning argument g (worst-case < 100), giving us $O(g * k)$. The total big O notation is therefore:

$$O(\log n) + O(2 * k) + O(g * k) \xrightarrow[n,k \to \infty]{} O(\log n + k)$$

Chapter 5
Evaluation

5.1 Results

The results show the execution times for the two databases under comparison on multiple different queries explained in section 4.4. Most of the queries were run 250 times, except for query A.13, query A.5 and query A.4, calculating the mean and standard deviation values. Query A.13 was run 10 times and query A.5 and query A.4 were run 50 times. The reason for these discrepancies in the number of executions is the fact that python times out if these queries are called more times, due to them taking to long to complete. These values are then plotted onto bar charts comparing Neo4j and MS SQL. Finally the expression complexity is compared for the same queries. The databases small, medium and large in the bar plots correspond to the databases explained in further detail in section 4.1.3. All queries that are part of the results are described in detail in section 4.4.

5.1.1 Equipment

When performing a benchmark, it is very important to mention what hardware and software the benchmarking process was conducted on. The reason for this is so that someone conducting the same benchmarking has a reference point for their results. If the hardware and software specifications are available, it might be more clear why the results differed.

The benchmarking was conducted on a Dell Ultrabook equipped with an dual core Intel Core i7-5600U CPU with the max clock speed of 2594 Mhz. Furthermore, the computer has 4 logical processors and 16 GB of RAM. It is also important to mention that the benchmarking is run on Windows 10, as results might differ between operating systems as scheduling etc. might be different. If one is interested in the full list of specifications for the computer used, it is a Dell Latitude E7250.

5.1.2 Performance Benchmarks

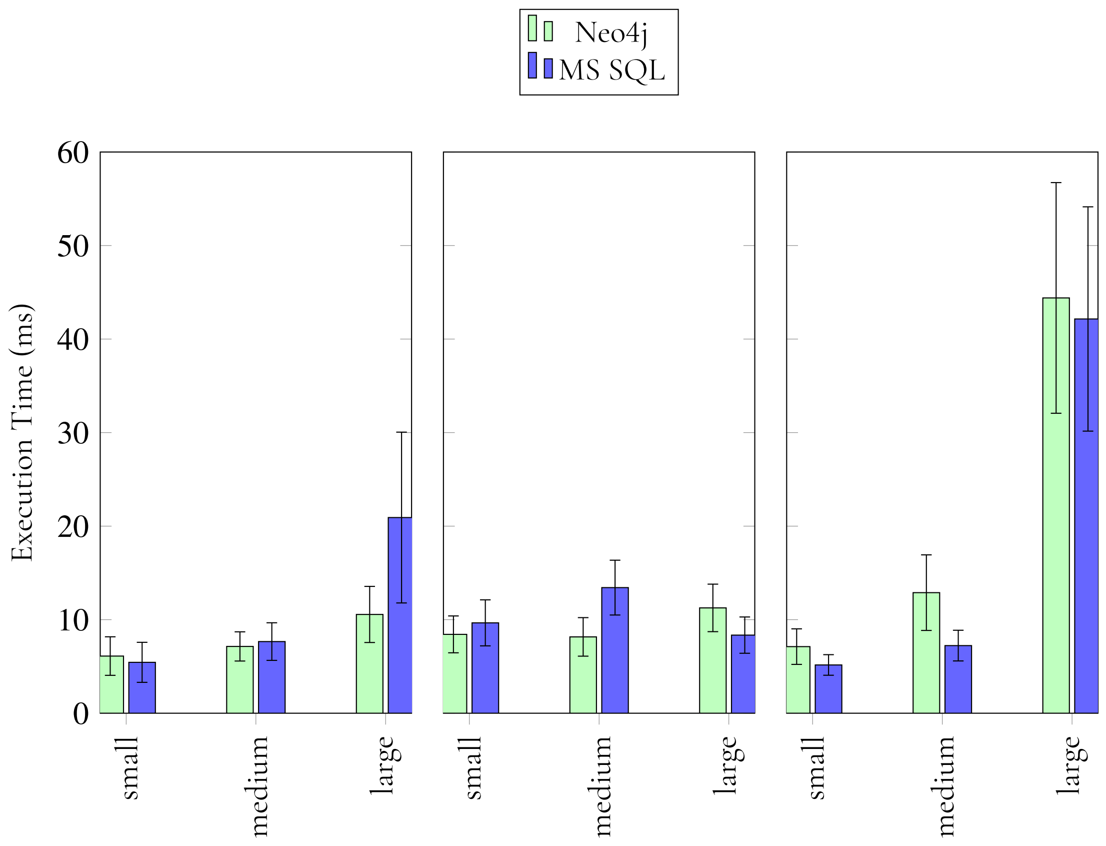


Figure 5.1: Filtering queries

The first comparison chart, seen in figure 5.1, are a set of three relatively non-complex queries. To the left we execute a query with three joins that filters out the results on the exit nodes. In the middle we increase with one more join but instead filter on the smaller table connecting the tables, on an indexed column. For all three queries in figure 5.1, Neo4j and MS SQL are very similar in performance when the smallest database is queried. For the medium database, there are a few performance differences in Neo4j and MS SQL. Neo4j performs better for query A.1 and MS SQL performs slightly better for query A.3. For the large database, however, Neo4j performs better for query A.1 and MS SQL performs better for the other two queries.

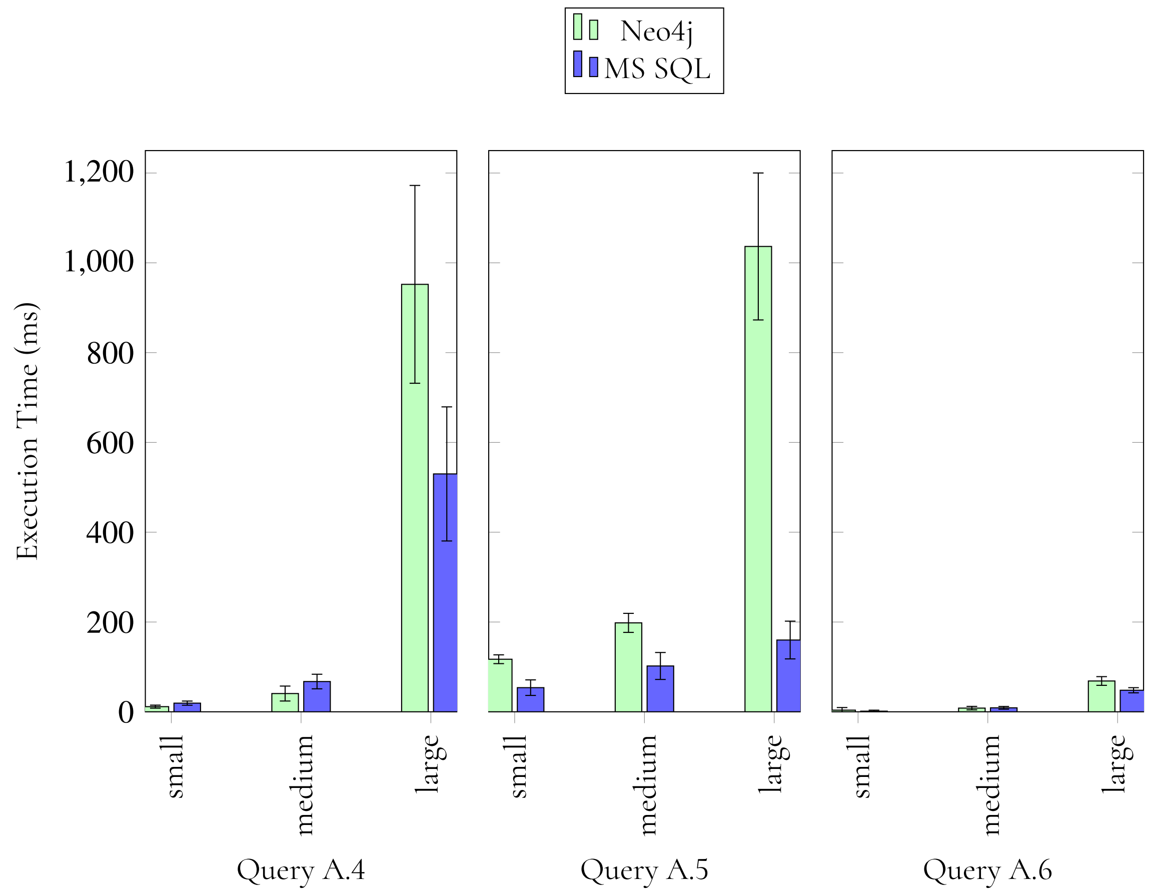


Figure 5.2: Queries with a long execution time

The second comparison chart, seen in figure 5.2, are a set of three more complex queries, in the sense that the queries have a longer execution time. To the left we execute a nested **`WHERE`** clause with depth of two. In the middle we execute four joins and filtering on a non-indexed column in the big table. To the right we once again execute a nested **`WHERE`** clause filtering on a non-indexed column. The conclusions drawn from the results are that that MS SQL performs far better than Neo4j in all cases except for the medium database in query A.4.

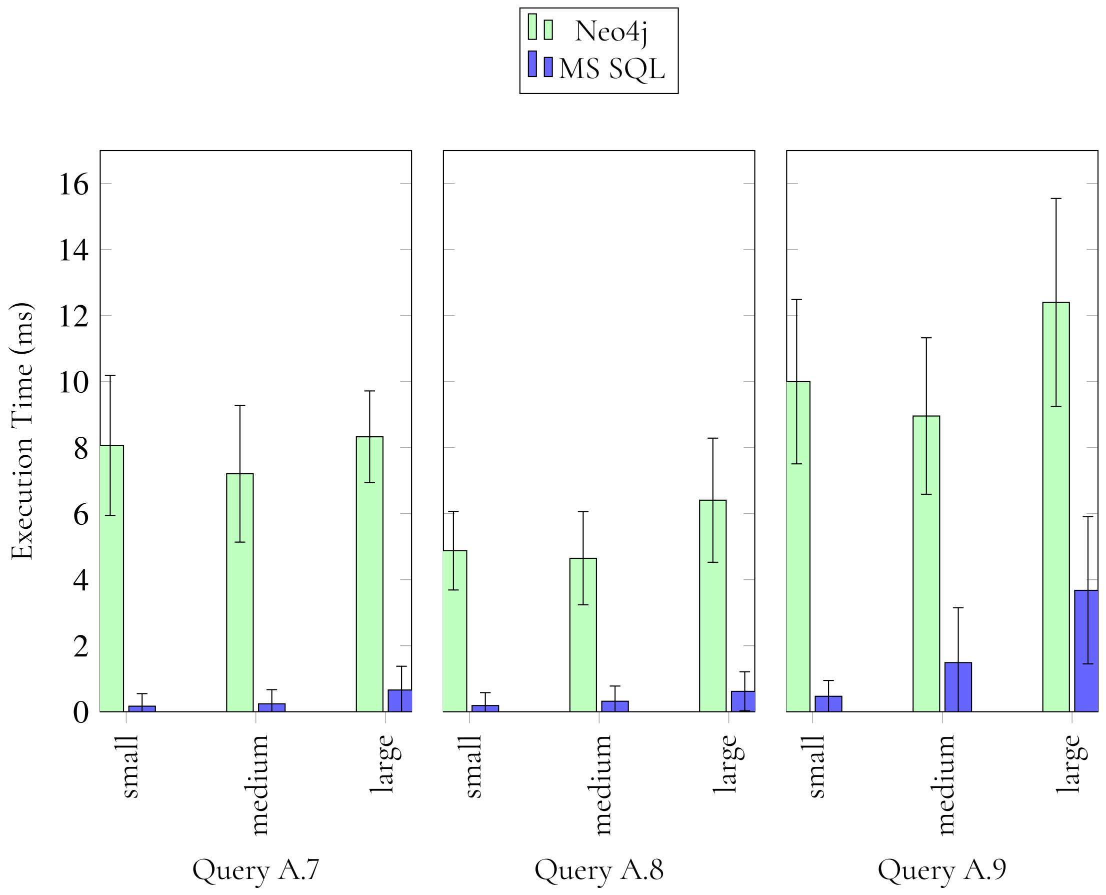


Figure 5.3: Insertion queries

The third comparison chart, seen in figure 5.3, are a set of some simple insertion queries. We insert into a small (left chart), medium (middle chart) and big (right chart) table. The conclusions drawn from these results are that Neo4j once again performs much worse than MS SQL, which was expected. In this figure, it can also be seen that an increase in indices gives an increased insertion time.

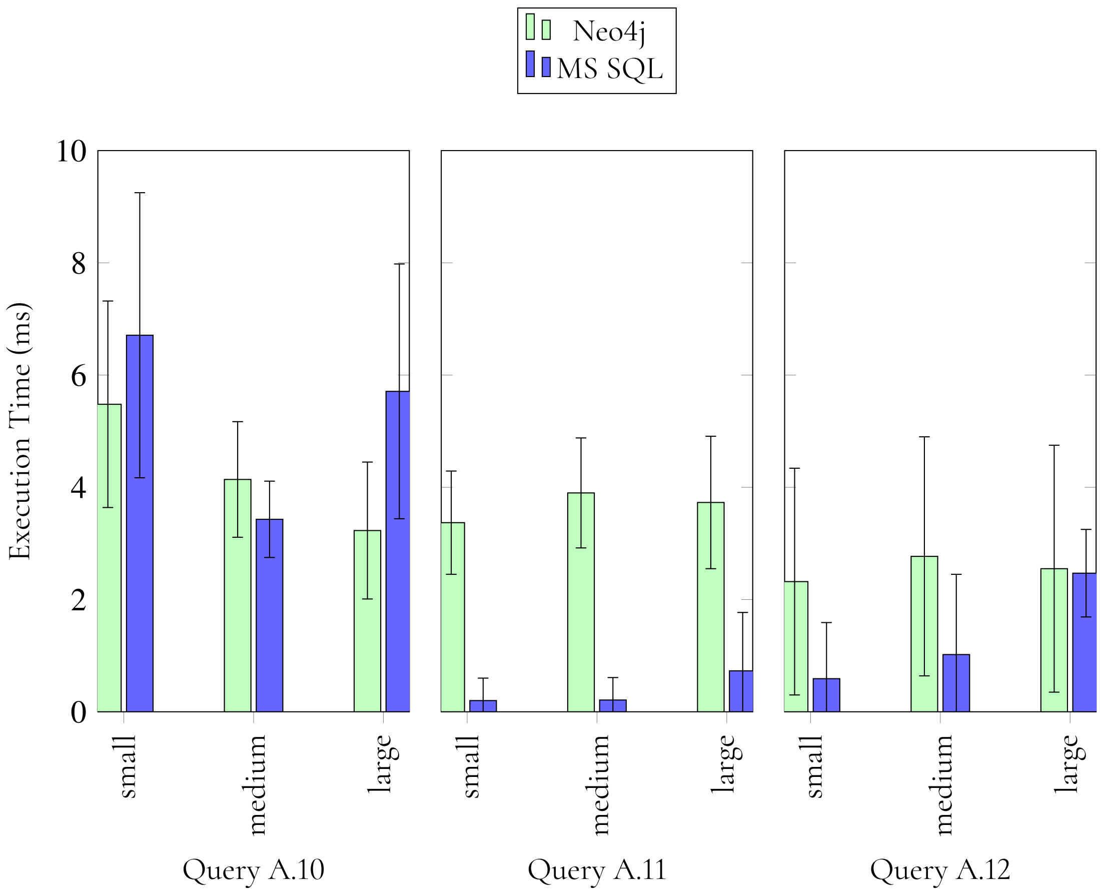


Figure 5.4: Update and basic retrieval query

The fourth comparison chart, seen in figure 5.4, are a set of some simple update queries as well as a basic retrieval query in the right most bar plot. In the left chart we update the database using a nested **`WHERE`** clause. In the middle chart we perform a simple update query using an indexed column. The right most bar plot in figure 5.4 represents a retrieval query that retrieves all employees at a given company. For the retrieval query, Neo4j performs worse than MS SQL, however, the larger the database, the closer Neo4j gets in terms of performance. In the middle bar plot which represents updating a company's name, it can be seen that Neo4j performs significantly worse than MS SQL for all database sizes, which was expected. However, in the left bar plot in figure 5.4 it can be seen that Neo4j performs slightly better than MS SQL for the more slightly more advanced update query.

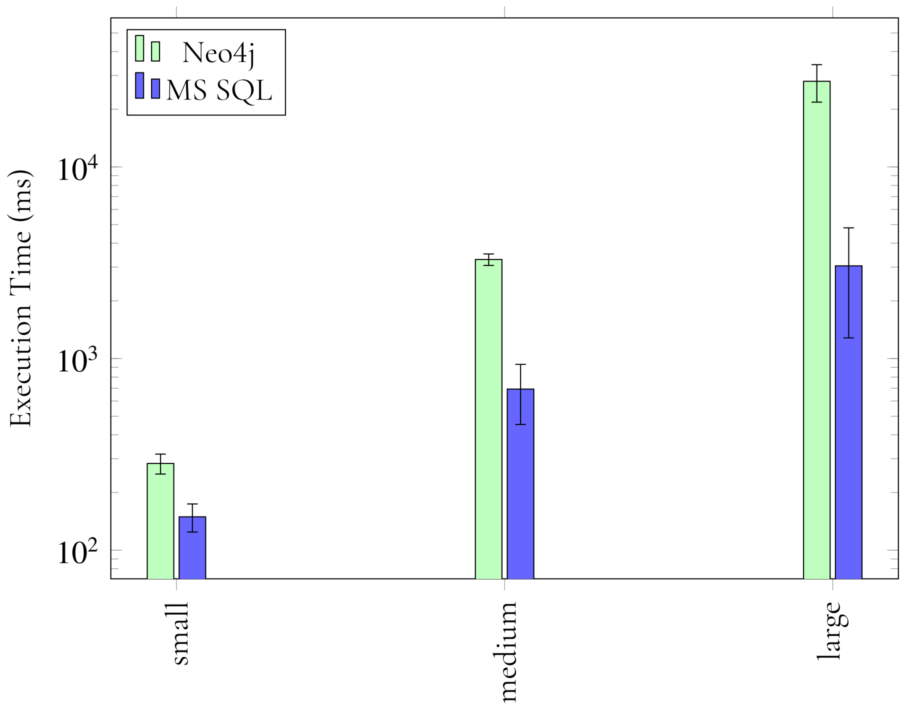


Figure 5.5: Complex filtering. Query A.13.

In figure 5.5, the query executes four joins with multiple **`WHERE`** statements using *OR* and *AND* to filter multiple columns. Once again we conclude that MS SQL performs faster than Neo4j. However, they both scale linearly in execution time. It should be noted that the scale of the y-axis is logarithmic, meaning that MS SQL is approximately 10 times faster than Neo4j for all database sizes.

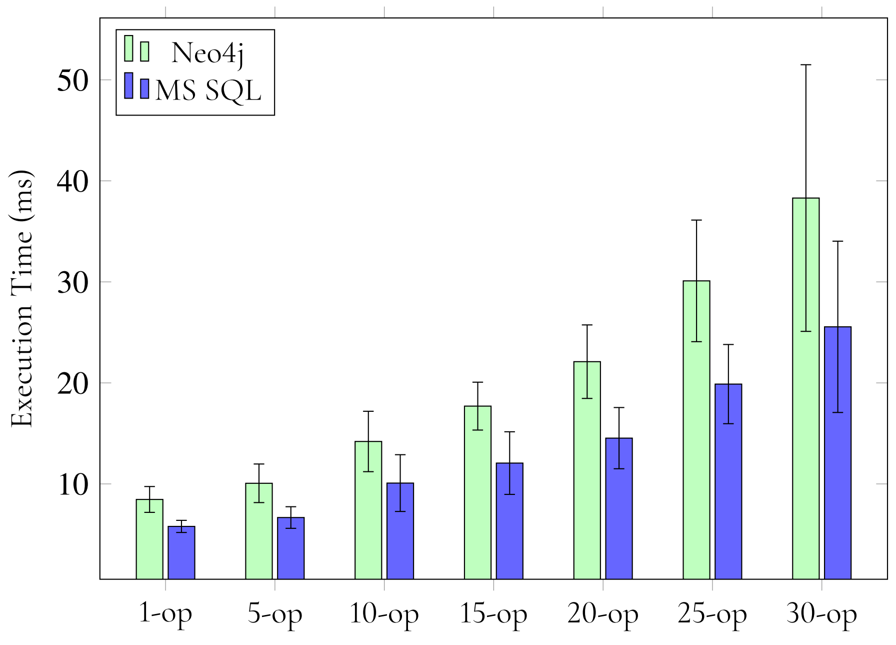


Figure 5.6: Regex vs String Operations in Query A.1

In figure 5.6, the query A.1 using the small database was executed with different amount of regex/string operations. This to test if there is a noticeable difference between the two types of string comparison operators. MS SQL uses simple string operations to compare strings, since its operator `LIKE` is a string operator. Neo4j, on the other hand, uses regular expression to compare strings, since its operator `=~` is a regex operator. Seen in the graph the distance between Neo4j and MS SQL increases, while the percentage difference stays the same. MS SQL is continuously (approximately) 50 percent faster than Neo4j.

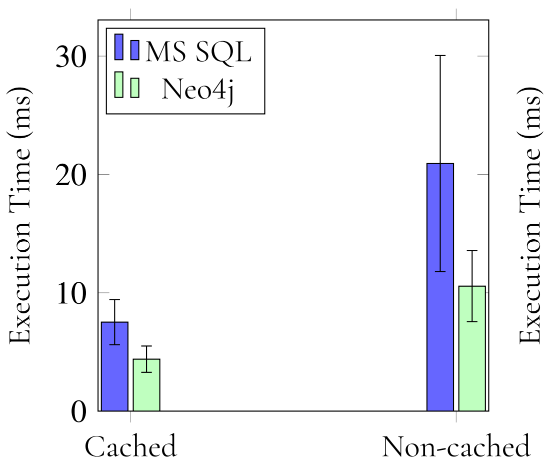


Figure 5.7: Cache difference for filter query on the large database. Query A.1.

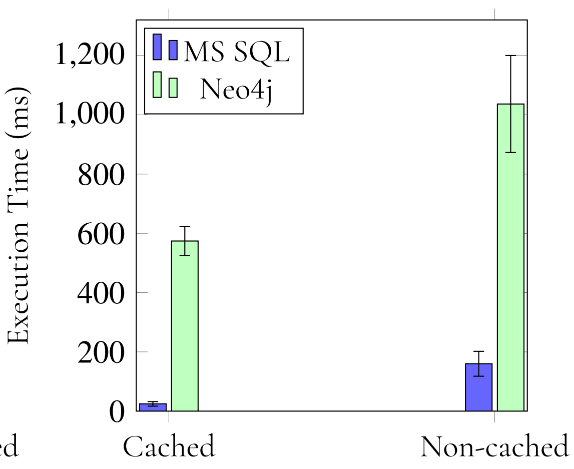


Figure 5.8: Cache differences for retrieval query on the large database. Query A.5.

The benchmarking conducted in this thesis was done by specifically designing the experimental setup not to cache the query results due to our understanding that customers using Lime CRM do not call the same queries that often, which results in limited data caching. However, benchmarking using caching was also performed by calling the same query many times in a row to see how the databases handle caching of data. The results of two queries being cached and benchmarked can be seen in figure 5.7 and 5.8. In figure 5.7, Neo4j's performance increased by 58% and MS SQL's performance increases by 64% when the queries are cached. At this point, both databases are approximately equally good at caching data.

However, as seen in figure 5.8, the results are completely different for queries that return a lot of data. In given figure, it can be seen that Neo4j's performance increases by 44% whereas MS SQL's performance is increased by 84%. These numbers really show the difference in caching for MS SQL and Neo4j. Neo4j is efficient at caching small amounts of data and smaller paths, however, when much data is returned in a single query, Neo4j is having a harder time caching it.

5.1.3 Execution Plans

Query A.1. MS SQL scans through a small table a (companies) with an attribute complying with the query filter, returning a small amount of rows k and then performing index seek on a larger table b (persons) to join additional data, taking $O(\log a + k \log b) \approx O(k \log b)$ time. Then, scanning all rows in a large table c (deals) and performing a hash match on all these row, taking $O(c + c)$. Resulting in the total big O notation:

$$O((k \log b) + (c + c)) \xrightarrow[b,c \to \infty]{} O(c)$$

Neo4j starts with the smallest table h (companies) and then filtering this already small table further, taking $O(h + h)$ time. Now we have k companies we would like to continue to traverse from. By expanding and filtering three times, usually finding more nodes than wanted when expanding that then is filtered out, taking $O((ak + k) + (bk + k) + (ck + k))$ time, when filtering with a linear path. Meaning, that we for each expand operator get a, b or c times irrelevant data that then is filtered out back to k. However, if the traversal path would instead be binary tree-structure for finding the relevant nodes, depending on how many nodes each relationship has, it would result in a higher order complexity, taking $O((ak + 2^1k) + (b2^1k + 2^2k) + (c2^2k + 2^3k))$. And for the general case it would result in taking $O((ak + c_1k) + (bc_1k + c_1c_2k) + (cc_1c_2k + c_1c_2c_3k))$. Resulting in the total big O notation:

$$O((h + h) + (ak + c_1k) + (bc_1k + c_1c_2k) + (cc_1c_2k + c_1c_2c_3k)) \xrightarrow[h,k \to \infty]{} O(h + k)$$

It is hard to come to a solid conclusion in this case, since it really depends on how the traversing path is structured. If each company has one or few relation(s) to the next node(s), and it continues somewhat linear, Neo4j would perform better than it would if it expands a lot.

Query A.13. MS SQL performs an expensive scan through all rows in a huge table n (histories). Then, passing all of those rows to three hash matches, each taking $O(n)$ time. However, all of these operations is running in parallel, for each gathered row from the huge table, decreasing the cost to the time it takes to scan the huge table once. Resulting in the total big O notation:

$$O(n)$$

Neo4j needs to make several operations through the huge table n (histories) and their relations, similar to MS SQL passing the table to multiple hash matches. However, this is not done in parallel making this an expensive query. Furthermore, when performing *expand* we often receive more relations than what we are interested in that then is filtered out. Lastly, we perform a filter. Resulting in the total big O notation:

$$O((n + n) + (3n + n) + (2n + n)) = O(9n)$$

Resulting in that MS SQL should perform faster than Neo4j, since the big O notation directly is about 10 times faster.

Query A.5. MS SQL once again does an index scan through the huge table n (histories), but filtering out the one complying with the filter, finding k entries, taking $O(\log n)$ time.

Those entries is then used to gather information from two other rather large tables a (deals) and b (persons), taking additional $O(k \log a + k \log b)$. Resulting in the total big O notation:

$$O(\log n + (k \log a) + (k \log b))$$

Neo4j finds a medium size amount of rows that complies with the query in the smallest table n (companies). Then, continues its process of expanding and filtering through the nodes, finding a small amount of relationships from the start and keeps traversing. Lastly, projecting some values to return. Similarly to what we concluded in our first execution plan, our big O notation once again depends on which values would go towards infinity.

Query A.2. MS SQL seeks through a huge table n (histories), finding all rows complying with the specified filter k and then performing a key look-up for each complying row, taking $O(\log n + k \log n) \approx O(k \log n)$. Then, four joins and tree more index seeks are performed, taking $O(k \log a + k \log b + k \log c)$. Resulting in the total big O notation:

$$O(k \log(n * a * b * c))$$

Neo4j finds the specified row in the small table n (deals), by performing an index seek, that complies with the query and the specific rows relations from the huge table k (histories), taking $O(n + k)$. Resulting in the total big O notation:

$$O(n + k)$$

5.2 Discussion

5.2.1 Indexing

Indexing primary and foreign keys is very important in MS SQL since they are then stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently. If a column that is part of a query is not indexed in MS SQL, a full table scan has to be done to find the data entries in the query. This costs $O(n)$, whereas the corresponding operation when indexing is used only costs $O(\log n)$. As a result of this, the difference in execution time is not that noticeable when having small tables, but when size increases the execution time scales badly. Therefore indexing is very important to MS SQL.

Indexing can also be done in Neo4j, however, due to the fact that Neo4j is an index-free adjacency graph, it ensures that every node keeps track of adjacent nodes, which is a sort of index itself. In fact, this is what we found to be a key strength of Neo4j and graph databases. One fact that was noticed is that it is important to have indices on at least one of the properties stated in the `WHERE` clause in each query for Neo4j. The reason for this is that if there is no index on any property that is part of the query, Neo4j has a hard time deciding from where to start the graph traversal. This leads to the graph evaluating all nodes of a given label, which is very expensive if there are many nodes in the database. The difference with index and without is easier discussed with the following example: In query A.3, none of the properties in the `WHERE` clause are indexed. As a result of this Neo4j's query planner decided that it needs to traverse all 10,000 companies in the database at the beginning of the queries to then filter out the companies that do not fulfill the name constraint specified in the `WHERE` clause. This is expensive as the path to and from 10,000 nodes has to be evaluated at the start of the query. On the other hand, in query A.2 the property **`deal_id`** is present in the query and by utilizing this Neo4j knows to start at that specific deal node which leads to a much narrower graph traversal and is therefore highly beneficial.

It is also important to mention that the fact that indexing is only used on the id:s in Neo4j and on the id:s and the foreign keys in MS SQL which might have an impact on the results. If specific indices were to be used, Neo4j might have performed better than MS SQL for some specific queries or the performance difference might have become even greater in favor of MS SQL. However, we decided to stay true to a simple model with close to no indices as this was more realistic than creating arbitrary indices that do not actually exist in the present Lime CRM system.

5.2.2 Performance

In Lime-CRM there are a few different types of queries that are executed when using their systems. When using their system to retrieve information, without modifying or creating new data, the retrieve or select queries are executed in the back-end. These queries could be categorized into two groups: joining and filtering. Joining is the queries that simply joins multiple tables on primary and foreign keys and retrieve large amounts of data. Filtering is the queries that only joins rows in tables with specific data values.

As mentioned in our theory section the key aspects with many tables and multiple nested relationships are met, indicating that graph databases should perform better than relation databases for this kinds of CRM systems. However, even when we perform extreme case complexity queries MS SQL still outperforms Neo4j even though it should perform better according to the studies conducted by Vicknair et al. [33] and Batra et al. [3]. This indicates that there might be one or several sources of errors in the benchmarking conducted by us, which could have had an impact on the result. These potential sources of error are discussed in section 5.2.5. But, much of the theory is not really backed-up with comparison results they are just stated. The comparison results that exist are instead queries that performs traversal queries and other graph-operations that is much more graph-related than MS SQL, and thus, in these studies, SQL performs worse than Neo4j.

For example, in [33], it is shown that for queries that are evaluating payload data, meaning that they need to evaluate the properties of the nodes and not just traverse the graph and find a pattern, Neo4j performs immensely worse than a SQL database. This is the same result as was discovered in this thesis, however, the results differ slightly as the databases used in this thesis are far larger than the ones used in [33]. A reason for these results is the fact that Lime CRM is built as a relational database, due to the interface of the system corresponding to large tables displaying large amounts of data. With this data model, Neo4j is heavily restrained due to the fact that every query being called in the system needs to evaluate many payload properties of many nodes. This means that even though the data is heavily connected with relationships in Lime CRM, the user interface of Lime CRM is built to display large tables with data that is heavily joined together. As long as the users and the user interface of Lime CRM operate the system by displaying and navigating in large tables, it is not enough to make Neo4j perform better than MS SQL just by the fact that the data has deep and many relationships. To achieve what was observed in other papers, i.e. [33], Lime CRM would need to perform more graph operations, such as finding connections between nodes that are very far apart in the graph structure, e.g. deep graph traversal, or finding specific paths in the graph instead of returning large amounts of data as it would do in the current state of Lime CRM. This phenomenon was also observed in the query profiler for Neo4j, where it was evident that projection of the results took a significant amount of the total query time whereas in MS SQL, the projection of the results is close to instant.

5.2.3 Caching

It is worth mentioning that Microsoft SQL Server is a paid program whereas the Neo4j client that is used is the free 'Community Edition'. If the enterprise edition of Neo4j would be used, the caching differences might have been different as the enterprise edition of Neo4j offers a feature called 'Active Page Cache Warmup' which records cache profiles by e.g. recording what data is stored in memory and what data is not present there. This facilitates that the cache contains some of the regularly queried data on database start up as well. This feature might have shown even further improval of performance when the queries are cached.

5.2.4 Queries

This section will discuss performance results for the individual queries presented in section 5.1. The discussion will mostly focus on the queries where a significant difference in performance can be seen, as it is not equally interesting to evaluate why they are performing approximately as good. An overall discussion about the differences as a whole will be conducted in the rest of section 5.2.

Insertion queries

One interesting result heavily interconnected with indexing is the execution times of the SQL queries in figure 5.3, where the middle bar plot is an insertion into a table with one index, the left most bar plot is an insertion query into a table with two indices and the right most bar plot is an insertion query into a table with four indices. It can clearly be seen that the fewer indices the table has, the shorter the execution time of insertion queries is for MS SQL. Having twice as many indices in a table where data is to be inserted results in approximately three times the execution time. This shows that creating indices on the data is not only positive, as the insert operations are influenced a lot. However, the execution times for the insertion queries are so small that the significant increase in execution time for different amount of indices does not have a noticeable impact on the performance of the system unless large amounts of data is bulk inserted at once. Overall though, Neo4j performs far worse than MS SQL. The reason for this performance difference is the fact that when insertion is made in Neo4j, every outgoing relationship from the created node has to be created. In theory, one could create a node that is completely isolated without any relationships, however, that node would serve no purpose and therefore all nodes are connected in this benchmark process. To be able to create relationships to the nodes that the newly inserted node is to be connected to, the existing nodes have to be found. E.g. in query A.8, the company with id y has to be found before the relationship between the person to be inserted and the given company can be made. This is a fairly simple process in that specific query, however, for more complicated queries such as A.9 there are several nodes and several relationships to be created for every insert that is to be made. This is the reason that MS SQL performs so much better for insert queries, as in MS SQL the associations to other tables are handled by foreign keys so there is no need to perform expensive look-ups whenever something is to be inserted.

Retrieval queries

Query A.1. As seen in the left bar plot in figure 5.1, MS SQL performs significantly worse than Neo4j for the large database. This is one of the few queries were Neo4j is faster than MS SQL so it is worth evaluating why this is the case. Firstly, Neo4j strongly benefits from the fact that some of the focus of this query is on coworkers, which is the entity with the fewest database entries. The way that Neo4j executes this query is by finding all nodes with the coworker label and thereafter filtering away all nodes that do not fulfill the condition in the `WHERE` clause. After this operation is done, there are approximately 50 coworkers left that fulfill the condition. From these coworkers, all deals that they are salespersons for are found and the graph expands one step. Thereafter, all persons that are responsible for these deals are located by expanding the person relationships from the deals that are already found and lastly the `WORKS_AT` relationships from the persons are expanded to find the companies related to these persons. All of these operations are very cheap and evaluate very few nodes compared to the size of the complete graph. As explained in section 5.1.3, the time complexity of the queries are approximately the same order, however, Neo4j on average performed approximately twice as good for this query, as seen in figure 5.1. In total there are approximately 13,000 database hits being executed for this query in Neo4j, which is not a lot. However, for MS SQL, the outlook is completely different. In total, MS SQL evaluates about 100,000 rows for this query, which is significantly more than Neo4j. The reason for the difference in database operations for these queries is the fact that MS SQL first performs an index scan on the complete companies table to be able to filter out the companies that fulfill the predicate in the `WHERE` clause, which results in 10,000 data reads and it also performs an index seek on the persons table to find the persons that work at the companies found after the filtering in the first index scan was done. Thereafter, the given companies are joined with persons to be able to extract all the data from these rows. After that, a clustered index scan is performed on the deals table to find those deals whose `person_id`:s match persons that exists in the data that has been obtained through joining the companies and persons. This operation performs 50,000 row reads as it has to go through the complete deal table. Already at this point, MS SQL has performed 5 times the operations of Neo4j, which is very significant. Thereafter, these obtained deals are joined with coworkers and the data is later extracted. From this evaluation of the query execution plan, it is clear why Neo4j performs better for this query than MS SQL. However, MS SQL is not 8 times slower than Neo4j in this query even though the figures in the execution plan mentioned in 5.1.3 indicate this. This is mostly due to the fact that MS SQL executed several of its operation in parallel whilst Neo4j does all sequentially.

Query A.13. One of the largest differences in execution time is presented in figure 5.5, which makes it interesting to discuss. When observing the query profiler for Neo4j, it can be seen that the first filtering conditional results in approximately 14,000,000 database hits. These database hits can be seen as different paths that have to be searched in order to find the nodes that fulfill the filtering demands. From this number it is apparent how this query takes a very long time in Neo4j, as 14,000,000 path tries are very performance costly. Furthermore, finding all persons that attended a given history results in another 9,000,000 database hits which also affects the performance of the query. The reason for there being so many database hits for finding nodes in this particular query is the fact that the histories entity which is being filtered in the query is the entity which has the largest amount of nodes stored in the database, namely 3,000,000. This makes Neo4j spread into a huge graph, as the particular query that is being evaluated can not be optimized to create a smaller visit tree for its query. In total Neo4j needs approximately 36,000,000 database hits to be able to find the result, whereas MS SQL needs almost half of that, namely 19,000,000 need to be read. As seen from the previous statement, MS SQL also takes a very long time to execute, but the execution time of MS SQL is expected due to extensive filtering being a complex and time consuming operation no matter which database is used. After all, 19,000,000 rows to be read is time consuming no matter how you look at it. However, as mentioned earlier, the big O notation for this query, explained in section 5.1.3 indicates that MS SQL should be approximately 12 times faster than Neo4j, which is very close to the actual results, where Neo4j performed about 10 times worse than MS SQL for this query. Even though MS SQL has to perform three $O(n)$ operations, these are all done in parallel, whereas the big O for this query in Neo4j is $O(9n)$. These differences indicate that the query in theory should be approximately 9 times faster in MS SQL, which is very close to the measured results which showed a 10 times faster execution time for MS SQL.

Query A.5. Another interesting query result worth discussing is the result in the middle bar plot in figure 5.2. This query presents one of the largest performance differences between Neo4j and MS SQL. After the query planner was analyzed for Neo4j, it was observed that it is the projection of data that takes most of the execution time of the query. Neo4j is actually not that slow at finding the nodes where the properties to be returned are stored, however as mentioned earlier, it is very inefficient at returning large amounts of data (properties). The performance in this query is limited due to similar factors to the other queries that are slow in Neo4j, namely that Neo4j does not have a given start node to start from. Instead, for this particular query, Neo4j needs to expand all outgoing `WORKS_AT` relationships to find all persons working at the companies. Thereafter all `ATTENDED` relationships from the persons are expanded to find the histories that these persons attended. This goes on for several steps to be able to connect all node types and their relationships to be able to retrieve the requested data. It becomes clear why this is not sustainable in the long run, as the graph just keeps expanding and expanding with additional relationships. In this query, it becomes apparent that even though database reads are an indication that is worth noting when evaluating the performance, it is not the only impactful factor. When comparing the theoretical big O notation for the databases when this query is executed, there does not seem to be that big of a difference. Neo4j should in theory take approximately twice as long as MS SQL as mentioned in section 5.1.3 but that is not the case in practice. Instead, one of the reasons that MS SQL performs so much better for query A.5 is the fact that at least half of the work in MS SQL is run in parallel during the query execution. So if two tables are to be joined, they are both evaluated in parallel before the join is conducted. This is not the case in Neo4j, which slows down the query execution time noticeably. A conclusion that can be made out of this is the fact that Neo4j is bound to not perform better in the current Lime CRM system due to the system's core functionality being built around displaying large amounts of data and the data is stored in such a way that there is no natural starting point for the Neo4j queries. However, if Lime were to reevaluate how their CRM system is to be used, or find a potential customer that is interested in finding complicated relationships and/or patterns within the stored data, it might still be possible to utilize a graph database efficiently.

Query A.2. It is also important to discuss the queries where the databases are similar in performance to unravel how the different databases perform the given queries. As seen in figure 5.1, Neo4j and MS SQL are very close performance-wise for query A.2 and query A.3 for all database sizes. It is interesting that the databases have two completely different approaches that work approximately equally well as seen in the bar plots. For query A.2 Neo4j finds the deal with the specified id by utilizing an index seek which is very efficient. For specific query, the graph traversal is beneficial as the graph does not expand very much, instead it only goes a few steps in depth, as explained in section 5.1.3. In MS SQL on the other hand, the approach is very different. Here, the first action that is done is to do a key look-up to filter out all histories that do not fulfill the `WHERE` clause. Thereafter, coworkers, documents and persons are all join in one at a time. These join operations are cheap as they all perform index seeks that evaluate less than 100 rows to find the given ids that match the foreign keys available in the previous join. So both databases having very efficient query plans for this specific query and are therefore approximately equally efficient at execution it and retrieving the data. The execution plan discussed in section 5.1.3 show the same results as the actual result, namely that MS SQL and Neo4j should perform roughly the same. The small difference observed in the actual result is as mentioned above the fact that MS SQL performs some of the actions in this particular query in parallel, which makes it slightly faster.

Update queries

Some results worth noting are the execution times in the left and the middle bar plot in figure 5.4, which represent update queries. In the left picture, representing query A.10 the results show that Neo4j has a shorter execution time. Even though the execution time of Neo4j is not that much shorter than the one for SQL Server, it is worth noting that MS SQL takes a factor of five longer for the left query than for the middle query, whereas Neo4j takes approximately the same amount of time for both queries. The reason for Neo4j performing far better compared to SQL in query A.10 is the fact that this query is perfectly constructed for Neo4j to do well. For the databases to be able to update all deals for a given company, all connections between that company and its employees has to be found. In Neo4j, this is simply done by first performing an index seek to find the node with the given id and thereafter there is only a need to traverse all `WORKS_AT` relationships going to the specified company and thereafter all relationships going out of the found `Person` nodes to find the deals. This is very cheap as Neo4j lets each node keep an index to all adjacent nodes and there is only a depth of 2 being traversed. However, in MS SQL, it is an expensive operation to find all persons that are connected to a given company and thereafter finding all deals connected to those persons. The reason for this is that index seeks have to be done on the foreign keys of the deals and the persons table before the data can be updated. Therefore, SQL performs far worse for this query compared to Neo4j.

However as seen in the middle bar plot in figure 5.4 MS SQL is far more efficient at executing update statements that do not contain joins or nested sub queries. The reason for this is that it only needs to perform one index scan to find the row to be updated whereas Neo4j needs to find all company nodes and thereafter filter to find the specific node that is to be updated. The performance difference here is due to it being a much faster operation to perform an index scan on a fairly small table compared to finding all company nodes in a graph structure. These particular differences are only relevant for this specific benchmarking setup, namely having no indices on any property other than the id. The performance of MS SQL would be improved by approximately 91% according to data provided by Microsoft SQL Server Management Studio when running the query. This is showing that an index is very important for MS SQL in this particular query. It was tested if Neo4j would also perform far better with indexing is this case, but the results showed that there was only a small improvement in execution time. This is indicating that indexing is not as important for Neo4j as it is for MS SQL for this particular query.

5.2.5 Sources of error

As the results achieved in this thesis show different results than the research conducted by e.g. Vicknair et al.[33] and Batra et al.[3], it is important to mention factors that might have impacted the results.

Time measurement methods

Due to us using two different time measurement methods when measuring the execution time of the two databases under comparison, the gathered results might have been impacted by this. Even though both execution times measured for the queries correspond to the execution times displayed in respective official client tool, there is still a possibility that they measure the time at different points in the execution resulting in potential time differences. This might be part of the explanation to the big standard deviations observed in the queries with the shortest execution time, such as query A.12.

Heavy usage of string comparisons

As many of the queries used in this benchmark have some aspects of filtering within them, it is worth having a discussion about the impact this might have had on the results. Firstly, it needs to be mentioned that it is unknown how the string comparisons of the `LIKE` method in Microsoft SQL Server and the regular expression evaluator `=~` in Neo4j are implemented. Due to query A.13 being the query showing the largest execution time difference between the two databases and also being the query with the heaviest usage of the string comparison methods, this might indicate that the different string comparison methods might have had an noticeable impact on the queries where heavy filtering was done. In figure 5.6, we evaluate how the execution time for query A.1 is affected by adding more and more regex/string operators. It can be seen that the percent difference in execution time between the two databases stays approximately constant which indicates that the string comparison methods are similar in performance. Due to the database used for this test being very small, a small amount of data is returned by each query which leads to the results not being that affected by the data returned and instead being most affected by the string/regex operator evaluation times.

Standard deviation

It is also worth having a short discussion about the standard deviations of the query execution times and why they are very significant for some queries. Due to many of the queries performed in this benchmark being filtering queries, it is natural that filtering queries might not always find results. This is one of the reasons behind the queries in figure 5.1 having large standard deviations compared to the execution time itself. As mentioned earlier, these queries are called 250 times with random values, and some of the executions yield no results at all. This impacts the execution time significantly and therefore creates a large standard deviation as there is a noticeable difference in execution time when returning 100 values compared to none. Furthermore, the query execution times are also dependant on the CPU and memory usage of the computer, which may be another factor to the differences in execution times as there are many ongoing processes on Lime's work computers at all times and these processes might affect the execution time of the queries.

Differences between a model and the real system

Due to the fact that the databases used in this thesis do not represent the exact data relationships of the Lime CRM System, it is possible that this had an effect on the execution times measured in this thesis. Moreover, it is hard to conclude results for a complete system based on only 13 queries. Even though these queries perform common actions in the system and cover much functionality of the databases, there is far more functionality that has not been evaluated. Therefore, it is worth mentioning that the difference between the 7 entity model of the CRM system that was used in this benchmark and the real CRM system with far more entities might be noticeably different in performance.

Chapter 6
Conclusions

This research aimed to compare the performance of graph databases and relational databases by exploring key factors that affect the execution times of each database. Furthermore, it was aimed to determine if and when one database technology would perform better than the other.

Based on the data retrieved from testing, analyzing and comparing the performance in terms of execution time for 13 queries executed by the databases Neo4j and Microsoft SQL Server, it can be concluded that Microsoft SQL Server is a far more suitable database to use for representing Lime's current CRM-System. It was shown that Neo4j only showed a better performance than MS SQL for a very small subset of the queries that were benchmarked, which implies that MS SQL is more efficient at handling the CRM data.

While the results in this thesis only apply if there are only indices on foreign keys and primary keys in MS SQL and indices on primary keys only in Neo4j, the results can generally be applied, however, they might not be very accurate to how the CRM-systems would perform during actual run time as the queries with bad performance would be indexed in one way or another in a real system that is live on the market.

Given the guaranteed performance increase, according to Neo4j, when managing multiple joins, mentioned earlier it was a reasonable assumption to assume Neo4j would perform better at the more complex and nested join queries. Despite that, the results yielded from the research conducted indicates that Neo4j performed worse than Microsoft SQL Server for the majority of queries. Something we observed was that usually when mentioning that graph databases perform better at multiple joins it is usually also mentioned that Neo4j is index-free, which makes us believe that when mentioning their superior performance against relational databases they compare the databases without indexing anything. Naturally, Neo4j benefits from this as it already has 'built-in' indices.

As a result of this, our thesis helps improve the knowledge gap that exists in regards to when graph databases are suitable and when they are not. Our results indicate that it is not sufficient for the data stored to be highly interconnected to benefit from a graph structure. Instead, the queries have to be specified in a given way, preferably so that the sub-graph that is traversed in a query is narrowed. This means that the graph database is slow in queries that keep expanding the graph instead of having a narrow starting point and continuously filtering and expanding at the same time so that the path traversed stays narrow. Of course, these results are specific for CRM systems. Other data systems might have other uses for graph databases that cannot be expressed in SQL.

6.1 Future Work

The research conducted in this thesis focused on determining whether graph databases or relational databases are most suitable for Lime's CRM-system, which has heavily connected data but is restricted to displaying tables with data. In the future, it is worth researching if there are use cases for CRM-systems that are not represented in the current CRM systems on the market due to the limitation posed by the relational databases that represent the data. For example, it could be interesting to find whether there is a use for evaluation shortest paths between two nodes of interest in the graph, or if there is a desire to traverse the graph recursively. If database operations such as these would be found interesting to CRM users, this could be realized by having a graph database as a "slave database" that replicates the master database but is only used for operations where graph databases are beneficial, such as operations with graph traversal to an extensive depth. With some knowledge from some Lime employees we also learned that some systems at Lime actually used elastic search database which is achieved by adopting NoSQL for these specific cases. That is, when performing filtering or searching.

Another interesting point that was not evaluated in this thesis but is worth looking into in the future is the fact that Neo4j mostly utilizes the heap for query executions and state transitions. Therefore, it could be worth evaluating how Neo4j would handle concurrent access to a large and very busy CRM system to see if it is realistic to host the database with Neo4j and what kind of specifications a potential host machine/server would have to have to be able to handle the heavy memory load.

To further analyze the complexity of database queries, more research has to be done on the subject. To be able to draw any conclusions at all within this subject, one would have to perform a large empirical study to evaluate whether the study participants consider the complexities of the queries to be the same as the results of this thesis show to get real user input on the perceived complexity differences. Lastly, the model used to determine the query complexity in this thesis could be further improved by analyzing more than separate tokens, e.g. by conducting analysis on pairs or larger tuples of tokens to evaluate how different tokens combined affect the overall readability of the queries.

Bibliography

[1] Renzo Angles. A comparison of current graph database models. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 171–177. IEEE, 2012.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[3] Shalini Batra and Charu Tyagi. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2):509–512, 2012.

[4] Judith S Bowman, Sandra L Emerson, and Marcy Darnovsky. *The practical SQL handbook: using structured query language*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[5] Francis Buttle. *Customer relationship management*. Routledge, 2004.

[6] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] Ramez Elmasri and Shamkant Navathe. *Fundamentals of database systems*. Addison-Wesley Publishing Company, 2010.

[8] Daniele Faraglia. Faker documentation, 2014. `https://faker.readthedocs.io/en/stable/`.

[9] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

[10] Grant Fritchey. *SQL Server Execution Plans*. Red gate books, 2012.

[11] Seth Gilbert and Nancy A Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[12] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[13] Cypher Language Group. Cypher query language reference, version 9, 2019. `https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf`.

[14] Christoforos Hadjigeorgiou et al. Rdbms vs nosql: Performance and scaling comparison. *EPCC, The University of Edinburgh*, 2013.

[15] 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*, pages 195–204. ACM, 2013.

[16] Karl Huppler. The art of building a good benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 18–30. Springer, 2009.

[17] Reiny Iriana and Francis Buttle. Strategic, operational, and analytical customer relationship management: attributes and measures. *Journal of Relationship Marketing*, 5(4):23–42, 2007.

[18] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6), 2012.

[19] Michael Kleehammer. pyodbc wiki, 2019. `https://github.com/mkleehammer/pyodbc/wiki`.

[20] Vimtech Ltd. About unix timestamp, 2010. `http://unixtimestamp.50x.eu/about.php`.

[21] Surajit Medhi and Hemanta K Baruah. Relational database and graph database: A comparative analysis. *Journal of Process Management. New Technologies*, 5(2):1–9, 2017.

[22] Microsoft. sqlcmd utility, 2018. `https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility?view=sql-server-2017`.

[23] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, page 36, 2013.

[24] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[25] neo4j. Cypher manual, chapter 5.2 constraints, 2019. `https://neo4j.com/docs/cypher-manual/current/schema/constraints/`.

[26] neo4j. Cypher manual, chapter 7. execution plans, 2019. `https://neo4j.com/docs/cypher-manual/current/execution-plans/`.

[27] Neo4j. opencypher project, 2019. `https://www.opencypher.org/`.

[28] Adrian Payne and Pennie Frow. A strategic framework for customer relationship management. *Journal of marketing*, 69(4):167–176, 2005.

[29] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases.* " O'Reilly Media, Inc.", 2015.

[30] Neo Technology. Neo4j bolt driver 1.7 for python, 2018. `https://neo4j.com/docs/api/python-driver/current/index.html`.

[31] Richard J Trudeau. *Introduction to graph theory.* Courier Corporation, 2013.

[32] Moshe Y Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. ACM, 1982.

[33] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010.

[34] Victor Winberg and Jan Zubac. VictorWinberg/graph-relational-benchmark: Version v1.0.0. *Computer Science | Faculty of Engineering LTH*, May 2019. doi: `10.5281/zenodo.2659423`.

[35] Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012.

Appendices

Appendix A

Queries

In all queries below, x and y represent values that exist in the database, or in some cases, parts of values that exist in the database, e.g. the first three letters of an entity existing in the database.

A.1 Filter on Coworker and Company

A query that returns all coworkers with a randomized first name that work in a city that begins with a randomized letter combination.

Cypher

```
MATCH (c: Company)<-[:WORKS_AT]-(p: Person),
(d: Deal)<-[:RESPONSIBLE_FOR]-(p),
(d)<-[:SALESPERSON_FOR]-(co: Coworker)
WHERE co.name =~ 'x.*' AND c.city =~ 'x.*'
RETURN co.name, c.name, c.city
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 co.name, c.name, c.city
FROM companies as c
LEFT JOIN persons as p ON p.company_id = c.id
LEFT JOIN deals as d ON d.person_id = p.id
LEFT JOIN coworkers as co ON d.coworker_id = co.id
WHERE co.name LIKE 'x%' AND c.city LIKE 'y%
```

A.2 Filter on History and Deal

A query returning all histories that are of type 'Call', that occurred before a specific date and that are part of the deal with the specified id.

Cypher

```
MATCH (d: Deal)<-[:PART_OF]-(h: History),
(h)<-[:ATTACHED_TO]-(doc: Document),
(h)<-[:ATTENDED]-(c: Coworker),
(h)<-[:ATTENDED]-(p: Person)
WHERE d.id = x AND h.type = 'Call'
AND h.date < 'y'
RETURN h.date, c.name, h.type, p.name, doc.description
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 h.date, c.name, h.type, p.name, doc.description
FROM deals AS d
LEFT JOIN histories AS h ON h.deal_id = d.id
LEFT JOIN documents AS doc on h.document_id = doc.id
LEFT JOIN persons AS p ON h.person_id = p.id
LEFT JOIN coworkers AS c ON h.coworker_id = c.id
WHERE d.id = x AND h.type = 'Call'
AND h.date < 'y';
```

A.3 Filter on Deal and Company

A query returning all deals above the given probability at a company beginning with the given letter combination.

Cypher

```
MATCH (p: Person)-[:RESPONSIBLE_FOR]->(d: Deal),
(p)-[:WORKS_AT]->(c: Company)
WHERE d.probability > x AND c.name =~ 'y.*'
RETURN p.name, p.email, p.phone, d.name, c.name
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 p.name, p.email, p.phone, d.name, c.name
FROM persons AS p
LEFT JOIN deals AS d ON p.id = d.person_id
LEFT JOIN companies AS c ON p.company_id = c.id
```

```
WHERE d.probability > x AND c.name LIKE 'y%';
```

A.4 Filter on Deal then Transfer

A query that filters all deals containing a given word, selects all histories that are part of those deals and then returning all persons that are part of those histories.

Cypher

```
MATCH (co: Coworker)-[:SALESPERSON_FOR]->(d: Deal),
(d)<-[:RESPONSIBLE_FOR]-(p1: Person)
WITH d.id as d_id
WHERE d.name =~ 'x.*' OR p1.name =~ 'x.*' OR co.name =~ 'x.*'
MATCH (h: History)-[:PART_OF]->(deal: Deal {{id: d_id}}),
(h)<-[:ATTENDED]-(p2: Person)
RETURN COLLECT(DISTINCT p2.name), p2.email
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 p1.name, p1.email
FROM persons AS p1
LEFT JOIN histories AS h1 ON h1.person_id = p1.id
WHERE h1.id IN (
    SELECT h2.id
    FROM histories AS h2
    WHERE h2.deal_id IN (
        SELECT d3.id
        FROM deals AS d3
        LEFT JOIN persons AS p2 ON d3.person_id = p2.id
        LEFT JOIN coworkers AS co ON d3.coworker_id = co.id
        WHERE d3.name LIKE 'x%' OR
        p2.name LIKE 'x%' OR co.name LIKE 'x%'
    )
)
GROUP BY p1.name, p1.email;
```

A.5 Filter on History

A query returning all deals with the specified history type.

Cypher

```
MATCH (d: Deal)<-[:PART_OF]-(h: History),
(h)<-[:ATTENDED]-(co: Coworker),
(h)<-[:ATTENDED]-(p: Person),
(p)-[:WORKS_AT]->(c: Company)
WHERE h.type = 'x'
RETURN h.date, co.name, h.type, p.name, c.name, d.name
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 h.date, co.name, h.type, p.name, c.name, d
    .name
FROM histories AS h
LEFT JOIN deals AS d ON h.deal_id = d.id
LEFT JOIN coworkers AS co ON h.coworker_id = co.id
LEFT JOIN persons AS p ON h.person_id = p.id
LEFT JOIN companies AS c ON p.company_id = c.id
WHERE h.type = 'x';
```

A.6 Filter on Deal and Coworker

A query returning all deals above a given probability for the coworker that is responsible for the deal with the highest probability in the system.

Cypher

```
MATCH (d: Deal)<-[:SALESPERSON_FOR]-(co: Coworker)
WITH co.id as id, d.probability as prob
ORDER BY d.probability DESC LIMIT 1
MATCH (c: Coworker {{id: id}})-[:SALESPERSON_FOR]->(deal: Deal)
WHERE deal.probability > x
RETURN deal.name, deal.value, deal.probability, c.name
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 d.name, d.value, d.probability, co.name
FROM deals AS d
LEFT JOIN coworkers AS co ON d.coworker_id = co.id
WHERE co.id IN (
    SELECT TOP 1 co2.id
    FROM deals as d2
    LEFT JOIN coworkers AS co2 ON d2.coworker_id = co2.id
    ORDER BY d2.probability DESC
) AND d.probability > x
ORDER BY d.probability DESC;
```

A.7 Insert Deal

A query that inserts a new Deal node and creates all outgoing relationships from the newly created node.

A query that insert a row in the deals table.

Cypher

```
MERGE (d: Deal {
id: x, name: 'Best Deal Ever', value: 10, probability: 0.999
})
MERGE (p: Person {id: y})
MERGE (c: Coworker {id: y})
MERGE (p)-[:RESPONSIBLE_FOR]->(d)<-[:SALESPERSON_FOR]-(c);
```

Transact-SQL

```
INSERT INTO deals
VALUES (x, 'Best Deal Ever', 10, 0.99999, y, y);
```

A.8 Insert Person

A query that inserts a new Person node and creates all outgoing relationships from the newly created node.

A query that insert a row in the persons table.

Cypher

```
MERGE (p: Person {
id: x, name: 'Inserted Name',
phone: '07012345678', position: 'CEO', email: 'insert@insert.com'
})
MERGE (c: Company {id: y})
MERGE (p)-[:WORKS_AT]->(c);
```

Transact-SQL

```
INSERT INTO persons
VALUES (x, 'Inserted Name',
'07012345678', 'CEO', 'insert@insert.com', y);
```

A.9 Insert History

A query that inserts a new History node and creates all outgoing relationships from the newly created node.

A query that insert a row in the histories table.

Cypher

```
MERGE (h: History {
id: x, type: 'Call', notes: 'Created', date: '2018-03-15'
})
MERGE (doc: Document {id: y})
MERGE (d: Deal {id:y})
MERGE (p: Person {id: y})
MERGE (c: Coworker {id: y})
MERGE (h)<-[:ATTACHED_TO]-(doc)
MERGE (h)<-[:PART_OF]-(d)
MERGE (h)<-[:ATTENDED]-(p)
MERGE (h)<-[:ATTENDED]-(c);
```

Transact-SQL

```
INSERT INTO histories
VALUES (x, 'Call', '2018-03-15',
'Created', y, y, y, y);
```

A.10 Update Deal

A query that updates the deal probability for all deals linked to a specific company.

Cypher

```
MATCH (d: Deal)<-[:RESPONSIBLE_FOR]-(p:Person)-[:WORKS_AT]->(c:Company)
WHERE c.id = x
SET d.probability = 0.99
RETURN d;
```

Transact-SQL

```
UPDATE deals
SET deals.probability = 0.99
WHERE deals.person_id IN (
    SELECT p.id
    FROM persons as p
    WHERE p.company_id = x
);
```

A.11 Update Company

A query that updates the company name for a given company.

Cypher

```
MATCH (c: Company)
WHERE c.id = x
SET c.name = 'Updated'
RETURN c.name;
```

Transact-SQL

```
UPDATE companies
SET companies.name = 'Updated'
WHERE companies.id = x
```

A.12 Filter on Company

A query that returns all persons working at a specific company.

Cypher

```
MATCH (p: Person)-[:WORKS_AT]->(c: Company)
WHERE c.id = x
RETURN p.name, p.email, c.name
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 p.name, p.email, c.name
FROM persons AS p
LEFT JOIN companies AS c ON p.company_id = c.id
WHERE c.id = x;
```

A.13 Filter on multiple entities

A query that utilizes the filtering function in Lime CRM by typing two random data entities and looking through all possible columns for a potential hit.

Cypher

```
MATCH (d: Deal)<-[:PART_OF]-(h: History),
(h)<-[:ATTACHED_TO]-(doc: Document),
(h)<-[:ATTENDED]-(c: Coworker),
(h)<-[:ATTENDED]-(p: Person)
WHERE (h.type =~ 'x.*' OR c.name =~ 'x.*' OR
p.name =~ 'x.*' OR doc.description =~ 'x.*')
AND (h.type =~ 'y.*' OR c.name =~ 'y.*' OR
p.name =~ 'y.*' OR doc.description =~ 'y.*')
```

```
RETURN h.type, h.date, c.name, p.name, doc.description
LIMIT 10000;
```

Transact-SQL

```
SELECT TOP 10000 h.type, h.date, c.name, p.name, doc.description
FROM histories AS h
LEFT JOIN deals AS d ON h.deal_id = d.id
LEFT JOIN coworkers AS c ON h.coworker_id = c.id
LEFT JOIN persons AS p ON h.person_id = p.id
LEFT JOIN documents AS doc ON h.document_id = doc.id
WHERE (h.type LIKE 'x%' OR c.name LIKE 'x%' OR
p.name LIKE 'x%' OR doc.description LIKE 'x%')
AND (h.type LIKE 'y%' OR c.name LIKE 'y%' OR
p.name LIKE 'y%' OR doc.description LIKE 'y%');
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-06-11

EXAMENSARBETE A Comparison of Relational and Graph Databases for CRM Systems
STUDENTER Jan Zubac, Victor Winberg
HANDLEDARE Per Andersson (LTH), Johan Groth (Lime)
EXAMINATOR Flavius Gruian (LTH)

Grafdatabas, den nya lagringsmetoden?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Jan Zubac, Victor Winberg**

Det finns flertal fall då grafdatabaser överträffar traditionella relationsdatabasers prestanda. Är grafdatabaser potentiellt den nya lagringsmetoden vid stor mängd sammankopplad data? Vilka villkor krävs? För att sätta det på prov har vi skapat en testmiljö med en stor mängd relationsbunden data.

På sistone har grafdatabasers popularitet och därmed även användningen av dessa databastyper ökat avsevärt. Några av de hemsidor med störst besökarantal såsom Facebook och LinkedIn använder sig i stor utsträckning av grafdatabaser för att lagra sin data. Anledningen till denna ökning är det faktum att dagens applikationer lagrar stora mängder data som är sammankopplad med ett flertal relationer. Traditionella databaser, som lagrar data på ett sätt som kan liknas vid ett Excel-ark med stora tabeller, sägs ha svårt att hantera de nya kraven som ställs på databaser i dagens teknikdrivna samhälle. Stämmer detta eller är de traditionella tabellutformade databaserna trots allt bättre?

Grafdatabaser använder sig av grafstrukturer för att lagra data. Dessa strukturer består av noder som lagrar data och anslutningar som kopplar ihop datan.

I vårt examensarbete har vi undersökt om prestandan i ett kundhanteringssystem, även kallat CRM-system, kan förbättras genom att använda sig av grafdatabasen Neo4j. Dessa typer av system genomför komplicerade operationer som ställer höga krav på databaserna som hämtar datan. Dessa komplicerade operationer gör att det kan ta uppemot en halv minut innan den efterfrågade datan kan visas för användaren, vilket är problematisk i vardagligt bruk. Därmed var det intressant att jämföra två helt olika databastyper för att se vilken som gjorde bäst ifrån sig och om det därmed var intressant att övergå till att använda sig av grafdatabaser i CRM-system.

Enligt förespråkare för grafdatabaser är de snabbare än traditionella databaser i ett flertal fall. Det CRM-system som vi har gjort studien på, Lime CRM, använder sig i dagsläget av databassystemet Microsoft SQL Server och den lagrade datan uppfyller många av faktorerna som antyder att grafdatabaser bör briljera i systemet. Men resultatet visar något helt annat.

De traditionella relationsdatabaserna presterade bättre i de flesta användningsfallen av CRM-system, trots de påstådda motgångarna som denna databastyp sägs uppleva vid hantering av stora mänger väldigt sammankopplad data. En av anledningarna till detta är att Limes system inte uppnår en tillräckligt stor databasstorlek för att grafer ska vara fördelaktiga att använda.

Studiens resultat kan användas för att mer specifikt undersöka när grafdatabaser överträffar traditionella databaser, så att företag, specifikt inom kundhantering, vet vilken databastyp som hade varit bäst för deras system.