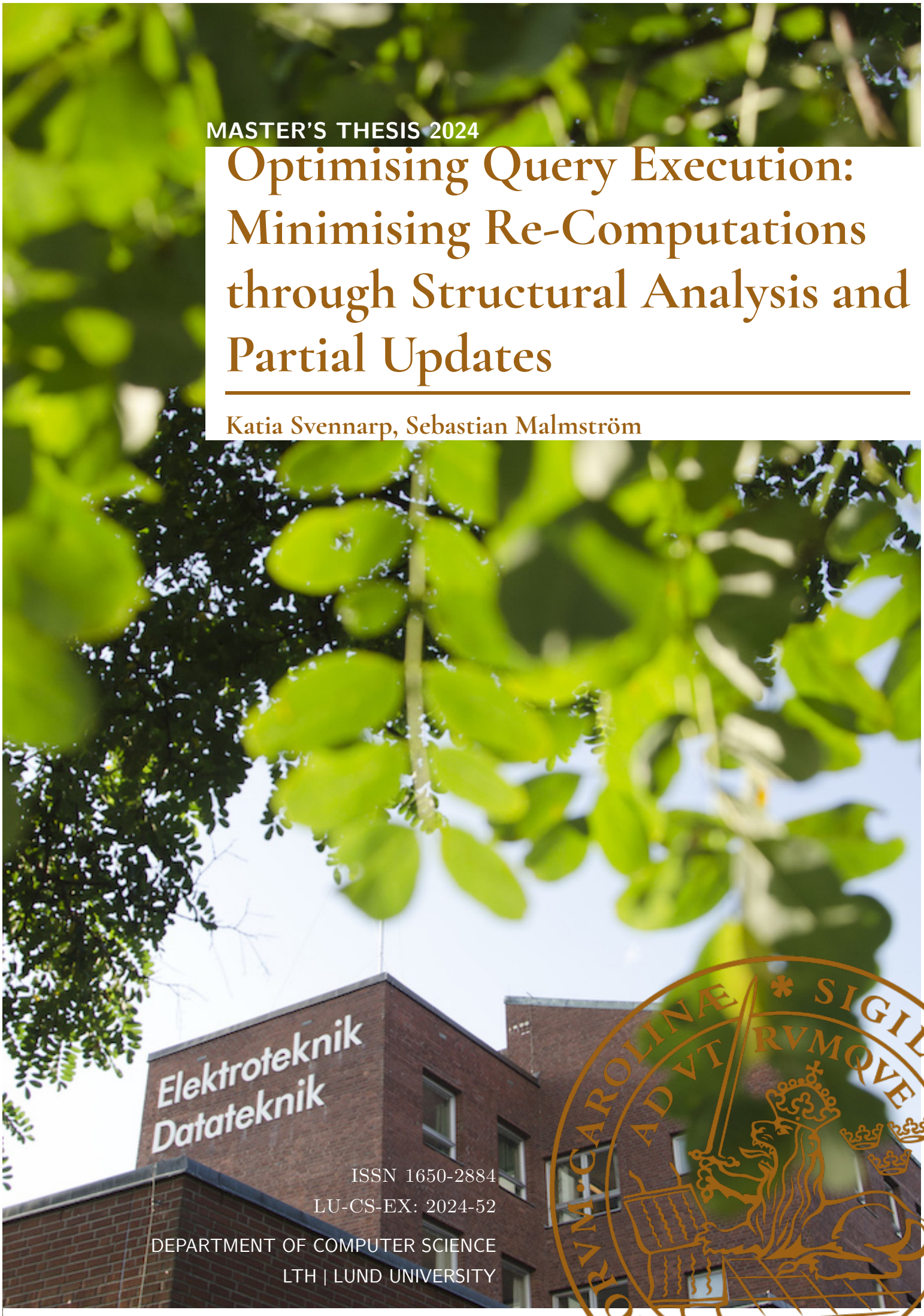

MASTER'S THESIS 2024

Optimising Query Execution: Minimising Re-Computations through Structural Analysis and Partial Updates

Katia Svennarp, Sebastian Malmström

ISSN 1650-2884
LU-CS-EX: 2024-52

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-52

Optimising Query Execution: Minimising Re-Computations through Structural Analysis and Partial Updates

Optimering av frågeexekvering: minimera omberäkningar genom strukturanalys och partiella uppdateringar

Katia Svennarp, Sebastian Malmström

Optimising Query Execution: Minimising Re-Computations through Structural Analysis and Partial Updates

(Strategies for Dynamic Refreshing and Re-Computations)

Katia Svennarp
ka7612sv-s@student.lu.se

Sebastian Malmström
se4872ma-s@student.lu.se

August 21, 2024

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

Supervisor: Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

In a world where the volume of collected data increases daily, larger and larger amounts of computations are needed to deliver data to the end-user. This leads to a need for more efficient data handling as well as identifying opportunities for avoiding unnecessary computation.

This thesis examines a view maintenance architecture, which involves precomputing the results of common queries and storing them in a separate data structure. It will analyse how the structure of a query can influence its execution time and explore whether computation can be optimised through partial updates. Two methods have been implemented. The first method updates only the parts affected by new changes. However, determining whether a query needs to be executed can sometimes take longer than running the query itself. To address this, the second method updates the affected parts and executes groups of queries with short execution times.

We evaluated these methods through three distinct experiments. The first two experiments focused on the static analysis of queries, aiming to identify appropriate operations to classify as complex as well as finding thresholds for classifying groups of queries based on individual results. The final experiment addressed the issue of partial updates in the context of view maintenance.

The key findings are that predicting computational complexity is feasible but heavily dependent on the specifics of the database. Implementing logic for deciding which queries to run instead of running them all can save significant time and computations.

Keywords: Partial Updates, View Maintenance, Gremlin, Queries, Optimisation, Classification

Acknowledgements

We want to thank The Case Company for this opportunity and for providing us with help and insight when needed.

Thanks to Flavius Gruian who listened to our problems and quickly took action.

We also want to give special thanks to our supervisor at LTH, Christoph Reichenbach. He came to our rescue in the middle of our thesis and supported us throughout the summer. Before his guidance, we felt lost, but with his help, we could finally see the light at the end of the tunnel.

Contents

Chapter 1
Introduction

1.1 Context

The demand for more efficient data handling is essential in a world where the volume of collected data increases daily. This rising volume necessitates larger and larger amounts of computations to deliver data to the end-user. On a large scale, data centres storing this vast amount of data are responsible for a substantial and escalating carbon footprint. Bhattacharya et al. illustrate how these centres contribute to 2% of the world's carbon emissions, a figure that is steadily increasing [13]. One strategy to alleviate this impact is by reducing the volume of computations or enhancing their efficiency.

The thesis is done at a company, hereby referred to as The Case Company. The Case Company is trying to reduce its volume of computations and enhance its efficiency. They operate in the realm of IT Inventory with their tool, hereby called *The IT-Inventory System*. The IT-Inventory System connects to various data sources and then polls them for useful information. Due to the wide range of information collected, it uses an in-house graph database, designed for general data storage. The IT-Inventory System analyses the connections from their graph database to give valuable insights about the IT environment of the current user. The tool is run locally by each customer so that no data is needed to be transferred or stored in any cloud service, as to increase their data security and governance.

The Graph Database in The IT-Inventory System integrates multiple data sources and offers various user views. As databases are updated, these views must also be refreshed. Currently, there are two main approaches to achieve this:

- full re-computations
- partial updates

Full re-computations involve re-executing all queries from scratch whenever the underlying data changes, which can be highly resource-intensive. On the other hand, partial updates aim to update only the parts of the view that are affected by the data changes. While partial updates can be more efficient in theory, they are not always preferable due to complexities in

implementation, maintenance overhead, and sometimes limited performance gains in certain scenarios. Efficient management strategies are crucial not only for optimising performance but also for mitigating the environmental impact of data centres, which contribute significantly to global carbon emissions. While The Case Company currently employs full re-computations, this thesis will explore the feasibility of transitioning to partial updates to mitigate these challenges. This exploration will primarily focus on analysing the trade-off between performing partial updates and conducting a full re-computation. The analysis will consider both the time involved and the number of computations required.

1.2 Problem Statement

This research aims to develop a method that minimises re-computations and enhances efficiency. The current framework re-computes all of the queries when the database is updated and a partial update could therefore be a better option.

This research will include an investigation of the possibility of predicting the execution time from the structure of a developer-written query. From other programming languages, for example, Java, we can tell that using a function to sort data will take a longer time to execute than a simple get function. This structure could be the same or similar in a graph query language, which is why it will be investigated. From this, it would potentially create the possibility to tell which query should always be executed, as it would have a low execution time, and which should only be executed if needed as they have a higher execution time. This could enable the development of a strategy for which queries to re-run and which not to. This strategy would eliminate unnecessary processing steps and further optimise The IT-Inventory System's performance.

The research will also analyse the potential benefits of doing a partial update by only executing the queries that are affected by the new information from the update of the database. There are multiple avenues to enhance the updating procedure while ensuring data accuracy. An effective strategy could be to collect information from the changed data and from that information run relevant queries. The effectiveness and number of computations could have a large overhead, meaning it could potentially be better for some database sizes than others. Therefore, different sizes of databases will be tested.

The research questions are:

RQ1 What can the structure of a query in a graph database tell us about the execution time?

RQ2a What is the trade-off between running all queries versus implementing logic regarding what query to run?

RQ2b How does the size of the database affect the trade-off?

1.3 Scope

This master thesis with corresponding research questions is limited to a scope which will be described below:

- Only adding vertices
  We will only consider adding a vertex/vertices to the database, meaning deletion or changes of vertices will not be evaluated.

- Not looking at edges
  The tests do not directly address edges in any way. However, when running queries, branching queries will be involved, indirectly incorporating edges.

- Limited to one query language
  In our thesis, there will only be tests regarding one query language, Gremlin. This means that the scope of RQ1 is only regarding the instance of Gremlin and is not intended to be generalisable to other languages.

- Limited to in-house database and system
  RQ2 depends on the implementation of the in-house database and system that is available to us. Therefore the results will not be able to provide an answer for the general case.

1.4 Outline

This report starts with the context and problem statement in Chapter 1 *Introduction*. It is followed by Chapter 2 *Background* that gives the background and necessary theory to understand the rest of the report. The *System Architecture*, Chapter 3, explains The IT-Inventory System that is used for this thesis. The experiments needed to evaluate and answer the research questions are explained in Chapter 4, *Experiments*. The result of the experiments is explained in Chapter 5, *Results*. In Chapter 6, *Discussion*, the result is then discussed and analysed together with the accompanying research question. Lastly is the *Conclusion* (Chapter 7) where the result and discussion are tied to the conclusion regarding each research question.

Chapter 2
Background

This section provides insights into existing concepts and theories that are relevant to this thesis.

2.1 The IT-Inventory System

The tool is an IT Inventory management system that is built on top of their in-house developed graph database. By using a graph database as its storage solution, the tool can take in any general IT system and represent their data as a graph. The Case Company has developed a system that automatically takes inventory from customers' databases and stores everything strictly on-prem, meaning The IT-Inventory System is installed and run locally, for security reasons. The company have chosen to develop an in-house database as they can customise it and add features as they please. A more detailed explanation of The IT-Inventory System architecture can be found in Section 3.

2.2 Graph DBMS

Relational databases may arguably be the most used type of database, which stores all data in tables, rows and columns. In a graph database, the data is instead stored in a graph structure, built by vertices/nodes, edges/relations and properties/attributes. These entities work as follows:

- **Node/Vertex**
  Vertices represent entities or instances in the databases. The same type of vertex might be present but represent different entities, such as different persons being represented by different vertices of the type `Person`. It can be related to a row in a relational database or a document in a document-store database.
- **Relation/Edge**
  Edges have no direct representation in relational databases but they connect vertices

through relationships in a graph, the closest representation being that of foreign keys connecting entries in two tables. Depending on the database management system (DBMS), these can be direct or indirect. Additionally, some DBMSs allow these to be labelled, while others do not.

- **Properties/Attributes**
  Attributes are Key-Value pairs that belong to a vertex. For example, if there was a `Person` vertex this could be the name or address of that specific person.

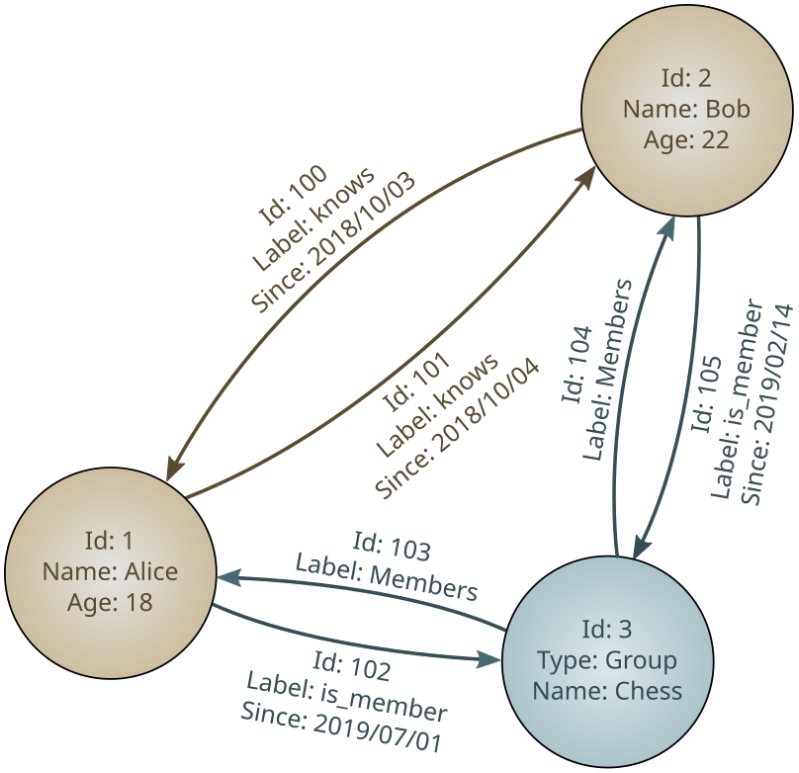

Figure 2.1: Visual representation of a graph database by Ole Mussmann[1].

The edges relate different vertices together, allowing them to represent information and their relations to each other[2]. An example of each of the components composed together into a graph can be seen in Figure 2.1.

2.2.1 Query Language (Gremlin)

To perform queries in a graph database it is convenient to use a query language, similar to SQL (Structured Query Language) which is used in relational databases. Today SQL is one of the most used languages for relational databases and it is originally based on the theory of relational algebra and tuple relational calculus[3]. Examples of query languages for a graph database are Cypher, SPARQL, Gremlin, GraphQL and AQL. At The Case Company the chosen language is Gremlin. Some practitioners consider Gremlin especially language- and

[1] Ole Mussmann - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=87002327
[2] *Graph database*. [Online]. Available: https://en.wikipedia.org/wiki/Graph_database Accessed: 2024-08-20.
[3] *SQL*. [Online]. Available: https://en.wikipedia.org/wiki/SQL Accessed: 2024-08-20.

database-agnostic[4]. Gremlin is developed by Apache TinkerPop and is a graph traversal language. It lets its user express traversals or queries on the underlying property graph. Every step is an atomic operation on the graph. It can either be map-step (transforming the objects in the stream), filter-set (removing objects from the stream), branch-step (branching the stream on a predicate) or sideEffect-stream (computing data about the current stream). These objects that are used to filter are from the graph such as vertices, relations and properties. Each step only affects the objects in the stream and not the actual graph. Using these fundamental principles, the developer can compose several steps to answer questions from the graph [5].

Gremlin is written as a chain of operations/functions on the incoming stream from the graph. Each operation receives the data from the output of the previous step and then does its transformation or filtering then gives the resulting stream to the next step in the chain.

```
gremlin> g.V().has('name', 'hercules').out('father').out('father').values('name')
==>saturn
```

Figure 2.2: Example of a traversal in Gremlin taken from JanusGraph[6].

Dissecting the query in Figure 2.2, each operation and explaining what it does:

- g → means that the starting point is graph g
- V( ) → transforms the input into the vertices of the graph
- has('name', 'hercules') → transforms the stream to only include vertices with the property name that equals hercules (Filter-step)
- out('father') → on the incoming stream from the last operation it returns all vertices that is connected by a relation father (Branch-step)
- out('father') → does the same again but this time on the fathers (Branch-step)
- values('name) → returns the name property on the current stream of vertices (Map-step)

In summary, the query in Figure 2.2, gives the name of all persons that are Hercules's grandfathers[6].

2.3 View

When optimising querying in databases there are different ways to go about it, one being creating *Materialised Views* of commonly used data. It is the technique of pre-computing the

[4] *Navigating networks: An introduction to graph query languages*. [Online]. Available: https://linkurious.com/graph-query-languages/ Accessed: 2024-03-26.

[5] *Apache TinkerPop™*. [Online]. Available: https://tinkerpop.apache.org/ gremlin.html Accessed: 2024-03-26.

[6] *Gremlin Query Language*. [Online]. Available: https://docs.janusgraph.org/ getting-started/gremlin/ Accessed: 2024-03-26.

most usual queries and storing the result as a snapshot in another data structure. This is to lower the response time of a query by already having parts of it done. Views can according to Currim et al. reduce the time for an analytical process of the data from hours, or in some cases days, to minutes or seconds [5].

The timing of creating the snapshot depends on the use case for the specific Materialised View. Some examples can be to update the view when there is bandwidth available for something more mobile, to push out updates when there has been an update to the database or to update it when a user requests it manually.

As with most optimisations, some caveats needs to be considered:

1. What data should be stored in a view (View Selection)

2. How to keep it up to date with the underlying database (View Maintenance)

2.3.1 View Selection

Currim et al. try to solve the first problem regarding what views to choose by quantifying different metrics and then using different methods of minimising the size of the view while still including enough useful data [5]. Failing to choose the correct data to materialise reduces the benefits of the optimisation as new data will still be required to load when users request. For this thesis we consider view selection to be out of scope.

2.3.2 View Maintenance

The issue of keeping the views updated with correct data is directly aligned with the focus of this thesis, as it represents a specific instance of the view maintenance problem. Since various types of databases function differently, they require distinct optimisation strategies. Not updating the data in the view as necessary results in the user not receiving up-to-date data and could require more time spent on requests to the database.

A lot of research has gone into relational databases and how to keep the views updated. For example, Qian and Wiederhold, Griffin et al. proposed an algorithm that models updates as incremental changes to relations in the database. Their algorithm then derives the minimal incremental relational expression that needs to be recomputed through the use of update propagation [8]. An updated version that tries to fix some apparent errors has been developed by Trickey et al. in their paper where they explore an evolution of the prior method. [3].

Another type of database is object-based. Also here has research gone into how to propagate changes into the created views. Alhajj and Polat proposed a solution using classes that include a modification list and a time span with all changes since the view was created or updated [1]. The idea of keeping a list of changes could be applied to other types of databases to create an effective system for view maintenance.

In the case of a more specific use of the View paradigm, there is the problem of using a remote database for mobile use. As the cost of using the network on a mobile unit is higher and it is more unreliable a view is often used. The mobile unit creates a view of the most used data in local storage to resolve these issues. Then the problem of how to update the view comes into play. K. Lee et al. suggest using a pull-based solution of the client requesting updates instead of the server pushing out changes to all clients [7].

2.4 Query Optimisation

Query optimisation is a broad field of research, that can be narrowed down into branches in different subjects based on information such as database type, type of optimisation and different algorithms. Ionnadis explains the basic components of optimisation in a single select-project-join query in a centralised relational DBMS[6]. For the thesis, the most important part of the paper describes the *Rewriter*. This part of the optimiser applies transformations to the given query to make it more efficient while still producing the same results. It does not take into account the query cost, what kind of DBMS or what kind of database, instead only looking at the static characteristics of the query[6]. Some more modules described by Ionnadis are

- The *Planner*
  Searches for the best plan of action for the given query by for example creating a search tree of all paths
- The *Algebraic Space*
  Determines the order of the operations of the queries sent to the Planner.
- The *Method-Structure Space*
  The execution order given by the Algebraic Space determines the implementation choices to be done.
- The *Cost Model*
  Estimates the cost of the plans made from arithmetic functions the Cost Model contains.
- The *Size-Distribution Estimator*
  Estimates the size of queries and subqueries. Also, the frequency distributions of values in the results help the Cost Model to estimate the cost.

When evaluating the database's performance in executing the written query, it is reasonable to consider the runtime of various queries that incorporate different factors that might impact execution time. In one such study, presented by Alamsyah et al., the results showed that a query containing a join, or multiple MATCH statements in Cypher (or general graph query language), took significantly longer to execute with some being a difference of almost 10x [11].

In a relational database, a SELECT statement could be classified as the baseline simple query. Then it can be seen that a join increases the processing time by about 6x and a query that also sorts the data takes about 20x the baseline. The most complex from here was a query that included some arithmetic, at about 30x the baseline [10].

Hayath et al. explored how rewriting SQL queries could lead to an increase in performance and speed [14]. They mention that the query writer needs to decide what optimisations to use depending on the situation at hand. Some things they tried were using ***REGEXP_LIKE*** instead of ***LIKE***, and instead of using a long list of ***IN*** using a temporary table and ordering ***JOIN*** from largest table to smallest.

Parallelism is a key aspect of software optimisation, including in databases and queries. Sadat et al. explored parallel environments in their paper [9]. They discovered that highly dynamic query scheduling, which adapts based on observed execution times, outperforms static methods in both complexity and load balancing. However, they noted that the overall

speedup is limited due to several factors, including the overhead of partitioning data and aggregating results, as well as issues with locking, such as waiting for the final processor to complete and competition for shared system resources.

In all theoretical ideas and inventions, there is the problem of translating them to practical use. Stuefer et al. researched how to handle optimism and pessimism when writing optimisations. They landed on the idea that it is better to build for the robustness of the execution plans instead of striving for unobtainable optimisations [4].

2.5 Asymptotic Complexity

Another crucial aspect of query optimisation involves analysing the execution time and computational effort required to process a query. This can be effectively assessed through the concept of *asymptotic complexity*, which examines the worst-case scenario concerning how frequently each element of a dataset is accessed. Asymptotic complexity provides a framework for understanding how the resource demands of an algorithm scale with the size of the input data.

For instance, if the execution time of a query grows disproportionately compared to the increase in data size, it may indicate inefficiencies in the algorithm's design. Consider an algorithm with a time complexity of $O(N^2)$; in this case, the effort required to compute results increases quadratically with the size of the dataset. In contrast, algorithms with more efficient time complexities, such as $O(N)$ or $O(N \log N)$, exhibit a more manageable growth rate, scaling linearly or logarithmically. As Skeppstedt notes, these differences in asymptotic complexity can significantly impact the performance and scalability of algorithms [12].

Chapter 3
System Architecture

This chapter describes and summarises the pipeline of The IT-Inventory System. The pipeline is created to make the user of the software able to see up-to-date data from multiple sources in one central hub. The pipeline is divided into three stages; Data Collection, Query Execution and Indexing.

In the first stage, searchable data is retrieved and updated through a *Data Collection* (Section 3.1). The Data Collection is either started manually from The IT-Inventory System's user interface or automatically by the system during the night. The subsequent stage involves running all the queries on the graph for the Data Collection. In the final stage, a *SearchTable* (section 3.3) is built from the extracted query data and is used by customers to perform their searches.

3.1 Stage: Data Collection

A *Data Collection* is a method to retrieve data from all sources connected to The IT-Inventory System. The *Indexer* (Section 3.3) initiates its operation through the Data Collection, and the indexing process determines which queries to run. Therefore most of the following processes are run within the Indexer thus the Data Collection.

When The IT-Inventory System wants to consume new information from data sources a Data Collection is started. From a Data Collection it is possible to get the changes from the underlying data sources and from that it is possible to retrieve which vertices the change belongs to. This information is pivotal to the thesis as it makes it possible to only execute the queries that fill the need of the specific vertex.

3.2 Stage: Query Execution

There is a fixed number of executed queries in the current system as all of the queries are written by the developers at The Case Company. This means that no customers can control the data gathered from the database, in the case of writing custom queries. While The

IT-Inventory System's queries were originally all in Gremlin code, The Case Company has manually translated some of them into equivalent Java code as an optimisation. The programmatic queries are Gremlin queries that the developers have optimised into Java, something that needs to be done by hand. Therefore not all queries can be or are worth it to translate to programmatic. During the testing and when gathering all queries around 46.8% of the queries were programmatic. When running The IT-Inventory System the queries are all parsed and optimised by the compiler as part of the start-up process. The query execution is part of the indexing process, but can also be seen as an independent process depending on the perspective.

A *QueryGroup* is a partition over the set of all The IT-Inventory System queries. A QueryGroup and a vertex are connected in such a manner that it is possible to identify the specific QueryGroup to which each vertex belongs. The queries in a QueryGroup are grouped according to the resulting name of the data retrieved. For example, multiple queries retrieve the name from different kinds of vertices and could therefore be under the group with the name *Computer*. All of these are run together and are therefore one group of queries. There are 68 QueryGroups, each with a different amount of queries included. In Figure 3.1 it can be seen how many queries are in different groups. A lot of the QueryGroups include the same queries as similar information needs to be retrieved but for different assets, for example, name and serial number. Therefore it is seen in the figure that 23 QueryGroups run 82 queries. All of the existing QueryGroups in the current system are used in this thesis.

3.3 Stage: Indexing

The end-user does not perform any direct search queries on the graph but instead on a structure providing indexing and searching in documents. This structure is hereby referred to as the *SearchTable*. This table contains all the information that should be searchable by the customer using The IT-Inventory System. For reference, this solution closely resembles the use of a *View*, as explained in Section 2.3. The problem of choosing what to have in the View does not apply to The IT-Inventory System as it is chosen from the business side of the company, as the customer is shown what they pay for and what the company offers. The larger problem is how to keep the SearchTable (View) updated to reflect the database.

As mentioned above, The IT-Inventory System relies on a set of predefined queries, hardcoded into the system. These queries are executed against the data stored in the graph database. The QueryGroups executes its corresponding queries which then extracts the data from the graph database. The gathered information is then saved and indexed in the SearchTable.

The *Indexer* takes the data from the Data Collection and indexes it. The indexer is automatically run after each Data Collection and it works by iterating through the QueryGroups (Section 3.2) and retrieving the relevant queries needed to gather data for the SearchTable. An overview of the dataflow of the system can be seen in Figure 3.2.

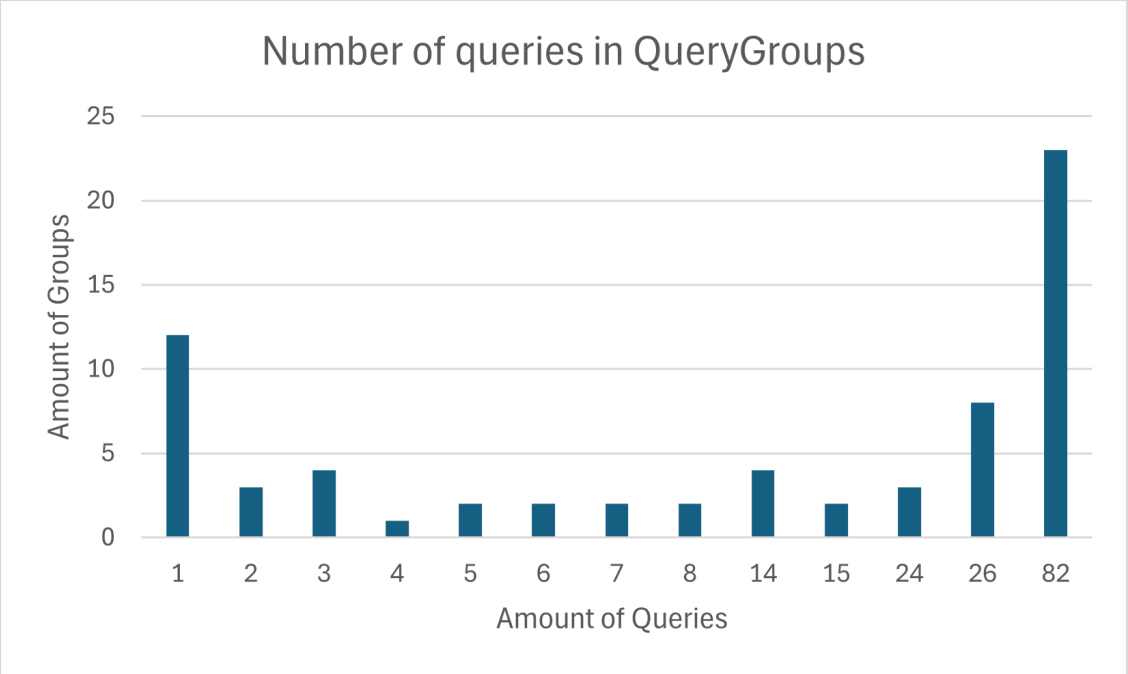

Figure 3.1: Overview of how many queries a QueryGroup have in general.

3.4 Integrating Implementation to The IT-Inventory System

To answer the research questions a new implementation of indexing was introduced to The IT-Inventory System. As the queries are executed between the SearchTable and the QueryGroups, the suggested implementation for handling partial updates would be positioned between them, see Figure 3.3. This means that the graph itself will not be altered to optimise the building of the SearchTable. Instead, the focus lies on lowering the amount of queries run on the graph. In total, there are two different methods to be tested and implemented into the system. The first only uses the vertex differences to see what QueryGroups are currently required. The second method is built on top of the first and includes a system of classifying the queries. These methods are independent of each other regarding how they are integrated into the source code, therefore no processing is needed to decide between the two methods. Both of these are described in more detail in Section 3.5.1 respectively Section 3.5.2.

3.5 The Implementation

The logic for determining which query to run is implemented between the SearchTable and the QueryGroups, see Figure 3.3. In this layer, it is possible to decide which QueryGroups to execute and it is the QueryGroups that return the result to the SearchTable. The implementation will be started through the process of a Data Collection, as described in Section 3.1, the component that initiates the indexing job which in turn initiates the execution of QueryGroups.

Two different approaches are developed for the re-evaluation optimisation layer, the

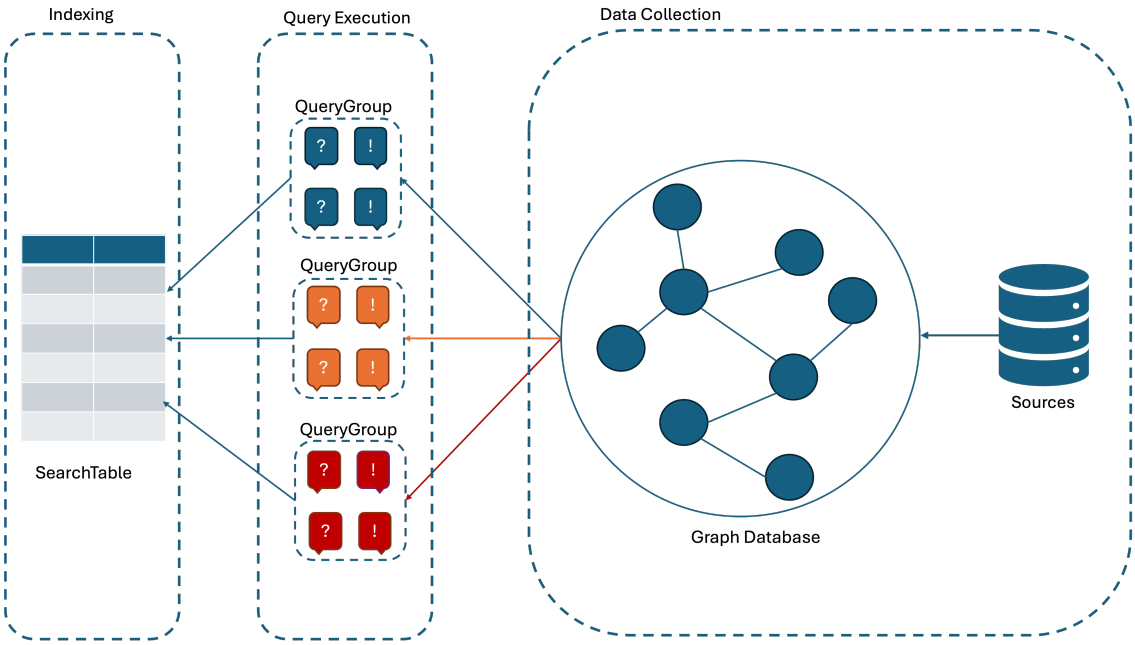

Figure 3.2: Overview of the dataflow of the system.

approaches are going to be named *Method 1* and *Method 2*.

3.5.1 Method 1: QueryGroup Filtering

The approach for Method 1 involves identifying which QueryGroups the changes are connected to and then only executing those relevant QueryGroups. Functionality for extracting the changed vertices already exists in the system but the data needs to be propagated. The approach relies solely on modified vertices, without considering other elements like edges.

3.5.2 Method 2: Conditional QueryGroup Filtering

The approach for Method 2 is the same as for Method 1 but with the addition of classification. The justification for this approach is that the overhead to compare the changes might make it faster to just run the simple ones directly. A classification of all queries is done when The IT-Inventory System is starting. The classification operates in the following manner: if a query is slow (the threshold for a slow query will be analysed in Section 5.3) it will be classified as complex and if not it will be classified as simple. The intuition came from asymptotic complexity (as explained in Section 2.4), the simple one can be represented by an asymptotic complexity of O(1), constant, while a complex one is a faster-growing function. For Method 2 the approach is to run all simple queries but only run the complex queries if they affect the changed vertices. The classification happens during the start-up of the Indexer and the queries are therefore classified in time for the comparison. A pseudo-code of the algorithm for Method 1 and Method 2 can be seen in Listing 3.1.

```
def processQueryGroup(queryGroup, changes):
    # Method 2: Added a conditional check for complex queries
    if isComplex(queryGroup):
        # Method 1 & 2: Checking vertex differences
        addedVertices = getAddedVertices(changes)
```

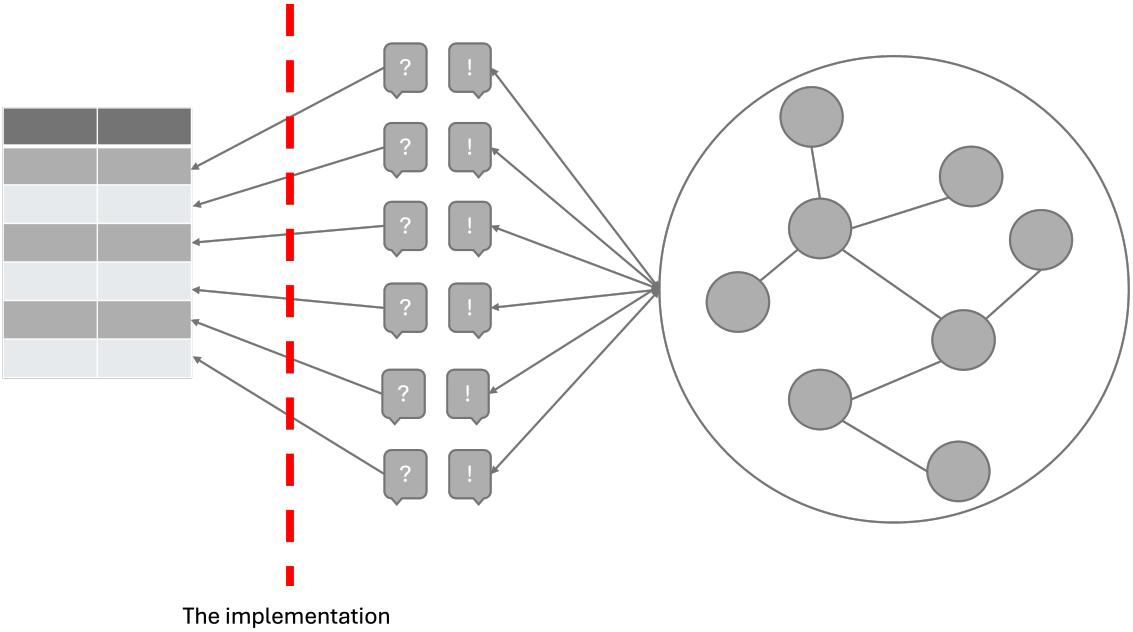

Figure 3.3: Our re-evaluation optimisation layer.

```
        if queryGroup belongs to vertex in addedVertices
            runQueryGroup(queryGroup)
            updateSearchTable()

    # Method 2: Always run the QueryGroup categorised as simple
    else:
        runQueryGroup(queryGroup)
        updateAffectedPartOfSearchTable()

def main(allQueryGroups, changes):
    for queryGroup in allQueryGroups:
        processQueryGroup(queryGroup, changes)
```

Listing 3.1: Pseudocode for the implementation of Method 1 and Method 2

Chapter 4
Experiments

This section will give insight into our approach to the different implementations and tests that were conducted during this thesis. Below are our research questions and the belonging experiments for answering the questions.

RQ1 What can the structure of a query in a graph database tell us about the execution time?

(a) Prototyping: Basic Gremlin (Section 4.2.1)

(b) Test: Classification of Queries (Section 4.2.2)

(c) Test: Classification of QueryGroups (Section 4.2.3)

RQ2 What is the trade-off between running all queries versus implementing logic regarding what query to run? How does the size of the database affect the trade-off?

(a) Test: Comparison of original implementation vs Method 1 and Method 2 (Section 4.3.1)

To answer RQ1 we needed to analyse the structure of a query. In this case, Gremlin Queries. We did some prototyping to decide what the query structure could tell us and how to move further with more experiments. Our prototyping revealed that we could identify queries with shorter and longer execution times. This allowed us to categorise them into simple and complex queries, with simple queries having shorter execution times. We then tested the accuracy of our classification on individual queries and conducted another test to classify QueryGroups.

To answer RQ2a we needed to implement logic regarding which queries to execute instead of executing all of them. For this purpose, we implemented Method 1 and Method 2 described in Section 3.5.1 respectively 3.5.2 above. For the experimental setup, we then compared these methods with the original implementation. The comparison included execution time and the number of run queries. The execution time was measured from the start of indexing to its completion, initiated by a pipeline update.

When comparing the original implementation with Method 1 and Method 2 we decided to conduct the test on databases with different sizes to be able to answer RQ2b.

4.1 Test Configurations

Below is the information about the hardware and configurations that were used for the experiments and implementations.

4.1.1 Hardware

All software ran on the same unit, an HP ZBook 15v G5. The computer was using:

- Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
- 16,0 GB Ram
- 64-bit Windows operating system
- x64-based processor

4.1.2 Run Configurations

The Java Virtual Machine(JVM) version in use is:

- Openjdk version "11.0.22" 2024-01-16
- OpenJDK Runtime Environment Temurin-11.0.22+7 (build 11.0.22+7)
- OpenJDK 64-Bit Server VM Temurin-11.0.22+7 (build 11.0.22+7, mixed mode)

The IT-Inventory System is given 8GB RAM to work with.

4.1.3 JVM Warm-Up Effect

When using the JVM to compile Java code the compiler does optimisations and dynamic compiling. Blackburn et al. [2] describe the warm-up effect as the phenomenon where the initial compilation of a Java program requires the most work. As the program is repeatedly compiled, it eventually reaches a steady state. All of the tests included running a pipeline update in The IT-Inventory System one after one which could lead to the warm-up effect affecting the resulting data.

Therefore, before starting the testing, a test to explore the effect of the warm-up was run. Meaning, that after how many runs the program gets a stable execution time and is fully optimised. The result shows that there was no significant warm-up effect which can be seen in the result section in Figure 5.1.

4.1.4 Benchmark Data

For the benchmark data, we used two different sets. One was a cache provided by the company, the data sources are not relevant for the optimisation but the technical information can be seen in Table 4.1. This cache contained real data that were used by the developers at the company, which means it includes vertices, edges and properties. These vertices belong to different QueryGroups to cover as many as possible of the queries. However, it was not

Dataset 1	Number of Rows
Alpha	751
Beta	2924
Gamma	13998
Delta	20

Table 4.1: Dataset 1: The different real-life test databases and their sizes.

Dataset 2	Number of Rows
Database 1	100
Database 2	1000
Database 3	10000
Database 4	50000

Table 4.2: Dataset 2: The different test databases and their sizes.

possible to add, remove or change vertices with this cache which is why another set was also introduced. This dataset was a synthetic one and contained four different databases with four different sizes, see Table 4.2. For the synthetic dataset, we used MySQL because the time to setup is very low and data can easily be imported from CSV files. The system can use a MySQL database as a source of data. With the help of a mockdata generator, the MySQL instance could be filled with data. By using MySQL it was possible to add, remove or change vertices, by changing the data in the database and then run a data collection on the source i.e. the MySQL instance. Modifiable data was needed for the last test when the different methods were compared.

The mockdata was generated using an online tool[1]. It was set to generate CSV files according to the format that The IT-Inventory System accepted. Then MySQL was used to directly load these files into the different tables in the databases.

In Table 4.3 it is possible to see which dataset and data sources were used for which test and prototype. Dataset 1 was used for the prototyping of the Basic Gremlin queries and classifying the queries. Dataset 2 was used for classifying the QueryGroups and the comparison between the different methods.

[1]*Data Generator*. [Online]. Available: https://www.rndgen.com/ data-generator Accessed: June 13, 2024.

RQ	Test/Prototype	Data
1	Basic Gremlin (4.2.1)	Dataset 1: Alpha & Beta
1	Classifying Queries (4.2.2)	Dataset 1
1	Classifying QueryGroups (4.2.3)	Dataset 2: Database 2
2	Comparison (4.3.1)	Dataset 2

Table 4.3: Test/prototype with belonging dataset.

Operations
Sorting
Comparison
Branching
Branch Traversing and Merging
Date Parsing

Table 4.4: Identified Slower Operations.

4.2 RQ1: Investigating the Query Structure

To better understand the typical queries and their execution times, several pipeline updates were conducted on Dataset 1. The code of the query, its execution time, and the number of vertices it retrieved were extracted. From the extracted information it was possible to analyse and identify patterns from the query structure which gave a foundation for determining possible classifications. This structure analysis is done to be able to implement Method 2 which aims to classify queries and execute the relevant ones.

From the literature study, some operations had been shown to increase processing time, such as sorting data, arithmetic or joining different tables (branches) [10][11]. By finding the equivalent operations in the in-house database language it would warrant a good pointer on where to start with the classification. At The Case Company, the graph querying language of choice was Gremlin. Therefore there was a necessity to gain knowledge regarding how to transform the queries from the theory to Gremlin.

Based on the literature study, we hypothesised which functions might take longer to execute. From this, we did an iterative process where we manually examined which functions appeared in queries with longer execution times but not in those with shorter execution times. For instance, the function `.out()` was present in most queries, so it was ruled out as having a significant impact on execution time. The operations that we identified to take longer time than average can be seen in Table 4.4.

4.2.1 Prototype: Basic Gremlin

After identifying the operations listed in Table 4.4, which were expected to have longer execution times than average, we needed to verify if indeed they took longer than the average query. Since The IT-Inventory System only includes a predefined set of queries they usually consist of more than one operation. We therefore needed to create our queries that only include the identified operations in a constrained test environment where the whole original pipeline is not tested. These queries were written to have as small amounts of other operations as possible that could affect the resulting measurements. These were also chosen so they gave the same amount of resulting vertices retrieved. If a query impacts more vertices, it could affect execution times. Thus, having the same number of results eliminates this uncertainty.

To establish a baseline, we created one of the simplest possible Gremlin queries. This would be a simple GET of a property on a vertex. This function can be seen together with the created example queries in Table 4.5. The example queries were created from already existing ones in The IT-Inventory System, they were shortened so that only the relevant function was tested.

Function	Query
Sorting	`it.out('parameter').gather{it.sort()}`
Comparison	`it.both('parameter','parameter').property ('parameter')`
Branching	`it.sideEffect{parameter=it.getProperty ('parameter')}.ifThenElse{parameter != null &&parameter.equals('parameter')} {it.getProperty ('parameter')}{it.getProperty ('parameter')}`
Branch traversing and merging	`it.copySplit( _().property('parameter'), _().out('parameter').has('class_', 'parameter').property('parameter')).exhaustMerge`
Date parsing	`it.out('parameter').property('parameter') .transform{try {dateFormat.parse(it).getTime()} catch (Exception e) { null }}`
Baseline	`it.property('name')`

Table 4.5: The created example queries. *it* is a vertex retrieved from the iterator over vertices.

The trial queries were run 13 times each against Alpha and Beta from dataset 1, see Table 4.1, and the gathered data was the execution time of each of the queries (the recorded time only included the query execution) together with the number of found vertices. The amount of runs was to account for statistical variation in execution time and the datasets were chosen as it gave a large spread on the amount of queries ran. The different measured times also stabilised the average execution times of the queries and created a smaller margin of error. The result of the accuracy of the 13 runs can be found in section 5.2.

A summary of the results that are to be used in other tests is that the chosen operations (see Figure 4.4) had on average longer execution time, as can be seen in Figure 5.2.

4.2.2 Test: Classification of Queries

To implement the classification some modifications had to be done to The IT-Inventory System. These changes were only small modifications to an already existing code-base and will be described. In the class containing all the information related to a query, a new field named `ComplexValue` was added. This field is a boolean, defaulting to `False`, and can be set to `True` if the query is complex. Each query starts as a simple one and then is assessed on the first run of the Indexer. The query assessment is based on the operations obtained from prototyping the Basic Gremlin, see Figure 4.5. We use string matching on the Gremlin query for each instance of the Query object to determine if it includes any of the predefined complex operations. As a precaution against false positives in the string matching the results all of the queries ran against Dataset 1, Alpha and Beta in Table 4.1, were manually checked to be correctly matched.

Testing the accuracy, defined here as the number of queries classified as complex in the top X% of execution time, of these classifications is done by running 12 different pipelines updates on dataset 1, Table 4.1. We measured the execution time for each query as well as recorded the classification (ComplexValue) of the query. The result can be seen in section 5.3.

4.2.3 Test: Classification of QueryGroups

Since query execution cannot be controlled at the granularity of individual queries, it is essential to classify the QueryGroups and not only the individual queries. The filtering for method 2 can then be applied at a group level. The goal is to determine how many queries within a group are complex, and based on that, classify the entire group accordingly. This means that a threshold must be established to optimally evaluate a QueryGroup. The evaluated thresholds were fixed numbers and percentages.

The implementation was made so that if there were more complex queries than the threshold the QueryGroup was evaluated as complex. The test was conducted by doing five different pipeline updates, using database 2 (Table 4.2), on each threshold. We used the total indexing time for the database as a measurement. In section 5.4 the result of the test can be observed.

4.3 RQ2a & RQ2b: Trade-Off from Partial Updates

For answering RQ2a and RQ2b one test was done. The test involves comparing the original implementation, which runs all queries, with two alternative methods that selectively execute queries for partial updates. This comparison is conducted across databases of varying sizes.

4.3.1 Test: Comparison between original, Method 1 and Method 2

The comparison is between the original implementation vs Method 1 (section 3.5.1) and Method 2 (section 3.5.2).

After consulting with software engineers and the customer success group at The Case Company we decided to test 0, 1 and 20 changes. 20 was picked as it is an estimate of the average amount of added nodes during a nightly pipeline update by customers according to people at The Case Company. By also running 0, a benchmark, and 1, the smallest possible change, we covered a lot of possible high-level cases. The test was conducted on each synthetic database, see Table 4.2, by doing one pipeline update then inserting a new vertex, redoing the pipeline update removing the newly added vertex and redoing the same thing for 20 new vertices. For each test, we measured the time for the indexing (which includes query execution and offloading the data into the searchable document) as well as the amount of executed queries. The result can be seen in section 5.5.

Chapter 5
Results

Below are the findings from the experiments explained in Chapter 4. It will introduce the result and tell about trends and conclusions based on the presented result.

5.1 Warm-up Effect

To see if there were any warm-up effects to the JVM we needed to run some tests. A pipeline update was run 10 times in a row for each method (original, Method 1 and Method 2) on database 2. The result can be seen in Figure 5.1. The left y-axis shows the execution time for the original implementation and Method 2 in ms and the right y-axis shows the execution time for Method 1 in µs. The execution time includes the time for the program to start and the Indexer to finish. The results from the experiment show no obvious warm-up effects.

5.2 Basic Gremlin

To test whether the structure of a query affects execution time, a prototype set of example queries was developed. The execution times for these test queries (see Table 4.5) are displayed in Figure 5.2. The box plot shows that the example queries take a significantly longer time to execute then the baseline query `property`.

5.3 Classification of Queries

From the prototyping, a classification system was made that put the queries in two categories depending on the operation included in the Gremlin. The purpose of this classification was to divide queries that would have a longer execution time to complex and the other queries as simple.

To assess if the classification was successful the queries were ran and then a threshold for what is a long execution time was set up. The thresholds meant that the top 1-10% of

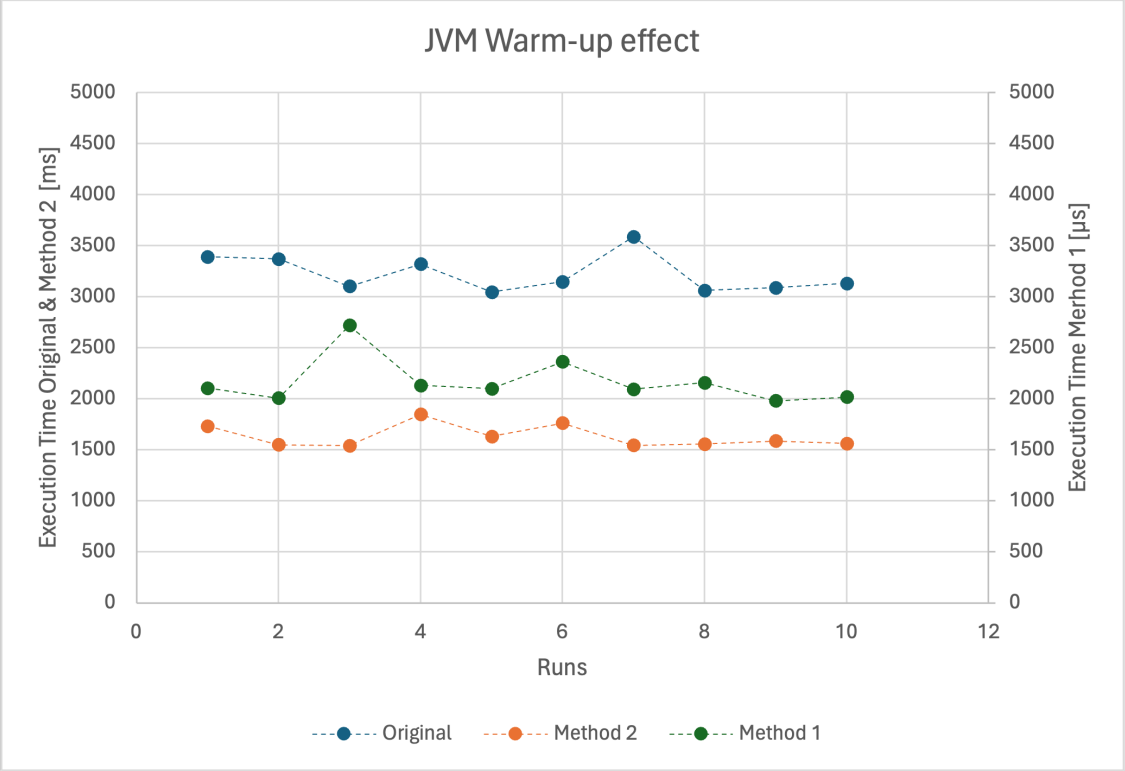

Figure 5.1: The JVM warm-up effect for the three methods on database 2.

queries in execution time were said to be complex. Then it was calculated on what level this was achieved. The Gremlin operations that were evaluated are the ones derived from the prototyping, see Table 4.4. The result of the accuracy from this classification of individual queries with the different thresholds can be seen in Table 5.1. A successful Simple classification means that a query in the bottom (100-X)% of execution time were classified as Simple and a Complex is the opposite, i.e queries in the top X% of execution time were classified as Complex.

The result shows that accuracy for the simple classification is in the interval of 75% to 77% and for the complex classification it is between 34% to 100%. The classification is more accurate for simple queries than complex ones, meaning that it is easier to tell when a query should be evaluated as simple than when it should be evaluated as complex. The accuracy of the classification seems to be better when the threshold is lower. For the threshold between 1-6% the accuracy for both the simple and the complex classification is above 50%. As a guideline the 5% threshold had a cut-off of average execution time of 120,9 ms (12 runs in total) for the queries evaluated as complex.

5.4 Classification of QueryGroups

As an extension of the classification of individual queries, a classification of QueryGroup had to be made for this system, see Section 3.2. The test for QueryGroups included evaluating the best threshold for a QueryGroup to be said complex, see Section 4.2.3. The tests were ran with 0 changes on the data to test indexing time for the original time. The evaluated threshold values can be seen in Table 5.2. From the table, it is evident that setting the

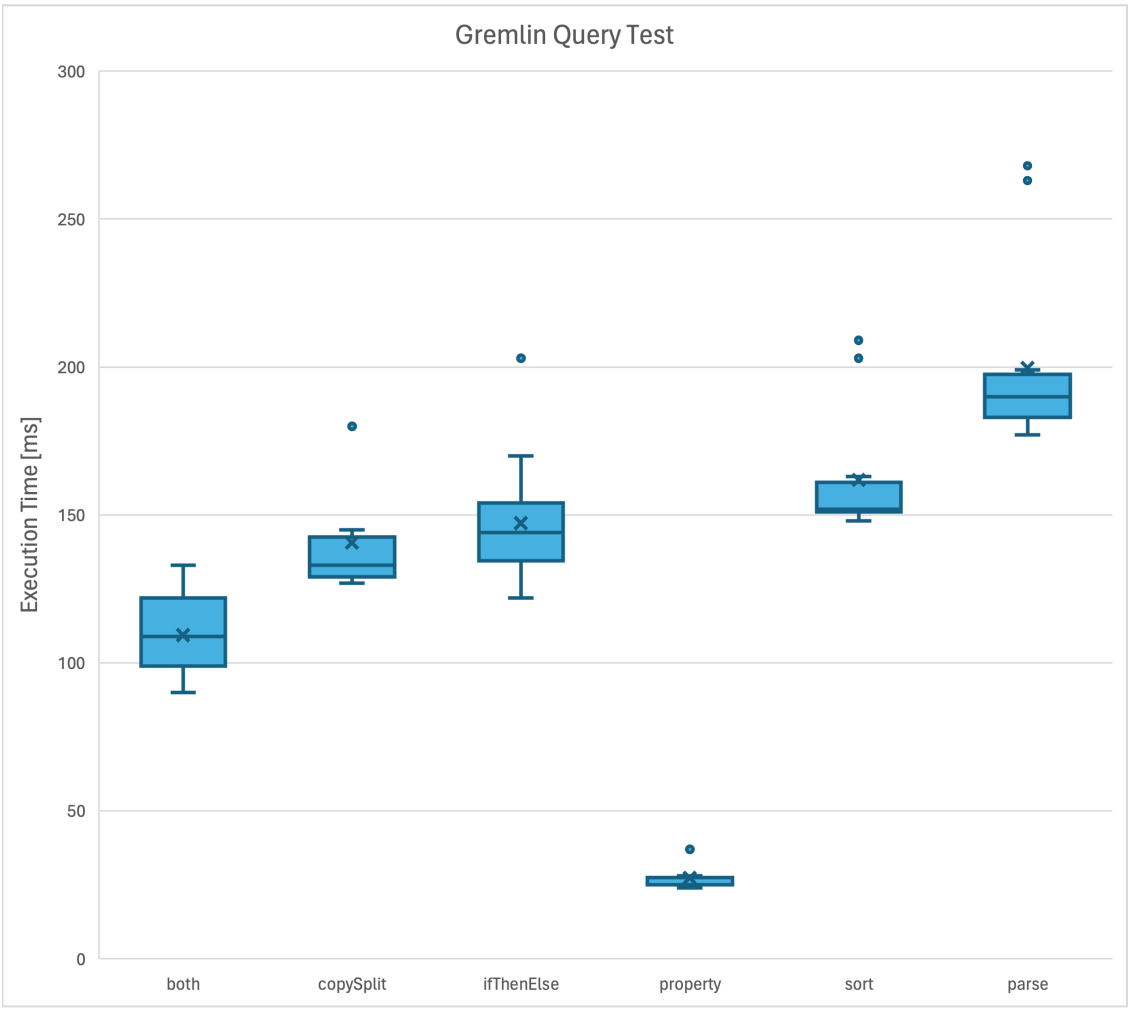

Figure 5.2: A box plot of the execution time [ms] for the gremlin query test. The x-axis consists of the operations from Table 4.4

threshold to 0 would result in no queries being executed. Additionally, thresholds of 1 and 1% exhibited the fastest execution times and involved the second-fewest queries executed within a QueryGroup, both running a total of 14 out of the 68 QueryGroups .

The table also provides the 90th percentile confidence intervals for the average times. For instance, the threshold of 1 has an average time of 876.4 ms with a 90% confidence interval of ± 23.6 ms, and the threshold of 1% has an average time of 906 ms with a 90% confidence interval of ± 21.2 ms. These confidence intervals indicate the range within which we can be 90% certain that the true average execution times lie.

The threshold of 1 has a lower average execution time than the threshold of 1%, although a very small difference. The variation between them on the 90th percentile is so small it can be negligible. Therefore, the chosen threshold that will be used in the implementation for Method 2 will be 1.

X [%] Slowest Queries	Correct Simple Classification		Correct Complex Classification	
	Mean [%]	σ [%]	Mean [%]	σ [%]
1	75,00	0,37	100	0,00
2	75,75	0,19	85	9,05
3	76,22	0,17	83,33	5,56
4	76,42	0,00	70,00	0,00
5	76,54	0,20	64,58	3,77
6	76,44	0,00	57,14	0,00
7	76,13	0,00	47,06	0,00
8	75,91	0,00	42,11	0,00
9	75,65	0,18	37,12	1,77
10	75,47	0,21	34,38	1,88

Table 5.1: Share of correctly classified queries when the threshold is for the X% slowest queries.

Threshold	Average Time [ms]	Number of run QueryGroups [run/total]	90% confidence [ms]
0	10	0/68	± 2.4
1	876,4	14/68	± 23,6
2	1488,8	26/68	± 19,3
5	1884,6	43/68	± 27,2
1%	906	14/68	± 21,2
5%	1135,2	17/68	± 17,7
10%	2022,8	48/68	± 26,5

Table 5.2: Data from the suggested thresholds on QueryGroups. The run number tells how many QueryGroups that were evaluated as complex.

5.5 Comparison between original, Method 1 and Method 2

To evaluate if it is worth implementing partial updates or not a comparison between the original implementation vs Method 1 and Method 2 was conducted. The result of the time for the Indexer to start and finish for the different implementations is seen in Figure 5.3, Figure 5.4 and Figure 5.5. The figures show the comparison with 0 changes, 1 change respectively 20 changes.

From Figure 5.3 it can be seen that Method 1 took significantly less time than the other two implementations with 0 changes. Method 2 took about half of the execution time as the original method. As Method 1 does not execute any queries if there are no changes, the execution time that is seen in the figure is the time it takes for the indexing process to start and finish.

With one insertion (Figure 5.4) the results are similar to the zero insertion case, except for Method 1 which takes a bit more time in this comparison. For each implementation method (except database 4) it seems that the execution time is reduced by half, where the original implementation is the slowest and Method 1 is the fastest.

For 20 insertions (Figure 5.5) it becomes less clear which method is the best. Both the

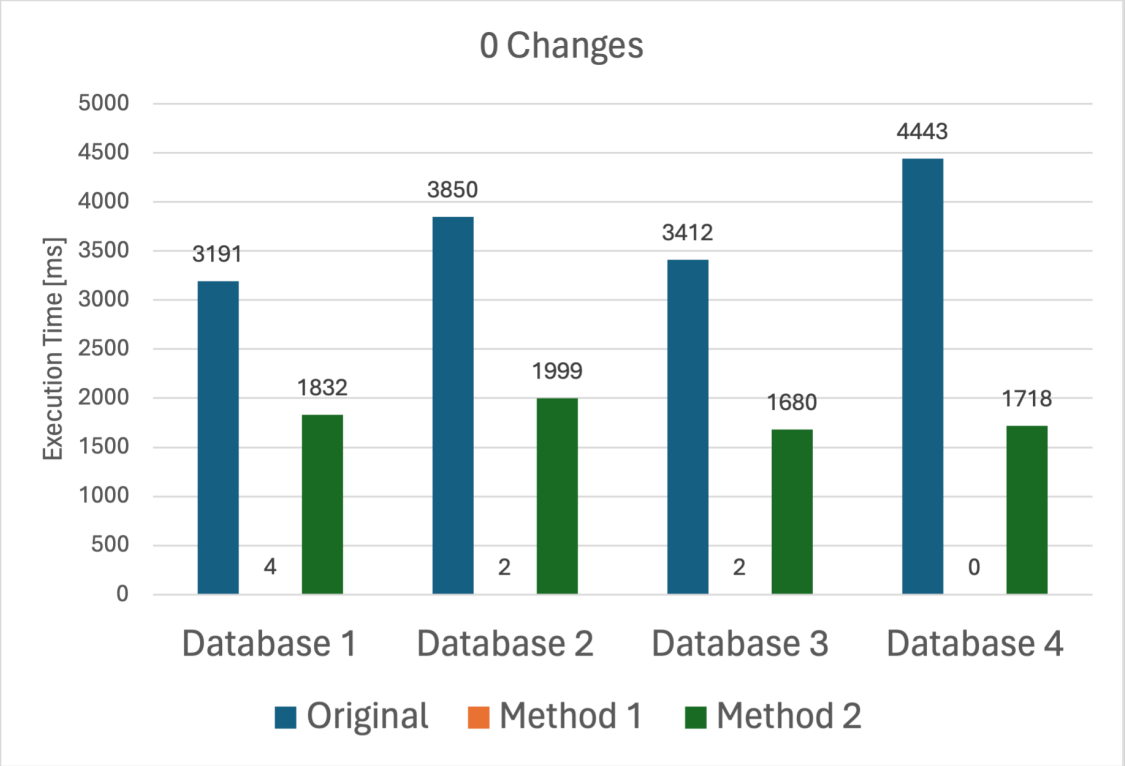

Figure 5.3: The execution time for running the methods on the four different databases with 0 changes.

implementation for Method 1 and Method 2 are faster than the original implementation, but between them, the time difference has been reduced. In the case of database 3, Method 2 has a faster execution time than Method 1.

In Figure 5.6 the number of executed queries for each method can be observed. The executed queries for each database stayed the same for every pipeline update, it only changed depending on the number of insertions. From the figure, it is apparent that the original implementation runs a lot more queries than the other two implementations. The queries are counted so that for each QueryGroup that is run there is a count for each query in that QueryGroup. This means that some queries could be part of multiple QueryGroups and are therefore counted multiple times. Nevertheless, Method 1 runs zero queries when there are no changes, whereas Method 2 runs 159, meaning there are a total of 159 queries included in QueryGroups classified as simple (including duplicates).

Method 2 was based on the hypothesis that verifying a query's connection to new vertices might be more costly than executing the query. To test this, an experiment was conducted to compare the program's execution times (without executing any queries) with and without the connectivity check. With the check, the average runtime was 3.5 ms, with a standard deviation of 1.2 ms. Without the check, the average runtime was 3.8 ms, with a standard deviation of 0.87 ms. The result shows that the check does not increase the execution time of the program in any significant way and therefore in our system, the approach of Method 2 is not worth it.

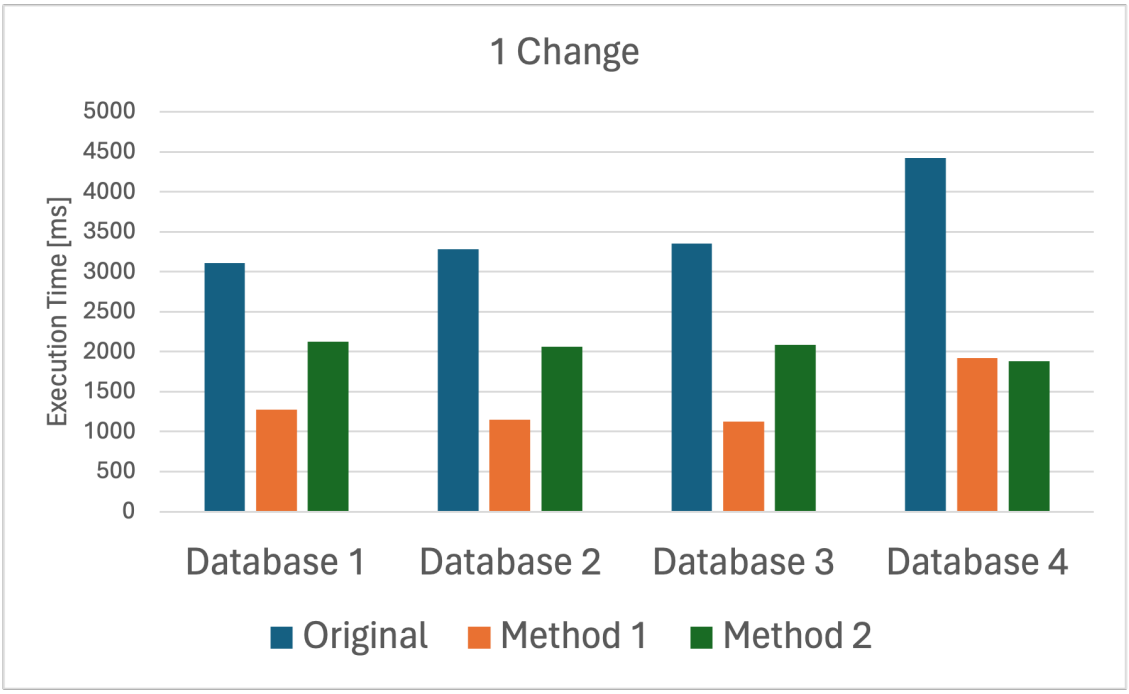

Figure 5.4: The execution time for running the methods on the four different databases with 1 change.

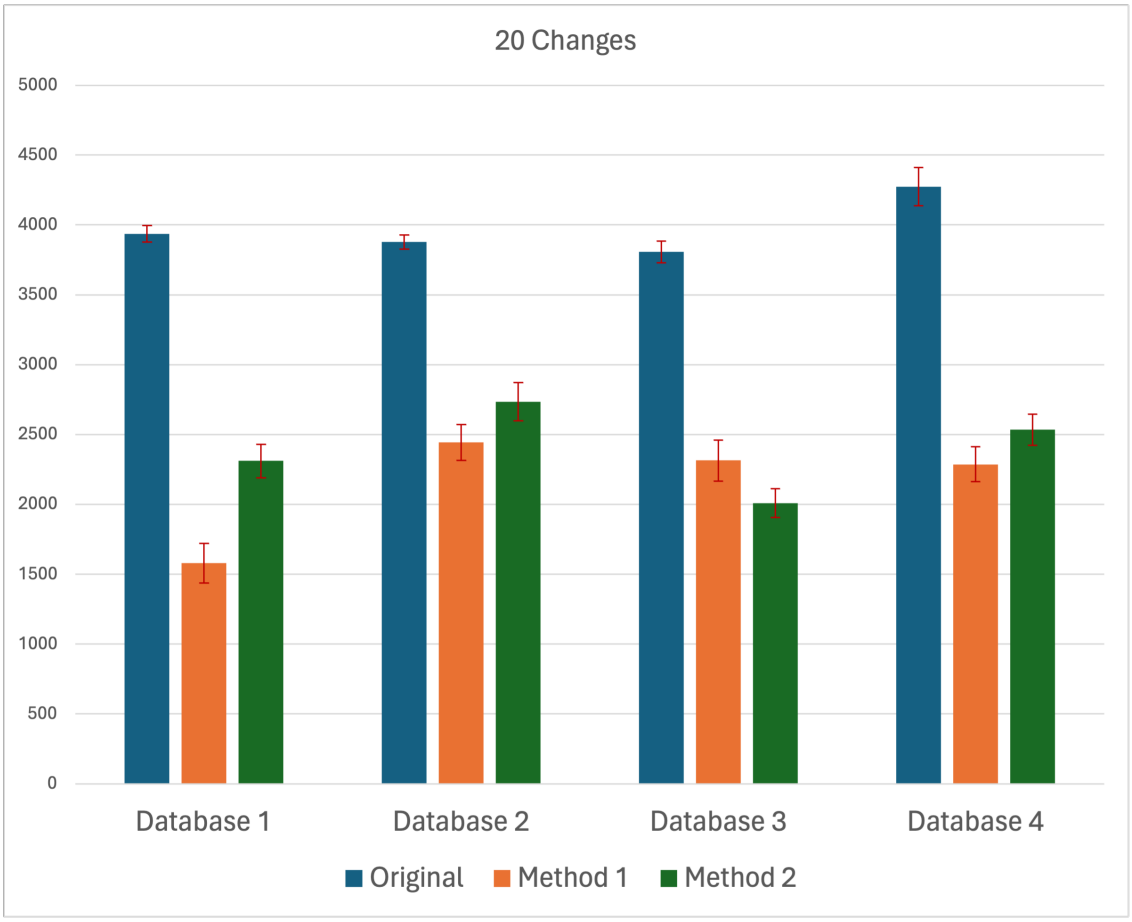

Figure 5.5: The execution times for running the methods on the four different databases, each with 20 changes, along with their corresponding standard deviations.

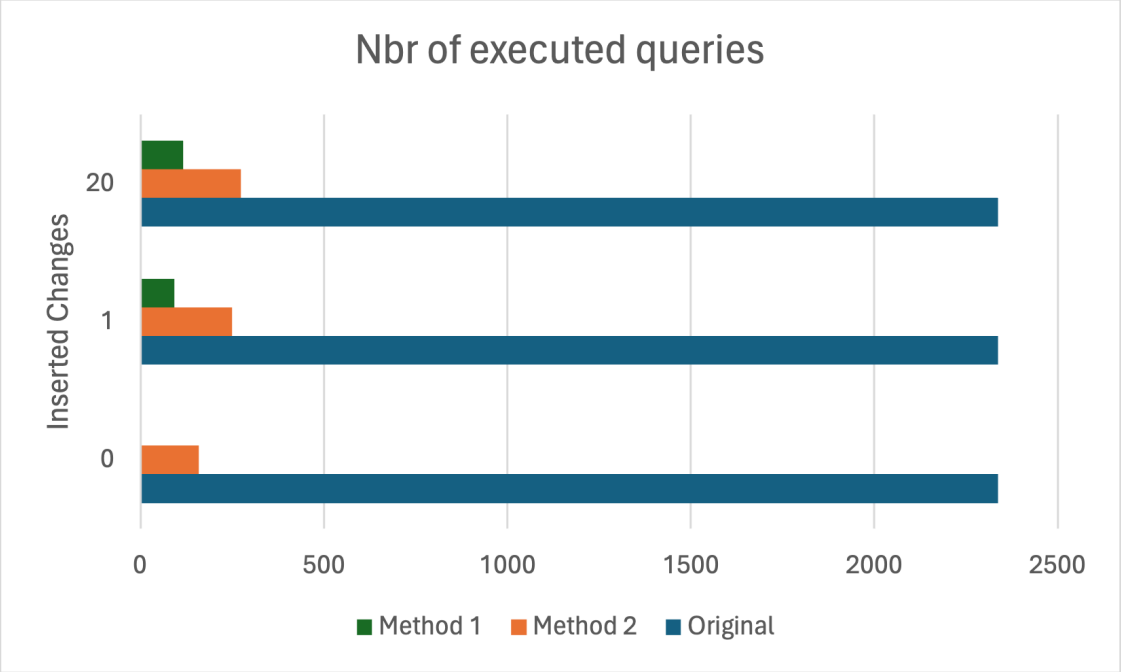

Figure 5.6: The total amount of executed queries on each method for each change.

Chapter 6
Discussion

This chapter will provide a more in-depth analysis of the results from the chapter above. The research questions will be answered in order together with the corresponding result, according to the Scope described in Section 1.3.

6.1 RQ1: Classifying Queries Based on Their Structure

RQ1

What can the structure of a query in a graph database tell us about the execution time?

The result from the classification test suggests that it is possible to assume what query will have a longer or shorter execution time compared to other queries based on the functions it contains. It seems possible to classify the queries based on the various characteristics and patterns observed within the query itself. However, achieving 100% accuracy in this classification is challenging, almost impossible. It was found that it is easier to identify queries with shorter execution times than average, classifying them as simple, than to identify those with longer execution times than average.

Furthermore, the prototyping of which operations have longer execution time, see Figure 5.2, supports the general observation that operations which typically take a long time in conventional programming, such as sorting, traversal, and complex arithmetic operations, also tend to consume significant execution time in query processing. These operations inherently involve more computational steps and data manipulation than other operations, thereby increasing the overall execution time.

The result suggests that queries containing either `copySplit`, `sort`, `both`, `ifThenElse` or `parse` tend to have a longer execution time than queries containing a simple property `get`. As the queries belonged to a QueryGroup there was a need to decide a threshold for how many queries needed to be complex for the entire QueryGroup to be labelled complex.

The result from the observation shows that the threshold with the fastest execution time and the lowest amount of run QueryGroup is either if one query of the entire QueryGroup is complex or if 1% of the entire QueryGroup is complex.

To determine the appropriate threshold for the implementation of Method 2, it was necessary to evaluate performance at different levels. A threshold of 0 resulted in no queries being executed, resulting in Method 2 being almost the same as Method 1. This left thresholds of 1 and 1%, both of which had the second shortest execution times and executed some queries. The decision was made to use a threshold of 1, as it had the shortest average execution time between the two options.

6.2 RQ2a: Performance Impact of Database Modifications on Query Execution Strategies

RQ2a

What is the trade-off between running all queries versus implementing logic regarding what query to run?

There were three different scenarios in the last experiment with the comparison between the original implementation vs Method 1 and Method 2, see section 4.3.1:

- There were no changes made in the database.
- There was one vertex added to the database.
- There were 20 vertices added to the database.

The different cases and their result will be discussed and analysed below.

6.2.1 No changes

In the case of no change, it can be seen in Figure 5.3 that there are large differences between the different configurations of methods. The original implementation that always executes all of the queries, disregarding whether there is a change or not, had by far the longest execution time of the Indexer.

In these tests, Method 2 is always faster than the original but lags behind Method 1. The lag of Method 2 compared to Method 1 is due to Method 2 always running the QueryGroups classified as simple, whereas Method 1 skips executing all groups not affected by any changes. In this case, with 0 changes, Method 1 skips all groups. Consequently, the execution time of Method 1 is nearly zero, as the only recorded time is from running the Indexer itself, without executing any queries.

6.2.2 One change

The results of running all methods with one change can be seen in Figure 5.4. The results from these tests are similar to those from the test with 0 changes. The original is always

slowest while Method 1 is faster than Method 2. A difference is that a change has been inserted in the database, meaning Method 1 has increased in time compared to no changes.

So far it seems better for the execution time to look at the changes directly and only execute the necessary queries rather than to classify the queries.

6.2.3 20 changes

For the case of 20 changes in the database, the result for the execution time becomes more even between Method 1 and Method 2. Figure 5.5 shows the result for running the different methods with 20 inserted changes. Method 1 and Method 2 still outperform the original implementation by a large margin but the distance between the methods has shrunk considerably.

One possible explanation for this is that a significant portion of Method 2's execution time is spent on the classification step. As the number of changes increases, the time per vertex decreases for Method 2. Consequently, Method 2 becomes more efficient and gains time when both methods need to process a higher number of changes.

6.2.4 The Amount of Executed Queries

Another interesting metric to look at is the amount of queries executed for each method. In Figure 5.6 it can be seen that there is a significant gap between the original method and the other two methods and then another significant gap between Method 1 and 2. Interestingly, in the test with 20 changes, the execution times between Method 1 and Method 2 did not differ significantly, despite the difference in the number of queries executed. An explanation for this could be that the query classification effectively identifies and quickly executes simple queries, thereby mitigating noticeable increases in overall execution time, even when complex queries are subsequently processed.

6.2.5 Summary of Discussion for RQ2a

Below is a summary of the discussion for RQ2a, focusing primarily on execution time. These findings are based on the observed data and should be interpreted with consideration of the study's limitations and potential threats to validity.

1. The original implementation exhibits significantly slower execution times compared to Method 1 and Method 2, regardless of the number of changes inserted into the database. This suggests that Method 1 and Method 2 are preferable over the original implementation based on the observed data.

2. Method 2 spends a noticeable amount of time classifying all queries. This becomes evident as the number of changes and vertices increases, causing Method 1's execution time to increase more rapidly than that of Method 2.

3. For fewer insertions, Method 1 appears to be the best option based on our observations.

4. According to the data, Method 1 and Method 2 show similar performance for larger insertions and databases.

5. Method 1 and Method 2 execute significantly fewer queries compared to the original implementation. Specifically, Method 1 demonstrates a notable reduction in the total number of queries executed.

These findings suggest certain trends in query execution times and efficiencies, but further research and testing would be necessary to generalise these results beyond the specific conditions observed in this study.

6.3 RQ2b: Impact of Database Size on Query Execution

RQ2b

How does the size of the database affect the trade-off?

The research indicates that there is no direct correlation between the size of the database and the trade-off observed. This suggests that regardless of database size, the implemented logic remains beneficial in terms of time savings and reduction in the number of executed queries. This can be interpreted as the efficiency gains from the implemented logic being consistent irrespective of database scale.

Interestingly, the execution time increases less than linearly with the size of the database. This could imply that while there is an increase in complexity with more vertices (or data), the efficiency gains from the implemented logic mitigate this increase to some extent. This non-linear relationship might suggest that the overhead and startup costs are managed effectively, leading to a less-than-proportional increase in execution time despite a growing database

The size of the change and database affect the methods in different ways. Method 2 scales better as it always needs to do the classification and therefore performs better for large databases with lots of changes whereas Method 1 has a more persistent execution time. So a larger amount of added vertices does not increase the time to classify them and then Method 2 saves time per added vertices.

A significant portion of the execution time is attributed to overhead and startup costs rather than being directly proportional to the number of vertices. This observation highlights that as the database size grows, these fixed costs become relatively less significant per vertex or operation. Therefore, the efficiency improvements from the implemented logic play a crucial role in offsetting the overall execution time increase.

6.4 Limitations and Threats to Validity

This study included several limitations and potential threats to validity, which are outlined and discussed below.

6.4.1 Hardware and System Constraints

All implementation and testing were conducted on a company-provided laptop, with detailed specifications available in Section 4.1.1. The hardware limitations affected the results,

as using a dedicated server or better hardware could have reduced processing times and allowed for more extensive testing. Running the tests on specialised hardware, such as dedicated GPUs with no interference from other processes, could accelerate specific parts of the program and potentially skew the results. However, comparisons against experiments conducted on the same hardware remain valid, but these constraints suggest a direction for future work.

Additionally, the existing implementation at the company limited the thesis's scope, constraining the extent to which we could conduct tests and integrate our code during the implementation phase. This is because the implementation was situated deep within the abstraction layers.

6.4.2 Test and Data Constraints

Due to time constraints, we conducted a limited number of tests, affecting the validity of our results. For instance, Figure 5.4 shows an unusually high execution time for the original implementation on database 4 compared to other databases. This anomaly indicates that more tests would have been beneficial for ensuring accuracy.

In the comparison experiment, we used a synthetic dataset, which posed a threat to validity since it lacked real data. However, using synthetic data was necessary because we needed data we could modify. The company recommended this approach, and we followed their setup for constructing a MySQL data source.

Our testing was confined to evaluating the addition of vertices to the graph, which is the most commonly used operation. Although testing other operations could provide further insights, it would have extended the testing duration significantly.

6.4.3 JVM Constraints

As discussed briefly in Section 4.1.3 the JVM could be affected by the warm-up effect, due to optimisations and memory handling.

By looking at Figure 5.1 it is possible to see that the warm-up effect has not had a significant effect on the results. The measured times remain relatively stable throughout the tests. If a warm-up effect had been present, we would have seen a rapid increase followed by a sharp decrease in the execution times. Therefore, no warm-up effect was observed.

6.4.4 Generalisability Constraints

The solution is specifically tailored for the company and it is not a plug-and-play option for other software, as it is deeply ingrained into the company product. However, the result shows nonetheless that both time and number of computations could be reduced by implementing logic to decide which queries to run instead of running them all.

6.4.5 Limitations

Time

Due to limitations regarding available time, the amount of tests for each method had to be reduced. During the testing phase, the time required to complete tests increased significantly

beyond our initial expectations, resulting in fewer tests being conducted than anticipated.

Statistical Variation

Not all of the experiments account for the statistical variation by running the same tests for multiple iterations. As explained in the section above the time limited us in how much testing could be done. Therefore some of the tests do have a limited scope regarding accounting for variation in the results.

String Matching

Another limitation is that the classification was done using simple string matching of queries. This means a query may contain one of the keywords in another capacity than as an operation. For example, `it.property("copySplit")`, `property` is an operation that looks after the property of "copySplit", would classify the query as complex while in reality being simple. For this thesis, the results were checked manually to see that there were no false positives for the string matching, but in a more general case this could be of concern.

Chapter 7
Conclusion

In conclusion, classifying queries based on execution time is feasible but heavily dependent on the specifics of the database and query language. Simple queries are generally easier to classify than complex ones.

Implementing logic to determine which queries to run for partial updates, rather than performing full re-computations, is beneficial as it can save both time and computational resources.

However, the effectiveness of this logic can vary depending on the existing implementation. While the size of the database does not impact the trade-off results, it does increase execution times due to a larger number of vertices. Additionally, the extent of changes in the database influences the methods used, with Method 2 incurring higher initial classification costs but potentially yielding lower execution times for updates involving numerous vertices.

7.1 Future Work

A potential area for future research is the classification of queries. These tests have been run on a particular graph database with a particular setup regarding how the queries were grouped, what language they were written in and what operations were chosen. Since it is an in-house database the execution of queries has also been created by the developers at The Case Company. To continue the exploration of categorising queries by their operations it would be good to try it on some different databases. It would be particularly interesting to study the utility of the methods in different application settings with significantly larger change sets. Additionally, doing more extensive research regarding execution time and resource usage could lead to insights into optimisations that could be done before letting the queries be compiled.

References

[1] R. Alhajj and F. Polat. Incremental view maintenance in object-oriented databases. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 29(3):52 – 64, 1998.

[2] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, aug 2008.

[3] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[4] S. Helmer, P. T. Wood, and M. Stuefer. Optimism and Pessimism in Database Query Optimisation. *2024 IEEE 18th International Conference on Semantic Computing (ICSC), Semantic Computing (ICSC), 2024 IEEE 18th International Conference on, ICSC*, pages 269 – 276, 2024.

[5] R. Hylock and F. Currim. A maintenance centric approach to the view selection problem. *Information Systems*, 38(7):971 – 987, 2013.

[6] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121 – 123, 1996.

[7] K. C. K. Lee, H. V. Leong, and A. Si. Incremental View Maintenance for Mobile Databases. *Knowledge and Information Systems: The International Journal of Formal Methods*, 2(4):413 – 437, 2000.

[8] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[9] A. B. M. R. I. Sadat and P. Lecca. On the Performances in Simulation of Parallel Databases: An Overview on the Most Recent Techniques for Query Optimization. *2009*

International Workshop on High Performance Computational Systems Biology, High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on, pages 113 – 117, 2009.

[10] M. Salahat, M. A. Raza, Z. Khawar, J. A. Malik, H. Raza, H. K. G. Nair, and M. Jarrah. Analysis of Query Processing on Different Databases. *2023 International Conference on Business Analytics for Technology and Security (ICBATS), Business Analytics for Technology and Security (ICBATS), 2023 International Conference on*, pages 1 – 7, 2023.

[11] R. J. Sholichah, M. Imrona, and A. Alamsyah. Performance Analysis of Neo4j and MySQL Databases using Public Policies Decision Making Data. *2020 7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE), Information Technology, Computer, and Electrical Engineering, 2020 7th International Conference on*, pages 152 – 157, 2020.

[12] Jonas Skeppstedt. *Algorithms: a concise introduction*. Skeppberg AB, 2020.

[13] M. Rahgouy T. Bhattacharya and X. Peng et al. Capping carbon emission from green data centers. *International Journal of Energy and Environmental Engineering*, 14(4):627–641, 2023.

[14] H. Tm, U. K, M. Shafiulla, and D. Dadapeer. An Overview of SQL Optimization Techniques for Enhanced Query Performance. *2023 International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), Distributed Computing and Electrical Circuits and Electronics (ICDCECE), 2023 International Conference on*, pages 1 – 5, 2023.

EXAMENSARBETE Optimising Query Execution:
Minimising Re-Computations through Structural Analysis and Partial Updates
STUDENTER Sebastian Malmström, Katia Svennarp
HANDLEDARE Christoph Reichenbach (LTH)
EXAMINATOR Niklas Fors (LTH)

Minska Omberäkningar med Smart Strukturanalys och Smidiga Uppdateringar

POPULÄRVETENSKAPLIG SAMMANFATTNING **Sebastian Malmström, Katia Svennarp**

Volymen av insamlad data ökar dagligen vilket ställer krav på effektivare metoder att förvara och behandla datan. Detta examensarbete undersöker hur logik och analys kan användas för att förutsäga tidsåtgången för en fråga till databasen samt undersöker möjligheterna att minska antalet beräkningar som krävs vid tillägg i databasen.

Större mängd data innebär fler och längre beräkningar för att leverera data till slutanvändaren. Därför har det blivit än mer viktigt att undvika onödiga beräkningar. Detta examensarbete undersöker en arkitektur som kallas vyunderhåll (view maintenance), som innebär att resultaten av vanliga frågeställningar som hämtar information i databasen förberäknas och lagras i en separat datastruktur. Arbetet analyserar om man utifrån frågeställningens struktur kan förutse hur lång tid frågan kommer ta att besvaras innan är utförd. Arbetet utforskar även om underhållet av en vy kan effektiviseras genom att endast räkna om en del av frågeställningarna, specifikt de delarna som påverkats av uppdatering. Två metoder för att uppdatera vyn testas, den första som bara tar i beaktning vad som uppdaterats och kör frågeställningar utifrån det och den andra som även tar hänsyn till analysen av frågeställningarna för att se om det går att optimera ytterligare.

Vi utvärderade dessa två metoder genom tre olika experiment. De första två experimenten fokuserade på statisk analys av frågeställningarna. Målet var att identifiera lämpliga operationer för att klassificera frågeställningarna som komplexa eller inte samt applicera detta i en verklig produkt. Det sista experimentet behandlade frågan om partiella uppdateringar av en vy i samband med uppdatering av databasen.

De viktigaste resultaten är att det är möjligt att förutsäga beräkningskomplexitet i en förfrågan, men detta är starkt beroende av databasspecifika detaljer. Det visade även att implementeringen av logik för att avgöra vilka frågeställningar som ska köras i stället för att köra alla kan spara betydande tid och beräkningar.

Framtida forskning kan utveckla resultaten genom att generalisera dem. I dagsläget är det väldigt specifika operationer och metoder som har behandlats. Även tester på flera olika databastyper samt databasimplementeringar skulle ge resultaten mer bredd.