MASTER'S THESIS 2022

Categorization of Cypher Queries to Improve Benchmark Coverage for Graph Databases

David Johansson, Jonathan Paul

ISSN 1650-2884
LU-CS-EX: 2022-29

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-29

Categorization of Cypher Queries to Improve Benchmark Coverage for Graph Databases

Kategorisering av Cypher queries för att förbättra täckningsgraden av benchmarks för grafdatabaser

David Johansson, Jonathan Paul

Categorization of Cypher Queries to Improve Benchmark Coverage for Graph Databases

David Johansson
nat15djo@student.lu.se

Jonathan Paul
jur15jp1@student.lu.se

June 9, 2022

Master's thesis work carried out at Neo4j Sweden AB.

Supervisors: Simon Priisalu, simon.priisalu@neo4j.com
Jens Wollert Ehlers, jens.wollert-ehlers@neo4j.com
Per Runeson, per.runeson@cs.lth.se

Examiner: Björn Regnell, bjorn.regnell@cs.lth.se

Abstract

Benchmarks are often used to find regressions to avoid performance dropping over time. To make benchmarks relevant for a product, the benchmarks should mirror the users' needs and uses of functionality. To achieve this, user data can be used as a foundation when creating new benchmarks and thus improving the coverage.

This thesis was carried out at *Neo4j* who develops the most frequently used graph database. Using data from their database as a service (*AuraDB*), we focused on finding a way to improve the coverage of the benchmark suite run by them. Using the Design Science Paradigm we formulated problem constructs through interviews with developers at Neo4j. From these problem constructs, a solution was designed, and multiple technological rules were created. Through a validation process using interviews, we identified the validity of our thesis.

The solution was designed by identifying weaknesses in the workloads by categorizing Cypher queries using the *Abstract Syntax Tree* (AST) generated by the Cypher parser. The AST was then used to compare queries run by users of the AuraDB platform with the queries run by the benchmarking suite. The categorization helps finding use cases of the end-user that are currently not covered by the benchmarks. We identified categories for accurate categorization of Cypher queries based on the Cypher syntax and applied this to both the AuraDB logs and the benchmarks.

Using the categorization created from the AST, we initially identified the coverage of the benchmark suite to be 45.70% of the total number of user queries run on the AuraDB platform. With the tool developed in the thesis, new benchmarks were created to increase the coverage to 60.18%, with more benchmarks to be developed.

From our validation we saw that the tool developed in this thesis met the requirements based on the technological rules. This was validated through interviews with developers with insight in how the tool will be used. Due to the lack of database statistics in the AuraDB logs, our solution design does not solve all problem constructs.

Keywords: Benchmark Coverage, Cypher, Categorization, Time-Behavior Performance, Neo4j

Acknowledgements

We would like to thank our supervisors Per Runeson, Simon Priisalu, and Jens Wollert Ehlers for their guidance, ideas, and valuable feedback, and our examiner Björn Regnell. We also would like to thank the employees at Neo4j for their time and effort in helping us with this thesis, joining us for interviews, and helping to validate our solution.

Contents

Chapter 1

Introduction

Increased dependence on technology and software puts higher pressure on the performance and reliability of products and services. To control the quality of the products being produced, testing is used to find bugs and performance issues before they reach the user. With the introduction of agile software development and continuous integration and deployment (CI/CD), the time it takes from writing the software until the user has a version of it available is shorter. This increases the demand for reliable tests that can find performance regression and bugs before they become a known problem. To combat performance regression, *benchmarking* can be performed to find where the regression is occurring.

Benchmarking has been done for many years to validate that a product can uphold certain requirements and standards [1]. When benchmarking was applied to software, nothing changed but the medium that it was conducted on. The art of benchmarking is important to ensure quality during the product's lifecycle and, in software development, when new features are added. The benchmarks are usually created by the developers from either user feedback that there might be a regression in the performance or from the developers' domain knowledge. The optimal scenario is when the user does not notice any problem and is caught before a new version is released. Relying only on the developers' user and domain knowledge before a release can be costly.

This thesis was carried out at *Neo4j Inc.* (in this paper referred to as Neo4j) in close cooperation with their Benchmarking team. Neo4j is a Swedish global company that develops and maintains a database built on graph technology. For this thesis, Neo4j wanted a way to compare their benchmarks with the user data they collect through *Neo4j AuraDB*, a *database as a service* (DaaS) version of the Neo4j database. The aim was to find flaws and limitations in their benchmark suite and, to some extent, help them find where they can improve their benchmark coverage. Currently, they are developing their benchmarks based primarily on the domain knowledge possessed by their engineers. Going forward, they would like to increase the inclusion of user data when determining which benchmarks they run.

1.1 Case Company

Neo4j Inc. is a Swedish global company founded in 2007 with its headquarters in San Mateo, California. They have developed the Neo4j graph database which, according to DB-engine, is the most popular graph database on the market [2] and is used by companies ranging from Fortune 500 to start-ups. Neo4j currently employs around 700 people in their four offices located in Sweden, Germany, the United States, and the United Kingdom, and remotely across 41 locations around the world.

Other than AuraDB, Neo4j has multiple products which are based on their graph database. *Neo4j Graph Data Science* is a platform that is intended to help data scientists with its machine learning tools. *Neo4j Bloom* is a visualization and exploration tool for the database. It presents the graph in an easily digestible manner so that the user can see which data are present in the graph. On all of these platforms, *Cypher* is used as the query language of choice. Cypher is a declarative query language that is developed by Neo4j for their graph database and is the foundation of the new standard in development, *graph query language* (GQL).

1.2 Research Question

The focus of this thesis is to analyze user-produced Cypher queries to identify ways to increase the relevance of benchmarks regarding time-behavior performance at Neo4j. With this goal in mind, we identified the following three research questions.

RQ1 How can Cypher queries be used to assess the relevance of benchmarks?

RQ2 How can Cypher queries be used to improve the coverage of benchmarks?

RQ3 How can the correlation between Cypher queries and benchmarks be visualized?

1.3 Current Research

Current research on benchmarking is extensive and has been developed over many decades, while benchmarking for graph databases is a field that has been on the rise during the last decade. There exists research on how to benchmark graph databases, and for this thesis, we want to improve the coverage using logged Cypher queries. This approach has not been explored and thus needs more research.

Improving benchmark coverage has been done before e.g.[3], but not related to graph databases with the usage of Cypher queries. We found previous research that analyzes and creates a coverage criterion for SQL queries, which we want to relate to Cypher queries [4]. Another research paper describes the process of estimating the execution time for Cypher queries [5].

The related work is further presented in more detail in Section 2.6.

1.4 Outline

In this thesis, we begin by discussing the relevant research in Chapter 2 that introduces related works for this thesis, the concepts of benchmarking, the Cypher query language, and graph databases. In Chapter 3 we introduce the design science paradigm and the research methodologies this thesis is built around. In Chapter 4 we present a deconstruction of the problems to be answered, and in Chapter 5 we present seven technological rules that summarize our solutions to the problem constructs. The validation of our findings is presented in Chapter 6. We wrap up the thesis in Chapter 7 where we discuss our findings before we close with our conclusions in Chapter 8.

Chapter 2
Background and Related Work

In this chapter, we introduce the concepts used in this thesis. The chapter starts with the introduction of what a *graph database* is and the current landscape of the field in Section 2.1. In Section 2.2 we introduce the Cypher query language and in Section 2.3 we introduce quality requirements and how performance is defined. In Sections 2.4 and 2.5 we introduce the concept of benchmarking and how it is currently being applied to graph databases. Finally, we introduce the related work in Section 2.6.

2.1 Graph Databases

When people and data are increasingly connected, classic relational databases do not meet the requirements for handling modern data. From this need, the "Not only SQL" (*NoSQL*) movement arose and new ways of storing data began to be developed. *Graphs*, *Key-value* stores, and *Document* stores are three of the technologies used for these new databases. One way to meet the new demand for more connected data was to use graph databases [6]. The use of graphs to build databases has become more popular during the last decades. These are built using graphs instead of tables. A multitude of Graph Databases have been created, *Neo4j* [7], *Amazon Neptun* [8], and *JanusGraph* [9] to name a few.

The mathematical definition of a graph is a set of edges and vertices [10]. In the Neo4j database, the vertices are instead called nodes, which are objects with properties. The connections between these nodes are called relationships, instead of edges as they are called in graph theory. Relationships connect the nodes and contain information on the relation between the nodes. A graph database works with CRUD (Create, Read, Update, Delete) operations, which in turn uses a graph as the underlying data structure [11].

For a graph database to be optimized, the underlying storage must use *native graph storage* which according to Robinson et al. [11] is when the storage "is optimized and designed for storing and managing graphs". Not all graph databases use native graph storage, but will instead convert the data in the graph to a relational database [11].

To write and retrieve data from the database, a database engine is used. If the database engine makes use of *index-free adjacency* it can be said to use *native graph processing*. Index-free adjacency is when each node in the database stores a reference to its adjacent nodes instead of the database storing that information in a separate index. The purpose of native graph processing is to make query performance depend on how much of the graph needs to be searched, not on the total size of the graph [11].

2.2 Cypher

In the ever-growing industry of graph databases, several graph query languages exist. For the Neo4j database, this language is called Cypher and is developed by Neo4j. Cypher is a declarative query language that was developed to be used with their graph database Neo4j. In 2015 it was released as an open-source project and is now maintained by the *openCypher Implementers Group*. The principle of the language is to use pattern matching in the graph to retrieve data. This is done using certain keywords, or *clauses*, such as `MATCH` or `CREATE`, and "*ASCII art*" for their pattern matching [12].

ASCII art is a technique to graphically design pictures with the use of ASCII characters. In Cypher this is illustrated with the node being represented by parenthesis, the relation an arrow drawn between the nodes, and the information of the relationship in brackets. An example of this can be seen below.

```
(n1:Node)-[:RELATIONSHIP]->(n2:Node)
```

A Cypher query can be divided into two categories, write and read. Other than clauses, the query can contain functions (ex. `count`, `collect`, `date`) and operators (ex. `AND`, `*`, `/`) [13].

A simple read query retrieves all persons called "*Euler*" of the age of 29 and, if the person has any coworkers, it also retrieves them. The query returns the full name of "*Euler*", the number of co-workers if they have any, and sorts it in descending order on this.

```
MATCH (a:Person {age: 29})
WHERE a.name = "Euler"
OPTIONAL MATCH (a)-[:WORKS_WITH]->(p:Person)
RETURN a.name AS name, count(p) AS colleagues
ORDER BY colleagues DESC
```

The other type of query, write, can be created using five different clauses, `CREATE`, `MERGE`, `DELETE`, `SET`, and `REMOVE`. Below, two simple write queries are written, the first creating a new person node with the name "König" and the age 29, the second finding the person "König" with the age 52 and connecting him to "Euler".

Using CREATE:

```
CREATE (p:Person {age: 29, name: "Kőnig"})
```

Using MERGE:

```
MATCH (a:Person {age: 52, name: "Kőnig"})
MERGE (a)-[:GRAPH_THEORY]-(b:Person {age: 29, name: "Euler"})
```

A Cypher query can match a pattern with a relationship chain that is more than one hop away. We can find paths through the graph to find patterns that match the query. To find a chain with two jumps, we can write it as follows, where the second version is in a shorter form [14].

```
(n1)-->()-->(n3)
(n1)-[*2]->(n3)
```

To find longer chains, we can also use *variable-length pattern matching*. Variable-length pattern matching can take either no bounds, or upper and lower bounds to find the chain of nodes matching the pattern. Without any bounds, we can find the longest possible chain, and with bounds, we can limit how long the chain can be. We can also find the shortest matched path. In the following example, we can first see a variable-length pattern matching with no bounds and in the second with both upper and lower bounds [14].

```
(n1)-[*]->(n2)
(n1)-[*0..2]->(n2)
```

When a Cypher query is executed, it will go through the *Cypher Planner*. The Cypher planner decomposes the query into an executable plan containing operators that is run in the database to find the result the query returns. To create the executable plan, the query is first parsed. The clauses, parameters, variables, and labels are all identified and an *Abstract Syntax Tree* (AST) of the query is created. The AST describes the syntax of the query in a tree structure. An example of the AST of a Cypher query can be found in Appendix C. The AST is used to process the different clauses, parameters, variables, and labels when planning the execution of the query. Neo4j offers a tool for generating an AST from a Cypher query in the Cypher front-end library [15].

The Cypher language supports indexing to help the query planner improve the time-behavior performance of a read query. An index is a separate table that stores the indexed nodes in order to increase the performance of the database. There are drawbacks with using indices, such as decreased performance for write queries and increased memory usage, due to copies of the data being stored and must therefore be written to [14].

The execution plan consists of operators which describe how to scan and search the database for data. These operators are, for example, the *AllNodesScan* that reads all nodes in the database. The planner tries to optimize the execution plan by using indices, constraints, and statistics from the database. This statistical information is maintained by Neo4j in their implementation as defined in the Neo4j Cypher manual 4.4 [14]:

1. The number of nodes having a certain label.
2. The number of relationships by type.
3. Selectivity per index (ratio between nodes in index and total number of nodes).
4. The number of relationships by type, ending with or starting from a node with a specific label.

Using these statistics, the planner can create an efficient execution plan with operators that are optimized for the specific dataset on which the query is run. Which operators are chosen for the execution plan affects the execution time of the query. One of the properties that significantly affects the execution time is whether the operator is eager or lazy. An eager operator accumulates all their rows before sending them to the next operator. Eager operators are, for example, sort and aggregation operators that require the collection of all rows to perform their operation. A lazy operator sends rows as they are being produced, which can speed up execution time as the time waiting for all rows being collected is eliminated [14].

When finding the most optimal execution plan, the planner tries different configurations in which the properties of the plan are compared. Each operator in the plan calculates an estimated number of rows that will be used to execute the query. The estimated rows are then multiplied by a cost constant for the operator. An estimated cost for the entire execution plan is calculated and used to find the best possible execution plan based on the constraints [14].

The execution plan is executed using the Cypher runtime. There are three different runtimes implemented in Neo4j, all of which have different optimizations and performance. The *pipelined runtime* is the newest and does not yet support all types of queries. Pipelined uses algorithms to group the operators from the execution plan in order to create a new ordering of execution. This will optimize performance and memory usage and makes the pipelined runtime superior to the two others. Pipelined runtime is still under development and will therefore not support every type of query. If a query is not supported, either interpreted and slotted runtimes are chosen instead. *Interpreted runtime* chains the operators together in a tree in which the operators are executed from the leafs of the tree to the root. Each operator will feed its result to the next operator. Interpreted runtime does not have as many optimizations as other runtimes but will work for every query. *Slotted runtime* is similar to the interpreted, but has more optimizations for the streaming of records in the iterators and has therefore better performance than the interpreted runtime [14].

2.3 Quality Requirements

When developing a system, setting up requirements and following them in production is important to make sure the system is correct in the eyes of the system stakeholders. There are many different definitions of requirements, but they are all similar. The definition used by Wiegers and Beatty defines requirements as:

> **Definition** Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system [16].

The practice of working with requirements is called requirement engineering. When working on a software system, there are usually two different types of requirements applied to the product, *functional* and *quality* requirements, sometimes called *non-functional* requirements. In the literature, the terms quality and non-functional requirements are divisive and sometimes used synonymously [17], while other times they are not [16]. In this thesis, we use the term quality requirements since we agree that this term is more explanatory.

Functional requirements is often the term used for the requirements that are needed for the system to work as intended. According to *International Requirements Engineering Board* (IREB), a functional requirement is defined as "A requirement concerning a result or behavior that shall be provided by a function of a system" [18]. These types of requirements are specified for the functionality of the system.

While functional requirements specify what the system should do, quality requirements specify how well it should do it. They are equally important since a system that does the intended use-cases bad will not be a good system. It might meet all the functional requirements, but it might not be secure or fast, rendering it unusable. IREB defines quality requirements as "A requirement that pertains to a quality concern that is not covered by functional requirements." [18].

When determining the overall quality of software, there are many aspects to consider and it is not always clear how to define these aspects. Therefore, the International Organization for Standardization (ISO) standardized a model to determine software quality through the ISO / IEC 25010:2011 standard. ISO/IEC 25010:2011 calls the model *Software Product Quality* (SPQ), which can be divided into two categories, internal and external, and eight characteristics [19]. The model can be seen in Figure 2.1.

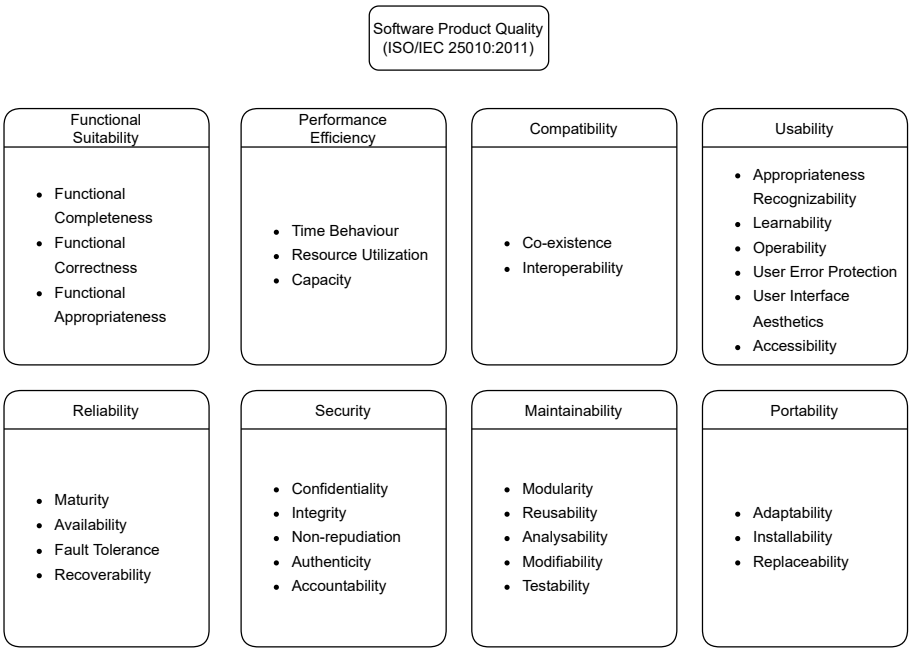


Figure 2.1: The Software Product Quality from the ISO/IEC 25010:2011 standard

In this thesis, *performance* is the main focus. As can be seen in Figure 2.1, the SPQ model breaks the performance down into three sub-characteristics, *Time behavior*, *Resource utilization*, and *Capacity*, which all in themselves are a way of defining performance. The SPQ model defines the sub-characteristics as [19]:

- **Time Behavior** Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet the requirements.

- **Resource Utilization** Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet the requirements.
- **Capacity** Degree to which the maximum limits of a product or system parameter meet requirements.

For this thesis, the main focus is on time behavior performance.

2.4 Benchmarking

In software development, performance is important to the overall success of a system. Performance is a broad term, and, depending on which context and to whom you are talking, it can mean different things. A common way to test the performance of a system is to use benchmarking. The system being compared is often called *System Under Test* (SUT). Benchmarks can be used to test different aspects of the SUT depending on the intended purpose of the benchmark. Traditionally the main focus of benchmarking has been to test the performance of the SUT but *security* and *reliability* have become more common [1]. The *Standard performance evaluation corporation* defines a benchmark as

> **Definition** *A "benchmark" is a test, or set of tests, designed to compare the performance of one computer system against the performance of others* [20].

Benchmarking is an important tool in software development to monitor that a system is running properly. A benchmark consists of a test or set of tests called *workloads* and the workloads are what generate the data that the benchmark uses for comparison. Five key characteristics should be the goal for a benchmark: *relevance*, *reproducibility*, *fairness*, *verifiability*, and *usability* [21]. Each characteristic is important to consider when designing the benchmark; otherwise, the workloads may not meet the intended purpose. Typically, there are two ways to benchmark a system. It can be done by testing each function call, called *microbenchmarks*, or by testing how the user would use the system, called *application benchmarks* or *macro benchmarks*.

When testing the performance of specific and isolated parts of a system the use of microbenchmarks is common. The type of performance that is being tested depends on the system and what the goal of the benchmark is. Microbenchmarks are often fast to run and easy to implement since they tend to be quite small. This method of benchmarking is used to test operations and functions to evaluate performance at lower levels. Since microbenchmarks test each function and evaluate the performance of these individual calls, both bottlenecks and the maximum capacity of the SUT can be found [1].

While microbenchmarks are used to test the performance of individual operations or functions, application benchmarks are used to test how well the system performs when the whole application is run. The optimal way to run these types of benchmarks is on the application that the users themselves run. The problem is that it is expensive and time-consuming to implement, but it gives a more accurate overview of the system. Application benchmarking is a useful tool when testing the overall performance of the system with real or synthetic data [1].

2.5 Database Benchmarking

When benchmarking databases, microbenchmarks and macro benchmarks are commonly used. Workloads are often created for macro benchmarks, where large scenarios and datasets are created to allow the database to be benchmarked. These benchmarks focus on targeting larger parts of the database. Microbenchmarks can be used as a complement to macro benchmarks. The microbenchmarks are more directed towards primitive functions in the database and are therefore better to use for detecting specific issues.

Ciglan, Averbuch, and Hluchy [22] introduced a way of benchmarking graph databases, focusing on traversal operations. Before this study, there were few frameworks for benchmarking graph databases. Ciglan, Averbuch, and Hluchy provide information on how the benchmarks were constructed for traversal operations, which gave the means to create a comparison of different database systems and their performance in terms of time efficiency.

Angles et al. [23] introduced a benchmark for graph databases using a social network workload. *Linked Data Benchmark Council* (LDBC) is a council that develops benchmark workloads for graph databases that combine user data with input from system architects. The paper contains a detailed description of the social network workload defined by LDBC. The benchmark is supposed to measure the performance of graph databases using a workload that can simulate how the database will be used in a social network. Multiple queries are defined that should run on a set of test data to find the performance of the database.

Lissandrini, Brugnara, and Velegrakis [24] build on the LDBC Social Network Benchmark to show that macro benchmarks often fail to show a justified picture of the advantages in different database systems. To test the primitive operators of the queries, they suggest using smaller microbenchmarks directed to these primitive operators instead of larger macro benchmarks covering larger parts. The complex queries that are performed in the macro benchmarks of LDBC cannot pinpoint which properties of a database system are the bottleneck, and the queries can almost always be deconstructed into the primitive operators that are benchmarked in the microbenchmarks. The methodology introduced in the study evolved into an open-source testing suite for graph databases to benchmark their capabilities without complex queries.

2.6 Related Work

Grambow et al. [3] present a method to improve the relevance of microbenchmark suites. The goal of their work was to find a way to quantify the relevance of microbenchmarks, speed up the execution runtime of the microbenchmark suite without affecting their relevance and also try to increase the relevance of the microbenchmarks. To evaluate the method, an experiment was set up with different scenarios for time series database systems. Application benchmarks were created for each database that simulated workflows, and smaller microbenchmarks were created to test specific functions. The method they used to compare the relevance included the use of call graphs. Call graphs were used for both application benchmarks and microbenchmarks, where they could compare the coverage and relevance of the microbenchmarks depending on how much of the call graph was covered. They could also use the application benchmarks to find which microbenchmarks covered the calls which were made in the call graph and find if any were redundant or not covered.

Zhenzhen, Jiong, and Binglei [5] propose an approach for estimating the execution time of a Cypher query. The existing methods for estimating the execution time of SQL queries were not applicable for graph databases. They wanted to implement a method that uses machine learning and neural networks to predict execution time. The prediction was made using queries and database statistics. From the queries, they extracted their execution plan, which was combined with the database statistics for the prediction. When training their model, they used three real-world datasets, which provided them with enough data to find correlations of execution times. They showed that their model predicted the execution times with a relative error of 16%.

Tuya, Suárez-Cabal, and Riva [4] developed a coverage criterion for SQL queries to find the test data coverage of a query. They based this criterion on *Full Predicate Coverage* and *Masking Modified Condition Decision Coverage* (masking MCDC). Masking MCDC defines a coverage criterion that all possible outcomes have been covered at least once and that each condition has been the individual determining factor for the outcome. Full Predicate Coverage is an extension of masking MCDC for predicates, where every predicate is covered, and there is only one single determining factor that determines the outcome. The criterion formed in their study included a wide range of different SQL semantics. For more common programming languages, a coverage analysis of the code usually involves a control-flow approach, where path and branch coverage plays a large role. In a database application, this can be difficult, as the program interacts with the DBMS in a single interaction where the query is sent to the database. Therefore, a control-flow approach is not easy to perform for the query. This study created coverage rules and criteria for SQL queries where test data in the database could be instantiated for full coverage of the query. Common SQL query semantics were analyzed to identify what the requirements were to have a fully covered test database. A tool was also developed to automate the generation and evaluation of the coverage rules for a SQL query.

2.7 Summary

In the background we have covered different areas which will help us during this thesis. We presented how the query language Cypher is structured, and how different aspects of the planner and runtime in Cypher affect the performance of a query. Quality requirements and performance were defined and this thesis focuses on time behavior performance. We continued with explaining how a benchmark can be used to ensure a system fulfills the quality requirements defined for it. There are multiple ways to perform benchmarking, microbenchmarks and macro benchmarks are often used. A macro benchmark is useful when creating a user flow which are benchmarked, which relates to this thesis where we want to use user data to improve the benchmark suite.

Chapter 3
Research Methodology

The methodology in this thesis is based on a research paradigm called *design science*, adapted for software engineering by Runeson et al. [25]. Design science focuses on human-made constructs, where a deeper understanding regarding the design of those is to be researched. This is often used in engineering sciences and medical sciences [26].

The method used in the thesis contains four steps that can be iterated multiple times.

1. *Problem Conceptualization*
2. *Solution Design*
3. *Implementation*
4. *Empirical Validation*

In the first step, we did a problem conceptualization to gain an understanding of what needed to be done and created problem constructs. When the problem constructs were created, they were used in the solution design to propose possible solutions for the problem. The design was then implemented to find a solution to the problem which went through an empirical validation to assess whether the problem constructs were solved. If not, a new solution design will be required, or the problem constructs were not correctly understood, and a new problem conceptualization will be required [25].

In Figure 3.1, the steps followed during the thesis are illustrated as arrows. The paradigm is divided into two theoretical and two practical domains. The practical domains consist of the problem that needs to be conceptualized and the solution to the problem. Theoretical domains are the conceptualized problem constructs together with the solution designs for these and can also be described in terms of *technological rules* [25].

A technological rule can be described as the connection between the problem domain and the solution domain. It can be expressed as [25]:

> *To achieve <Effect> in <Context> apply <Intervention>.*

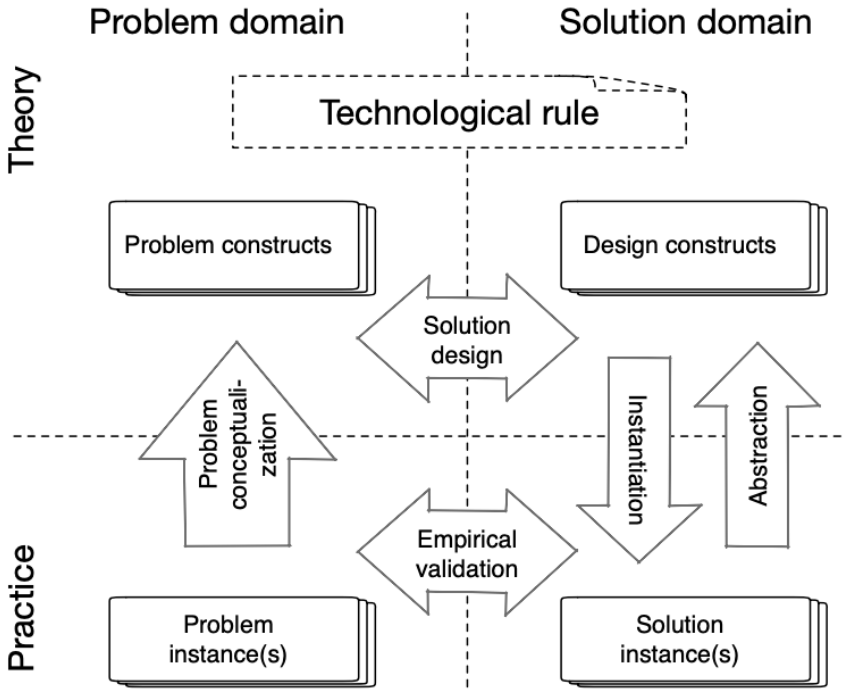


Figure 3.1: The steps for the design science paradigm [25]

where the effect of the technological rule is what we want to achieve that will solve the problem. Context refers to where the effect should appear and intervention is the design construct that will solve the problem [25].

Problem conceptualization and solution design are closely related in the theoretical domains. They are combined in a technological rule, which gives a generalized and short summary of the problem related to the solution. A problem and a solution construct can be described using multiple technological rules that can have different abstraction levels. Even if technological rules can have different abstraction levels, they should not be created at a very high or very low abstraction level. If it is too high the rule will lack substance and fail to describe the implementation needed. A too low abstraction level will cause the rule to have an insignificant scope and be irrelevant to implement [25].

Technological rules are a central part of the theoretical domains in design science and are vital to be constructed. A rule represents the direction that the implementation should take and contains information on both the problem and the solution. The rules will also help during the validation of the implementation in the practical domains [25]. Therefore, the technological rules were a central part of creating the theoretical background of the thesis. Both problem conceptualization and solution design helped to create multiple technological rules that were a baseline for the implementation and created a path forward.

3.1 Problem Conceptualization

Problem conceptualization is the first step we did when using the design science methodology. A problem has been identified in the practical domain and needs to be conceptualized

Table 3.1: Keywords used in the literature search for background.

Keywords	
Benchmarking	Macro benchmarks
Graph Databases	Neo4j
Benchmarking database queries	Testing database queries
Benchmarking graph databases	Cypher query language
SQL categorization	Database query categorization
ISO/IEC 25010:2011	Time-based performance

Table 3.2: Keywords used based on the interviews.

Keywords	
Benchmark Coverage	Test Coverage SQL
Execution time database queries	Execution time SQL queries
SQL categorization	Database query categorization
Abstract Syntax Tree	Syntax categorization

into a theoretical domain. This process is done by identifying elements in the problem domain called problem constructs. A theoretical background can be found which is related to the problem constructs. This theoretical background gives the material needed to find a solution design that can solve the practical problem [25].

3.1.1 Literature Search

A literature search was conducted to find related work that created a theoretical foundation for both the problem conceptualization and the solution design. The search was done following a task list defined by Thiel [27]. To find the literature, keywords were used to search across both *LubSearch* and *Google Scholar*. LubSearch has a large collection of academic databases available through Lund University, which was our main source when searching for literature. Google Scholar was used as a complement to find additional literature that was not available in LubSearch. The correctness of the publications was ensured by primarily searching for peer-reviewed publications. We did an initial search to collect background for the interview questions. The keywords used were related to the initial research questions that were constructed. After the interviews, additional literature searches were required using new keywords based on the interviews. Together with keywords found in the related literature, we collected literature for the background, problem conceptualization, and solution design. Keywords used to search for publications related to the background are listed in Table 3.1 and keywords used based on the interviews can be found in Table 3.2.

The literature found with keywords was determined by publication date, citation num-

Table 3.3: Participants in the interviews.

ID	Participant Role	Interview Session
SeSE A	Senior Software Engineer	Interview 1
SeSE B	Senior Software Engineer	Interview 2
SeSE C	Senior Software Engineer	Interview 2
SeSE D	Senior Software Engineer	Interview 3
SeSE E	Senior Software Engineer	Interview 3
StSE A	Staff Software Engineer	Interview 3
DirE	Director of Engineering	Interview 4

bers, and title. After this, we reviewed the abstracts of the relevant literature and if it still looked relevant the complete report was reviewed and analyzed to find the theoretical usage of the literature in the problem conceptualization and solution design. As the solution design evolved, new literature was needed. Therefore, the process was iterative and new literature was found when additional background was required.

3.1.2 Interviews

To conceptualize the problem in this thesis, interviews were conducted with engineers from Neo4j. Participants and their roles are listed in Table 3.3. There are multiple ways to conduct an interview, for example, unstructured, semi-structured, and fully structured [28]. A semi-structured interview is based on a set of questions planned beforehand. The questions can be asked in any order and a question might already have been answered by another question. These questions are open-ended and allow for the interviewee to freely talk [29]. The interviews in this thesis were semi-structured interviews to allow for deeper answers that helped during the problem conceptualization. The interviews also use a funnel model, where the questions start more open and during the interview will be more specific [28]. This is done using both predefined questions and follow-up questions. A funnel model helps get an overview that can be specialized into concrete problems for the conceptualization so that smaller problems can be formulated. The interviews were recorded and transcribed to allow us to go back and examine the details. During transcription, new details can also be found, making this step valuable for the conceptualization [28].

A consent form was also constructed for the interviewees. This form has information about privacy and how the interview will be used in the thesis. There is also information on how long the interview will be saved and how the interviewee can withdraw from the study [29]. The interview guide with the predefined questions used in the interviews can be found in Appendix A and the consent form can be found in Appendix B.

From the transcribed interviews, we started to analyze them. This was done using a process of *coding* and *grouping*. Coding is the process of identifying keywords in transcribed interviews. These keywords can be single words or whole paragraphs, as long as the context of the statement is captured. This is done to find the important key themes in the interviews, which are used later when grouping. When grouping, the keywords found from the different interviews are put in groups to capture the main essence of the keywords and how they are related to each other. The coding and grouping are then used to find a conclusion [30]. The conclusions are used when applying the knowledge gained from the interviews to the research

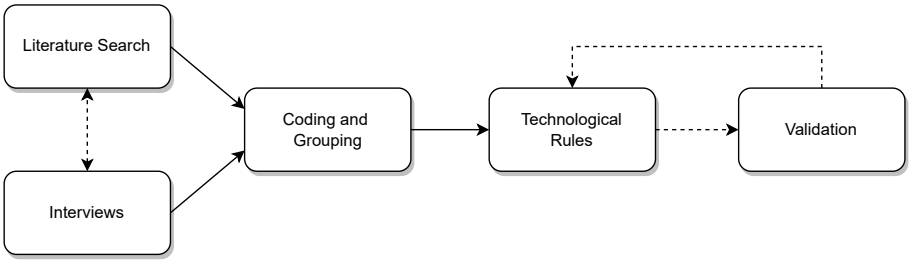


Figure 3.2: The work process for producing Technological Rules

questions that are defined in Chapter 1. The research questions were broken down into smaller problems that were used to create technological rules. In accordance to [25], the creation of smaller problems was done in multiple steps, starting with a higher-level problem. The process was repeated until implementation-specific problems were found, defining what technical challenges that might arise.

3.2 Solution Design

The solution design is closely related to the problem conceptualization and aims to find an implementation solution to the problems found in the conceptualization [25]. The interviews conducted for the problem conceptualization gave an understanding of the problem and a foundation for possible solution designs. Together with a literature search to establish a theoretical foundation, solutions were designed for the problem constructs. Technological rules were created with the design constructs together with the problem construct that it should solve. The process of creating technological rules is represented in Figure 3.2.

With the technological rules as a foundation, a prototype for the categorization and analysis of Cypher queries was created. This prototype was used to solve the problem constructs to give the user an insight into the current state of the benchmarks in relation to the user queries.

3.3 Validation

When the solution was implemented, it was validated to ensure that the solution met the quality requirements defined in the solution design. The work process was iterative, and during continuous validation with developers at Neo4j, the technological rules together with their implementation were revised and iterated.

Due to the investigative character of the thesis, it was determined that interviews were most effective to ensure the validity of the solution [30]. In Table 3.4 we present the participants in the validation interviews. The first set of validation interviews was conducted with software developers who managed the benchmarking suite, as their knowledge of the benchmarks helped determine whether the solution created an improvement for them. As with the problem conceptualization, semi-structured interviews were used for the interviewees to freely answer and elaborate on the asked questions, as explained in Section 3.1.2. To validate the categorization of Cypher queries, software developers from the Cypher teams were interviewed. The questions asked during the validation can be found in Appendix A.

Table 3.4: Participants in the validation interviews.

ID	Participant Role	Interview Session
SeSE A	Senior Software Engineer	Interview 1
DirE	Director of Engineering	Interview 1
SE A	Software Engineer	Interview 2
SE B	Software Engineer	Interview 2
SeSE F	Senior Software Engineer	Interview 2
SeSE G	Senior Software Engineer	Interview 2
StSE B	Staff Software Engineer	Interview 2
SeSE B	Senior Software Engineer	Interview 3
SeSE D	Senior Software Engineer	Interview 3
SeSE E	Senior Software Engineer	Interview 3
StSE C	Staff Software Engineer	Interview 3

The validation interviews were analyzed using a methodology similar to that used for the first interviews, where we coded and identified groups of keywords. Using the analysis of the interviews and the keywords found, we validated the technological rules which were defined in the solution design. With the validation process, we iterated and improved on the implementation.

Chapter 4
Problem Conceptualization

For the problem conceptualization, we conducted a literature search, with the method explained in Section 3.1.1, to find a relevant background for the construction of interview questions. The results of the literature search can be found in Section 4.1 and more detailed in Chapter 2. The interviews were analyzed which resulted in an additional literature search. Details of the interviews and the analysis are explained in Section 4.2. The interview analysis led to the creation of five subquestions to our initial research questions. These can be boiled down to

- How can the semantics of a Cypher query be separated into categories?
- How can the execution of a Cypher query be used in query categorization?
- How can we create a relevant categorization?
- How do we compare benchmarks with query logs?
- How can the results be represented?

4.1 Literature Search

The first step before conducting interviews were a literature search focused on finding related work and background for the thesis. The detailed results from this literature search can be found in Chapter 2. We found related work that had some elements of similarities, but not completely answering the problem we wanted to solve. A related article used call graphs to improve benchmark coverage [3], and we wanted to see if it was possible to relate it to Cypher queries. Another publication predicted the execution time of queries using the query plan and database statistics [5], which was something we wanted to examine in the interviews.

For the background literature, we wanted to gain an understanding of how different benchmarks are performed and what kind of performance is measured. We found literature that describes the different types of benchmarks, such as application benchmarks and microbenchmarks [1][21]. For performance, we used the ISO/IEC 25010:2011 standard as a starting point and searched for literature surrounding this standard. Cypher was also a subject for the literature search, as we wanted to have some knowledge surrounding the Cypher query language before moving into the interviews. This background was used as a foundation, together with the related work, for the interview questions. We formed questions to find what kind of benchmarks are being run at Neo4j, what kind of performance they are measuring from the benchmarks, how the Cypher language is structured, and which properties of the Cypher query are interesting for the categorization.

4.2 Interviews

We conducted three interviews with developers from three different teams and one interview with a manager, as shown in Table 3.3. The questions we asked were divided into two parts, one for the benchmarks, how they are created and used, and one for Cypher, where we asked about the properties they thought were interesting for categorization. When we analyzed the interviews, we found several keywords in the interviews that explained the multiple properties of queries that we used when starting with the solution design. Keywords were grouped into larger groups that identified the different nuances of the interviews and were used with the solution design. The groups with their containing keywords can be found in Table 4.1.

From the interviews, we saw that benchmarks are created in similar ways across the teams. Microbenchmarks, which focus on specific functions in the implementation, are created when new features are introduced or specific areas need to be benchmarked. Macro benchmarks are most often queries that are run on a database to simulate a real-life workload. These workloads are created based on datasets that can be used in real applications and are used to create benchmarks that emulate how the customers are using the product. The main function of the benchmarks is to measure the performance of the database over time. Currently, the main focus is on time behavior measurements, but other forms of benchmarks are also used, for example, memory measurements. Benchmarks are used to find regressions and verify that new features and implementations do not create a regression of the database performance. They are also used to find specific issues that can arise with a certain machine configuration, and, in that way, find issues before they are released to the public.

When asked about the different characteristics of a Cypher query and which parts were important for the categorization we got similar answers from the different teams, with a larger focus on the areas the team worked on themselves. We divided the Cypher characteristics into two parts. The syntax of the query and the execution of the query. When looking at the syntax of the query, the first characteristic that we should look at was read and write queries, which everyone agreed on. Read and write queries behave differently inside the database, which gives large differences in performance. Clauses were also an important part of the categorization, where we received examples of clauses we should look at, such as `MERGE`, `MATCH`, or `CALL`. Aggregations were another characteristic that was highlighted in all interviews as they change the behavior of the result return. Aggregations such as count and collect have to compute all rows before returning, which makes them slower than if not

Table 4.1: Keywords from the interviews divided into the four created groups.

Benchmark Usage
Time Behavior
Memory
Regression Hunting
Feature Testing

Benchmark Creation
Microbenchmarks
Macro Benchmarks
Workloads
Field Datasets

Syntax of Queries
Read and Write
Clauses
Aggregations
Eagerness and Laziness
Functions
Periodic Commit

Execution of Queries
Runtime
Indices and Constraints
Database Statistics
Shape and Size of Graph
Condensed Nodes
Execution Plan

aggregating the result. This was also mentioned under eagerness and laziness, where the different operators can be either eager or lazy, and was a big factor to look at. Other keywords we looked at were if the query contained any function or used periodic commit. Functions can be both internal as implemented in Cypher, and external by using extension libraries like APOC, which is a very popular library for Neo4j. Some functions affect performance more than others, and these functions were identified as interesting to look at. Periodic commits is a functionality where a write query can be written to the database in intervals, for example for the loading of large CSV files, which can affect the performance.

When asking about the execution of the queries, there were many factors that were mentioned as important. The execution plan itself can have optimizations depending on which operators are included in the execution plan, such as if certain patterns have optimizations. The runtime used would also be interesting for categorization, as the different runtimes have different properties and handle the query differently. The structure of the database itself plays an important role in the performance of the query. Indices or constraints in the database can make the queries go faster if correctly configured. The shape and size of the graph can also affect the query. One example which was brought up was that if there existed a condensed node with many relationships connected to it, the query might be slower as it needs to iterate over them. Database statistics are useful for categorization as this enables us to find the execution plan of the query, which would make it possible to categorize the operators used in the query as well.

Chapter 5
Solution Design

In this chapter, we present the solution design to the problem constructs discussed in Chapter 4. We first present the technological rules in Section 5.1 and describe them in detail. We then continue with how the queries were parsed and analyzed in Section 5.2 and 5.3 before presenting the result and visualization of the comparison in Section 5.4 and 5.4.2.

5.1 Technological Rules

To generate the solution design we used the interviews conducted in the problem conceptualization. From Table 4.1 we see the keywords and groups identified in the interview analysis. Using these keywords, we constructed our technological rules. The technological rules and the relation to the interview analysis are described in more detail in this section. In Figure 5.1 we can see the relation between keywords and technological rules. We can note that the group *Execution of Queries* is not connected, which is due to the lack of information from query logs and was therefore not used. Validation interviews were also used as a complement to the interviews done in problem conceptualization, where they helped iterate on several technological rules.

TR1 To achieve higher coverage of benchmarks, compare the benchmarks with user data.

TR2 To achieve a comparison between benchmarks and user data, categorize the queries.

- **TR2.1** To achieve an accurate categorization of the semantics of a Cypher query, generate and use an AST.
- **TR2.2** To achieve a relevant categorization, start by categorizing with larger categories first and then break the query down into more and more specific characteristics.
- **TR2.3** To achieve an accurate comparison between categorized queries and benchmarks, compare their categories and find similarities.

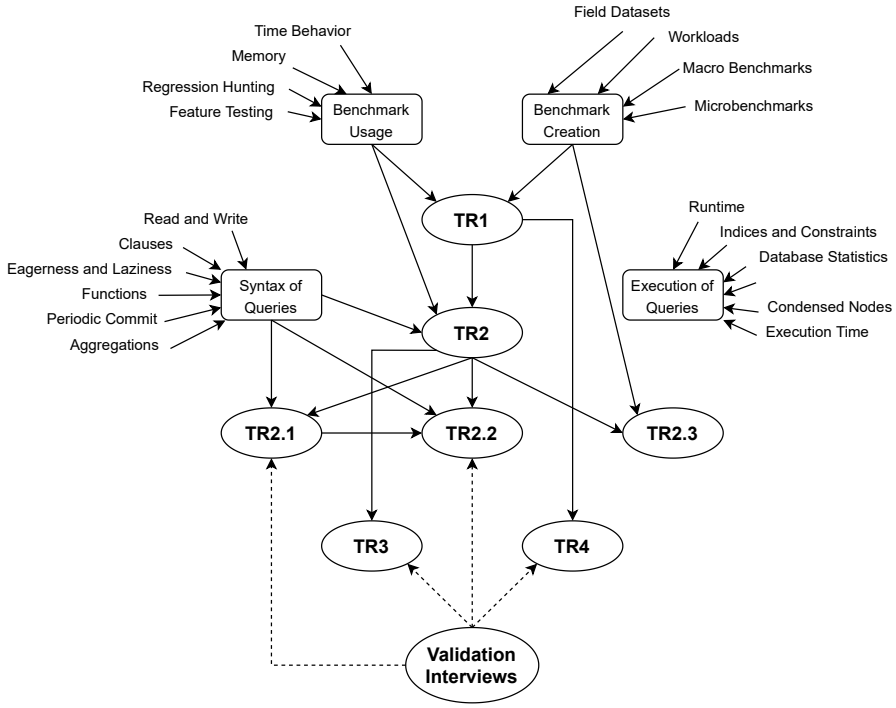


Figure 5.1: The relation between keywords and technological rules.

TR3 To achieve visualization of the correlation between categorized user data and benchmarks in Neo4j apply a heat map of the categories and highlight the categories with some related benchmarks.

TR4 To achieve the ability to automate the comparison of categorized user data and benchmarks in Neo4j create scripts that analyze and write data to a CSV.

TR1

The first technological rule is a generic rule which states what we wanted to accomplish. Based on the interviews, we found that the creation of benchmarks is most often not created with user data. We found that we can increase the coverage of the benchmarks using user data as a foundation when creating new benchmarks. Therefore, the technological rule constructed states that, to achieve a higher coverage of benchmarks compare the benchmarks with user data. Higher coverage of benchmarks increases the efficiency of regression hunting and feature testing, and with user data, this testing will be more relevant for the users.

TR2

To achieve higher coverage of the benchmarks, we constructed a more specific technological rule. Looking at how Neo4j are using the benchmarks we wanted to group the user data into categories that describe the queries. Together with the syntax of queries, we found that a query could be grouped into categories. The idea was to achieve a comparison between benchmarks and user data by using these categories that the queries from users are placed into. After this, we took each category to see if the category has benchmarks that cover that

specific category. This gave us an indication of the benchmark coverage and where more focus can be put.

TR2.1

When comparing two Cypher queries, we first analyzed the semantic to find the similarities between them. From the information given by the query logs, the most viable way to categorize a query was to produce the AST for the query and then deconstruct it into categorizable data.

One of the main problems identified in the interviews was that the query logs that were produced by the AuraDB platform do not contain information regarding the dataset, and thus some information that can improve the time-behavior performance might not be included. An example of this is if the database instance uses an optimally implemented index to speed up searches. This will decrease the evaluation time of a match query, but will not be visible to the logs. A fast execution time for a query might be because it uses indices, but it might also be a smaller dataset. The same applies to the number of nodes and relationships between said nodes.

TR2.2

When categorizing the queries we started by identifying the largest categories. This was a natural process when parsing from the generated AST as it is represented in a tree structure. The further down you go in the tree, the more specific the categories are.

The difficulty in making the categories more and more specific is knowing when to stop. We cannot have categories for every single query, as this would render the categories useless. The queries can not be too large either, as this would make the queries indistinguishable. Therefore, a balance was required.

TR2.3

We wanted to find a comparison between the query logs and the benchmarks. Categorization was performed on both the query logs and the benchmarks so that the categories could be compared directly between each other. As we only had access to the query for the benchmarks, semantic analysis was the main contributing factor for the categories.

TR3

To use the results from the data collected and analyzed in TR1 and TR2, a representative overview was required. Thus, visualization was needed to show the essential correlations intuitively. The data were visualized using a heat map to represent the correlation between the benchmarks and the user queries. In addition to the heat map, other data such as the execution time and the content of the queries were displayed for engineers to gain insight into the content needed to generate a benchmark to fulfill this category. The design of the UI was iterated through selected developers at Neo4j to represent the data in a simple yet clear way. With the vast amount of data collected for categorization, the problem became how to generate a UI that would present vital information without clogging the UI.

TR4

To be able to use the comparison and check the improvement overtime to make sure the benchmarks continue to keep a high coverage, we wanted to implement the possibility to automate the process. Therefore, scripts were created and the analyzed data was saved in a CSV file that could be compared with other analyses of other data. This technological rule is a continuation on TR1 and has its origin from the validation interviews where the developers saw a need for the task to be automated.

5.2 Query Parsing

Both the query logs and the benchmarks were required to be parsed before being able to use them for the thesis. The query logs were collected from user data using the AuraDB platform. The benchmarks were collected from Neo4j's internal benchmark suite.

5.2.1 Query Logs

Large amounts of query logs were supplied to the thesis work by Neo4j. The logs were collected over 24 hours on a normal weekday. More than 1 TB of AuraDB query logs were used, which is equivalent to 136 million queries. These logs were collected from AuraDB's Professional Tier which is one of three tiers in AuraDB. The professional tier is most often used for medium-scale applications, which gives a representation of queries used in production. Before being used, these query logs needed to be obfuscated and processed to extract relevant information. The logs are structured as JSON files with multiple data points that could be discarded. The relevant information that was collected from the query logs was as followed:

```
{
    "allocatedBytes": ...,
    "elapsedTimeMs": ...,
    "event": ...,
    "neo4jversion": ...,
    "pageFaults": ...,
    "pageHits": ...,
    "runtime": ...,
    "query": ...
}
```

Furthermore, the logs contained data that could be filtered out because they were not queries or did not contain relevant information for the comparison. We only wanted log entries where the event parameter was *success*. This filtered around half of the queries, as both a start and a successful event were logged for the same query. We also filtered out queries that had a runtime with the value *system* as these queries are mainly used to manage database users and internal database management and are therefore not relevant to the comparison.

As the query logs are gathered from real data, sensitive information had to be obfuscated before being used in the thesis. When the information was extracted from the logs, we first removed the information related to the database which the information came from. The

query itself had to be anonymized as variable names, labels, properties, and other names can be used for identifying sensitive data. This anonymization was done using a built-in Cypher tool where variable names, labels, and properties are anonymized and given generic names.

```
MATCH (n:Person)-[r:ACTED_IN]->(m:Movie)
WHERE n.name = "Euler"
RETURN n.age, m.title
```

The query above has sensitive information and will after the anonymization be represented as the following.

```
MATCH (var0:L0)-[var1:R0]->(var2:L1)
WHERE var0.P0 = "string[5]"
RETURN var0.P1, var2.P2
```

From the anonymized query, we can still find the syntactic properties of the query so that we can analyze and categorize the query. The specific names of the variables are not relevant in the categorization, which means that we can use the generic names instead.

5.2.2 Benchmarks

The benchmarks that were considered in this thesis were Cypher queries that are run on different datasets. To parse the benchmarks, each file containing Cypher queries was read and represented in JSON files with the benchmark name and the query that was run.

```
{
    "benchmark": ...,
    "query": ...
}
```

As the benchmark queries were collected from the implementation, the same log data was not available. Therefore only the query itself was represented.

5.3 AST Parsing

To categorize queries and their syntax, we generated an AST. Using the tool found in the Cypher implementation, we created the AST and parsed it to generate categories for the queries. As shown in the example AST in Appendix C, we had access to the query in a nested structure. We implemented a program for parsing this nested structure, where we started by determining the larger and more generalized categories. The first categories into which we could divide the queries were read queries and write queries. We could continue down the nested structure to find additional categories, for example, if it was a single query or a union between queries. The created categories can be found in Table 5.1. Clause categories include multiple categories created for different clause combinations.

The main part of the categorization was done with the clauses of the query. The clauses gave a lot of information about the query and we created categories for different combinations of clauses. The process was done by using information from the interviews and how

different Cypher queries are structured in the Cypher Refcard [13]. For example, one of the most common queries is a single `MATCH` with a `RETURN` such as the following.

```
MATCH (n:Person)
RETURN n.name
```

We created a clause category for every query containing one `MATCH` clause and one `RETURN` clause called matchReturn. We also wanted to create clause categories for queries with multiple `MATCH` clauses and one `RETURN` such as the following query.

```
MATCH (n1:Employee)
MATCH (n2:Customer)
MATCH (n3:Supplier)
RETURN n1.name, n2.name, n3.name
```

The clause category created for these kinds of queries was called multiMatchReturn. We wanted to create categories for the more common types of clause combinations. In addition to the refcard and interviews, we had access to a preliminary set of test data to find our first clause categories. When a category was too specific and not common to run, we created generic types of categories that were considered if the query did not match any of the other categories we created. These generic categories matched queries that had one or many of one or more specific clauses. An example is the genericMerge category, which matches all queries that contain a `MERGE` clause. These generic categories also had a weight property to them which described which generic category to match when multiple could fit for the query. Categories with a lower weight property were prioritized.

`UNWIND` and `FOREACH` are clauses that run operations on lists. These got separate categories telling us that the query contains the clause. They are therefore not considered in the clause categories. We also created categories for queries containing aggregations, order by, shortest path, hints, and variable-length pattern matching.

A query was assigned multiple of these categories we created and the combination of the categories created the category of the query. The full category could be for example [readOperation, singleQuery, matchReturn].

5.4 Comparison with Benchmarks

When comparing the benchmarks with the query logs, the common denominator between them was the query. Therefore, the comparison was made using the query syntax. AST parsing was done for both the benchmarks and the queries in the query logs. We could then compare the categories the queries were put into. We marked a category as covered by benchmarks if there existed one or more benchmarks with that category.

From the query logs, we had access to additional information that the benchmarks did not have. The additional information we extracted was allocated bytes, elapsed time in milliseconds, event, Neo4j version, page faults, page hits, and runtime. The relevance of the information was determined from the initial interviews. As the information was not available for the benchmarks we presented this data outside of the categorization.

Elapsed time in milliseconds for each query was split into larger groups for each category. A category could have queries that took between 0-1 ms, 1-5 ms, 5-20 ms, 20-100 ms, and

Table 5.1: Categories generated from AST parsing.

Category	Description
Read Operation	A query that only reads information from the database.
Write Operation	A query that has some form of write operation on the database.
Single Query	A single query without any unionized queries.
Union All	A query with multiple unionized single queries, including duplicated rows.
Union Distinct	A query with multiple unionized single queries, excluding duplicated rows.
Sub Query Call	A query using a subquery with the `CALL` clause.
Contains Unwind	A query that contains the clause `UNWIND`.
Contains Foreach	A query that contains the clause `FOREACH`.
Contains Aggregation	A query that contains an aggregation such as `collect` or `count`.
Contains Order By	A query that contains `ORDER BY` on return items.
Contains Shortest Path	A query finding the shortest path for a pattern.
Var Expand	A query with a Variable-length pattern matching. Different categories for if the Variable Expand has bounds, no bounds, or only a lower or upper bound.
Using Hints	A query using an index or join hints.
Clause Categories	Clause categories include multiple categories for the different clauses tailored to clause patterns.
Unknown Query	A query where it is not a Single Query, Union All, or Union Distinct
Unknown Clauses	A query where no clause category could be found for the specific clause combination.

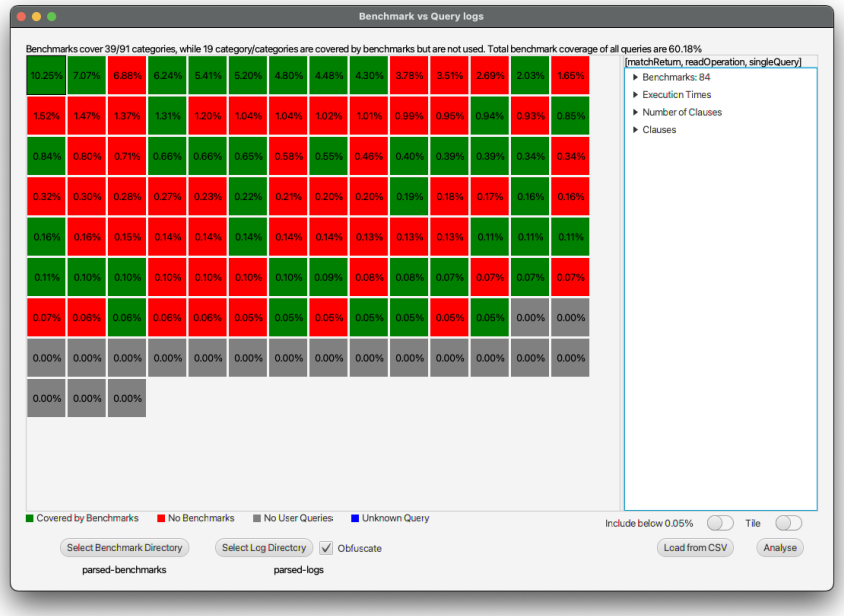


Figure 5.2: Tiled version of the visualization.

more than 100 ms. The number of queries in each group was calculated and visualized for every category.

The Neo4j version and the runtime used were counted for each query log to see the distribution of the version and runtime. We found that all queries were run on the latest version of the AuraDB platform, that more than 90% of all queries were using the pipelined runtime, and more than 97% of the queries that did not use pipelined were set explicitly by the user. These variables were not correlated to the categories and were only logged when running the analysis.

Allocated bytes, page faults, and page hits were not used in the final analysis. Their relevance was determined to not be enough to be used in the final iteration of the analysis.

5.4.1 Visualization of the Comparison

When visualizing the comparison, we created a JavaFX user interface. Categories were represented as an overview using a grid or as a list. Categories covered by benchmarks were marked in green, not covered categories were represented in red, categories covered by benchmarks with no user queries were gray and non-categorized queries were marked with blue. We could also use the user interface to load folders with benchmarks and query logs to parse, obfuscate, and analyze. The user interface with a grid view and list view can be found in Figure 5.2 and Figure 5.3 respectively.

Each category has a percentage connected, which represents the number of user queries that exist in that specific category compared to the total number of queries. Categories with less than 0.05% of the total amount of queries were filtered out by default but could be viewed using a toggle. To get more understanding of the category the user interface had an additional panel where extra information about the category is shown. In this panel, all extra information is printed out to analyze each category more in-depth. The information that is

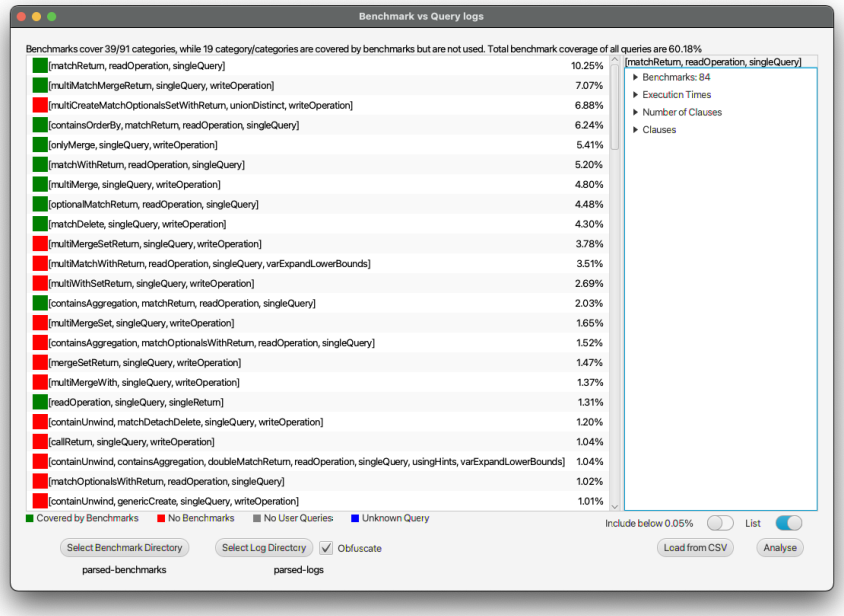


Figure 5.3: List version of the visualization.

represented depends on which category is selected, as the different categories only had some of the information available. All extra information that could be generated for a category is shown in Table 5.2.

In addition to the visualization done in the user interface, we created an output in a CSV file. The analyzed data would be written to a CSV with some additional logging done in a log file. This CSV file could later be loaded into the user interface to obtain the representation in the program. We also created scripts that could be run on the command line. One for parsing and one for analyzing, but also a script combining them both.

5.4.2 Results of the Comparison

The categories were created to show the nuances of each query, while not being too specific. The percentages represented for each category as the percentage of the total user queries in a category were used to find the most common queries. If a category was too broad, the category could be split into smaller categories, giving a more distinct categorization.

For the data we had collected, we generated a total of 513 categories, of which 91 were above the threshold of 0.05% of all user queries. The benchmarks were split up into 104 categories, where 19 of the categories the benchmarks covered did not have any user queries in them. When looking at the categories which were above the threshold, benchmarks covered 35 of 91 categories. When including categories under the threshold, the benchmarks cover 85 of 513 categories. This is presented in Table 5.3. The coverage of the benchmark suite is 45.70% of the total number of user queries run on the AuraDB platform. With the tool developed in this thesis, new benchmarks were created to increase the benchmark coverage to 60.18%.

The ten most frequent categories contained 58.41% of all user queries and can be found in Table 5.4. Of these top ten categories, the benchmarks covered five of these. During the

Table 5.2: Additional information is represented for each category.

Information	Description
Benchmarks	Contains the number of benchmarks covering the category and the name of each benchmark.
Execution Times	The distribution of execution time groups.
Number of Clauses	The distribution of the number of clauses.
Clauses	All clause combinations found in the category.
Example Queries	A single example query from the user queries for the specific clause combination.
Variable Expand	The structure of the variable-length pattern matching, showing if and which bounds were used.
Hints	Shows which hints and how many of each were used in the queries.
Aggregations	Shows which aggregations and how many of each were used in the queries.
Union Clauses	Shows the clauses from each subquery in a union, which is later combined into the clause combination.
Order By	Shows how many items the queries ordered on.
Procedure Names	Shows which procedures and how many of each were used in the queries.

Table 5.3: Results of the Comparison

	Excluding categories under 0.05%	Including categories under 0.05%
Covered by Benchmarks	35	85
Total number of categories	91	513

Table 5.4: Top ten categories with respect to the number of user queries.

Category	% of User Queries
matchReturn, readOperation, singleQuery	10.25%
multiMatchMergeReturn, singleQuery, writeOperation	7.07%
multiCreateMatchOptionalsSetWithReturn, unionDistinct, writeOperation	6.88%
containsOrderBy, matchReturn, readOperation, singleQuery	6.24%
onlyMerge, singleQuery, writeOperation	5.41%
matchWithReturn, readOperation, singleQuery	5.20%
multiMerge, singleQuery, writeOperation	4.80%
optionalMatchReturn, readOperation, singleQuery	4.48%
matchDelete, singleQuery, writeOperation	4.30%
multiMergeSetReturn, singleQuery, writeOperation	3.78%

thesis three additional categories from the ten most frequent categories were covered which resulted in the increase of benchmark coverage.

Chapter 6
Validation

In this chapter we present the results from the validation process and how we validated the technological rules constructed in the solution design from Chapter 5. We start with presenting the result of the validation interviews in Section 6.1 and continue with discussing what they meant for the technological rules in Section 6.2.

6.1 Interviews

When we had created a working version of the query comparison with visualization, we interviewed developers and managers who had insight into the applications of the comparison. The interviews gave us multiple points of improvement and valuable feedback on features and categories that could be added and removed. We implemented features that were useful when improving the benchmarks, such as the specific details of each category and user example queries. New categories were created after the validation interviews, such as a category for variable-length pattern matching. We also identified nested clause structures that had not been covered, such as the difference between procedure calls and subquery calls. In Table 6.1 we list the keywords extracted from the validation interviews.

From the interviews, we mostly received feedback on which parts were missing, what the developers would like to see implemented, or if they deemed any categories unnecessary. Generally, the developers were positive towards the tool and felt like it gave valuable feedback in the creation of benchmarks. We implemented most of the features requested from the validation interviews. Some of the requests were not possible to implement as we were missing the required information to solve these. Database statistics were mentioned throughout the interviews as a next step. With statistics, the execution plan can be generated which opens for more points of categorization. Unfortunately, this was not available to us in the logs and thus was not available to be used.

With the validation interviews, we identified categories where we had missed some features. The `CALL` clause is one example of this, where the same clause can be used both for

Table 6.1: Keywords from the validation interviews with suggested improvements.

UI Improvements
List View of Categories
Search Bar to Filter
Command-line Scripts
CSV Output

Query Syntax
Hints and Where
Procedure Call vs Subquery Call
Var Expand with Bounds
Shortest Path

Query Information
Manual Tuning of Queries
The Runtime Used

Query Execution
Indices
Generate Plan
Transactions

subqueries and for procedure calls. These differences were identified, and categorization was added to support this. We also found that there existed a need for automating the analysis. This enables continuous analysis to assess the coverage of the benchmarks without the need to manually perform the task. This led to the implementation of command-line scripts that can be run from a build script and output the result of the analysis to a CSV file.

The developers also pointed to additional information that was of interest to them, such as how the user patterns looked like. We had some information available to us, such as the runtime used and if the user had done some manual tuning of the query. We choose to present this in a sidebar in the UI. Additional analysis of the user patterns was outside the scope of this thesis and was therefore not implemented.

6.2 Technological Rules

With the validation interviews, we wanted to get an idea of whether the technological rules had solved the problem constructs. With the added features and categories from the feedback received in the interviews, we analyzed the technological rules and validated them. The technological rules are introduced in Section 5.1.

TR1

The first technological rule is a generic rule. For this technological rule, we looked at the result of the comparison. We increased the coverage of benchmarks from 45.70% of the user queries to 60.18% using the categories created. The coverage can continue to increase as the work continues with implementing more benchmarks based on the categories identified by

the tool developed in this thesis. If the tool is used continuously, changes in user patterns can be identified and benchmarks can be added to meet the new patterns.

TR2

For the second technological rule, we wanted to achieve a comparison between benchmarks and user data using categories. This technological rule was divided into three specific technological rules. **TR2.1** states the use of the AST to gain an accurate categorization. We generated and used the AST in the thesis. From the validation interviews, we saw that the categorization was representative of the queries. We also saw that there still existed information in the AST which are not categorized in this thesis. This leads to **TR2.2** where we wanted to achieve a relevant categorization by starting at a higher level in the AST and then breaking it down to more specific categories. We did not want to have categories that were neither too specific nor too generic. This led to a restriction on the information from the AST we used. We determined that, for example, a `WHERE` statement was not relevant for the categorization at this stage, as this is a statement that most queries include and do not affect performance to the same extent as other statements.

For **TR2.3** we wanted to compare the categories from the query logs and benchmarks. This was done by analyzing both the query logs and benchmarks, using only the query, as this was the common denominator between them. With this approach, benchmarks and queries from the logs were compared. As the same categories were used for both the benchmarks and queries, they were compared to each other by placing them in the same categories and identifying if the categories contained benchmarks. Where benchmarks existed, we considered that category to be covered. We believe the comparison is accurate as we are comparing the same parameters in both queries. The developers can also use the number of benchmarks in the category to see if there are too few or too many benchmarks in a category.

TR3

For the comparison visualization, we started by creating a tiled view in which a quick overview of how many categories are covered. During validation, we received feedback that it was hard to see where specific categories were. This led to us creating a list view where a larger focus was on the name of the category instead of if it was covered or not. This made it easier to find a specific category. The combination of these two layouts and the possibility to easily switch between them greatly increased the usability for the developers when they needed to identify which categories needed improvements.

TR4

TR4 was a technological rule which was created from the validation. The developers saw a possible need to automate the analysis so that the tool can be run multiple times. We created scripts and wrote the analyzed data to a CSV file for automatic running and reading of the results. Over time, the result can be further analyzed to see how the coverage varies.

Chapter 7
Discussion

In this thesis, we have developed a tool to compare user data from the AuraDB query logs, with the benchmarks run by Neo4j. This was done with the intent of identifying limitations in the benchmarking suites based on the usage of real data. We have also presented seven technological rules that can be used to develop this type of tool in regards to comparing Cypher queries. In this chapter, we first discuss the methods used and the challenges that have occurred during the designing of the tool. Furthermore, we discuss our result and the validity of it. We also discuss the limitations we had to set and the possibility for further work that can be done based on this thesis.

7.1 Methodology

The method for this thesis was based on design science. We wanted to analyze the possibility to compare user data with the benchmarks run at Neo4j to find if it was possible to access their relevance. Due to us wanting to investigate the problem stated by Neo4j and design a solution for the problem, we decided that the design science paradigm would be good to follow when constructing the methodology. As stated in Chapter 3, we split the methodology into four parts: problem conceptualization, solution design, implementation, and finally validation.

The first activity of problem conceptualization was important for us in order to grasp the problem that Neo4j perceived. To construct the conceptualization of the problem, we chose to use interviews together with a literature study. We used the interviews to create an understanding of what different parts of the organization saw as the main problem in categorizing Cypher queries and comparing them with the current benchmarking suite. The categories we identified were primarily based on these interviews and the literature study. We leaned heavily on the interviews because of the domain knowledge of the engineers, and we used the literature study to complement their knowledge.

For the interview questions, we used the research questions as a basis so that the questions

did not fall outside the scope of the thesis. We chose to construct the interview questions using open-ended questions to extract information specific to the benchmarking suite run at Neo4j and to get more depth into Cypher.

We conducted two group interviews with two and three people, respectively, and two individual interviews, all with people with different areas of expertise related to the Neo4j graph database and Cypher. All the developers we interviewed have an insight into how the Cypher query language, planner, and runtime are constructed, and thus gave us more information on how the queries could be broken down. All of them also had information on how current benchmarks are constructed and the process behind their choice of them. This gave us knowledge that might not have been easy nor possible to find in our literature study.

Designing the solution was the process in which we took the problem constructs from the conceptualization, found a possible solution for them, and then constructed the technological rules. We then took these technological rules and deconstruct them into more specific rules, which described the steps to follow. For example, we wanted to find a way to increase the coverage of benchmarks. This was done by using a comparison of user queries and benchmarks to find what was used in real situations. We took the technological rule that was created for this case and found a solution for how to compare them. Our choice was to use categories for the comparison. The technological rule was deconstructed into specific ways of categorizing the queries themselves. Through this iterative approach, we took the problem constructs and concretized them into specific sub-problems. Then solutions to these problems were implemented and validated.

The four parts of the methodology were designed to be iterative. Each part could be revisited to be elaborated on when parts were found to be missing. Multiple iterations were performed for the problem conceptualization, solution design, and implementation, which allowed us to find more accurate solutions and more correct problem constructs. This iterative approach ensured that each step was completed before we continued with the process. Since we knew that if there were issues with our problem construct or solution, we traced back to the previous step. We constructed the problem conceptualization and started working on the solution design and the implementation. Later, we discover that we needed to refine the problem constructs. We then refined the problem and continued the process. Validation was an important step in the iterative methodology. When validating, we wanted to find if the solution design were valid for the problem constructs we had created. From validating the solution we found both flaws and improvements for the solution.

7.2 Results

In this section we discuss the different aspects of the thesis. In Section 7.2.1 we discuss the related work and what it contributed to in our thesis. In Section 7.2.2 we discuss how the data collection was done, and how it affected the thesis. Section 7.2.3 discusses how the security of the data was handled. In Section 7.2.4 we cover the reasoning behind the categorization of the queries, and in Section 7.2.5 we cover the comparison and its results. Section 7.2.6 brings up the limitations we had when working on the thesis.

7.2.1 Related Work

The method of improving the coverage of benchmarks, as presented by Grambow et al. [3], involved using call graphs of their functions. When we constructed our solution design, we did not have access to the specific execution operators that were used in each query. Therefore, we were required to base the categorization on something other than the call graph of the operators.

The prediction of execution times, as presented by Zhenzhen, Jiong, and Binglei [5], gave us a great indication of the variations of Cypher queries. In their work, they used both the query plan and database statistics, which data we did not have access to and, therefore, it was not possible for us to predict the execution times of queries in the way they did.

In the work by Tuya, Suárez-Cabal, and Riva [4] of developing a coverage criterion for SQL queries, we can see that it can be hard to use a control-flow approach in our thesis. Their criterion was based on the semantics of SQL to find which structure the database should have. In our thesis, we use the semantics of Cypher to find the nuances of each query to find what makes a query unique and, in turn, try to find a way to fully cover all Cypher queries with benchmarks.

7.2.2 Data Collection

The thesis aimed to make it possible to increase the coverage of benchmarks by using user data. When using user data to construct benchmarks, we can make sure that we are benchmarking functionality that is used by database users.

In AuraDB, logs were collected from their Professional tier as a way to improve the database. The Professional tier is mostly used for medium-scale applications that are in production. This gave us an indication of what queries would run in a user application as opposed to the AuraDB Free tier, which is suitable for training purposes. The Enterprise edition is used for large-scale applications that are not logged in the same way. These applications generate too much data to be viable for this thesis to handle. Therefore, the Professional tier was the best fit for query logs to compare with the benchmarks.

The query logs we used were collected during 24 hours on a weekday from instances all over the world. By doing this, we collected data from an average day, which gave us a fair representation of which queries are usually run and not put more weight into specific areas of the world. The choice of doing 24 hours was a deliberation between completeness and management of resources. The optimal number of days is hard to define. The number of users varies between hours and by different regions, but might be fairly consistent depending on how the database is being used. Therefore, we believe that it was mandatory to use full days and due to resource limitations, we limited the data used to 24 hours. When we remove unnecessary data and obfuscate queries, we could reduce the size of the query logs by 95.9%. This sped up the analysis because we did not have to iterate over the entire amount of data, which was another advantage of only extracting the relevant information. Collecting the queries and parsing them occupies a large amount of memory and it was not possible for us to use more data with the current setup. This is a clear limitation, as more data might indicate that some categories are used more frequently than they are in our dataset, even if we believe that 136 million queries should give a good indication of the current usage.

7.2.3 Security

Security is a major concern when user data is used. Therefore, multiple steps were taken to ensure the anonymity of the queries. Only the query itself was interesting to us when comparing them with the benchmarks. This led us to strip all identifying information from the logs, as that information would not be relevant for categorization. We also made sure to obfuscate the query itself so that all sensitive information was anonymous. We only wanted to look at the syntax of the query, which in itself does not reveal anything about the person who made the query, but might still reveal sensitive information if not handled correctly.

One disadvantage of using completely anonymous queries is that we cannot compare queries made with the same data in one single database. This comparison would have been interesting to do as we might have found some connections regarding the performance of queries when comparing them from the same database.

7.2.4 Categorization

To have a more accurate picture of the database where the query has been run, the database statistics would have given an indication of the structure of the database. Using the database statistics, the execution plan of the query can be constructed, which would have allowed us to compare the operators in a more precise way. This could lead to a categorization where it would be viable to compare the runtime operators, which would give a more accurate picture of which kind of data the query is running on and give a possibility to compare the performance of the queries even more. Database statistics is not something that is logged by default, so it was not possible for us to use it in our categorization.

From the query logs, we received the information shown in Section 5.2.1. We got the anonymized query, but also some additional information such as the elapsed time in milliseconds, the allocated bytes for the query, and which runtime the query used. From the benchmarks, we only had access to the query itself. As the common denominator between query logs and benchmarks is the query, this became the first step of comparing and categorizing. Elapsed time, allocated bytes, page faults, page hits, and runtime were all candidates for the categorization, but since we did not have any database statistics, no conclusions surrounding these variables could be drawn. A simple match query where we only look for a small certain pattern can differ a lot in execution time based on how the structure of the graph is. The node we are matching might have a large number of relationships, which slows down the query compared to a node with only one relationship. Therefore, we cannot only use the query to determine the execution time of a query, which in turn makes it hard to compare a query to a benchmark as we only have the query for that specific benchmark. We kept the execution time as a metric for each category for the developers to see the distribution of execution times for the categories. This helps them gain an understanding of how long a query of that category generally takes and construct databases for benchmark workloads where the execution times can be considered.

Runtime was another variable that we wanted to examine. From multiple interviews in the problem conceptualization, we got answers mentioning the importance of runtime when looking at the time-behavior performance of a query. The pipelined runtime is the newest and does not yet have support for all queries. The user can change the runtime the query will use, which affects the performance. When we looked at the runtime, we saw that more than

90% of all queries were using the pipelined runtime, and more than 97% of the queries that did not use the pipelined runtime were set explicitly by the user. This made the runtime metric not as relevant for categorization as we might have thought, as the majority were pipelined and would not give a significant difference in the categories. Therefore, we only used the runtime as a statistic metric for the developers who were interested in the distribution, and not in the categorization of the queries.

When we constructed the solution design for the categorization of queries, we started to look at the AST of the query. As we wanted to find the differences in the syntax of the query, the AST had the properties we needed. There exists a function for generating the AST in the Cypher code base. Using the AST we iterated further down in the tree structure to categorize each part of the query one after one. The usage of the AST made the implementation easier, as we did not have to implement each syntax keyword in our program. We also got some additional functionality, such as a function to see if the query was a write or read query. We also knew that the function was correct since the developers are working closely with Cypher on a day-to-day basis.

7.2.5 Comparison

From the comparison, we found that 45.70% of all queries had benchmarks in their category, as presented in Section 5.4.2. The number of categories covered is 85 out of 513, or 16.57%. This shows us that the more frequently used categories are more often covered, while less used categories are not covered. What we can see is that queries in the less frequently used categories are often larger and more complex. For example, the third largest category was dominated by a single type of query. This query was a longer and more complex query, which we assume comes from a single database that runs continuously each day. It looked too specific and complex to be a general query that is run by multiple customers on different databases. Thus, this category did not have any benchmarks. For a larger and more complex query like this, it can take a longer time to implement a new benchmark for it.

Similar categories existed where the categories were very specific. Most of these categories had few queries and were not high on the priority list to cover with benchmarks. In an optimal setting, every category should be benchmarked. Since resources are often limited, the priority will most often be on the most frequently used categories.

With the tool that we have developed, we identified different syntactic patterns that users are writing in their queries. With the information retrieved from this tool, new workloads can be created to benchmark specific patterns to find out if the time-behavior performance is regressing with new releases. Many of the categories we found can easily be implemented as new workloads. Three of the ten categories most frequently used were covered by extending existing benchmarks with new workloads. This led to an increase in benchmark coverage from 45.70% to 60.18% equal to 14.48 percentage points, or 31.69%.

7.2.6 Limitations

For this thesis, we did not use allocated bytes, page faults, or page hits. These were extracted from the query logs in case they became relevant for use. As the majority of the categorization was done using the query and looking at the time-behavior performance of them, we determined that the extra extracted information was not required in the categorization.

One disadvantage of using the already implemented query anonymizer and AST generator was that they did not support the APOC extension library. APOC has a vast number of extension functions that can be called from a query. We saw multiple queries that used APOC as a way to create subqueries. These subqueries would be converted to an anonymized string statement when anonymizing and would be shown as a simple function call in the AST generator. We opted not to make the distinction between APOC and regular function calls, due to not getting all information from the APOC query and the fact that APOC currently is not a focus for benchmarking. There exists an ambition to include APOC in benchmarks in the future, and when that happens, a new iteration would have to be done to include the nuances of different APOC procedures.

7.3 Further Work

In the query logs collected by Neo4j, information and statistics were not collected about the database. Therefore, information such as indices, the number of nodes, and the relationships between nodes could not be used as parameters when comparing benchmarks with query logs. If the data that was available contained this information a deeper comparison in regards to the time-behavior performance could be conducted. The paper written by Zhenzhen, Jiong, and Binglei [5] could be used as an extension if the data were available, as their paper uses the generated execution plan and statistics to estimate the execution time of queries. We could use this predicted execution time as an additional categorization property, which could make the categorization more centered on time-behavior performance.

The data collected in this thesis were taken from a single day during 24 hours. To improve on our thesis, this data collection could be done more consistently during longer periods to make sure the data truly represents the usage. The tool might be extended into an automated process that evaluates the relevance of benchmarks over time, and how the coverage varies over a longer period. Neo4j is under constant development, where new features are continuously introduced. With a continued measurement of the coverage, variances in the usage patterns can be adapted to and benchmarks can be created to cover new features that are implemented.

A tool like the one developed by Suárez-Cabal et al. [31] where they use a coverage criterion for SQL queries to find what data would be needed in a database for every criterion being covered could be extended for this thesis. A tool that uses the analyzed data and automates the creation of workloads and benchmarks could increase coverage as new categories and new queries are added automatically.

This thesis focused on the syntax of queries to form a categorization and comparison with benchmarks. Another way of performing a categorization of queries is to look at the user patterns of the users. Instead of only looking at how the queries are structured, we can combine it with how the users are using their database. With this information, new workloads can be constructed which are representative of how the user normally runs queries in the database instead of only looking at a single instance of a query.

Chapter 8
Conclusions

Our goal for this thesis was to analyze Cypher queries produced by users to assess the relevance of benchmarks to improve their coverage. Throughout this thesis, we have discussed how to categorize Cypher queries and have implemented a tool to compare user queries with the benchmark suite run at Neo4j based on the three research questions presented in Section 1.2. In this section, we present the conclusion we have drawn.

For **RQ1** and **RQ2** we concluded that using the syntax and generating an AST from the query was the most viable way of assessing the relevance of the benchmarks due to the lack of database information in the logs. With the AST, we categorized the syntax and compared queries to find missing workloads. From this conclusion, we then developed a tool that compares the benchmarks with the query logs, which can be used to find flaws in the benchmarks and to improve the coverage. We managed to identify multiple new workloads that were missing in the benchmarking, which increased the coverage from 45.70% to 60.18% by adding a few new workloads.

For **RQ3**, we concluded that the use of a heat map can provide a good overview of the current benchmark coverage, providing a functional tool to use when finding new benchmarks to implement. A list representation was also created to give a quick overview of the names of the different categories that were created.

From our validation, we can determine that our solution design partly solves the problems formulated in the problem conceptualization. We believe that the problem is not fully solved due to the lack of information regarding the database statistics in the logs. A continuation of the work with the inclusion of database statistics could give additional results. We conducted interviews with the developers who have insight into how the tool that we have developed will be used, and based on the technological rules, the tool met the requirements.

References

[1] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking: For Scientists and Engineers*. Springer, 2020.

[2] solid IT, "Graph database ranking," https://db-engines.com/en/ranking/graph+dbms, accessed: 2021-04-14.

[3] M. Grambow, C. Laaber, P. Leitner, and D. Bermbach, "Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites," *PeerJ Computer Science*, vol. 7, p. 548, 2021.

[4] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full predicate coverage for testing sql database queries," *Software Testing, Verification and Reliability*, vol. 20, no. 3, pp. 237–288, 2010.

[5] H. Zhenzhen, Y. Jiong, and G. Binglei, "Execution time prediction for cypher queries in the neo4j database using a learning approach," *Symmetry*, vol. 14, no. 1, p. 55, 2022.

[6] M. A. Hassan, "Relational and nosql databases: The appropriate database model choice," in *2021 22nd International Arab Conference on Information Technology (ACIT)*, 2021.

[7] Neo4j, "The leader in graph databases," https://neo4j.com/, accessed: 2022-02-17.

[8] Amazon, "Neptune," https://aws.amazon.com/neptune/, accessed: 2022-02-28.

[9] JanusGraph, "Janusgraph: Distributed, open source, massively scalable graph database," https://janusgraph.org/, accessed: 2022-02-17.

[10] R. Diestel, *Graph Theory*. Springer, Berlin, Heidelberg, 2017.

[11] I. Robinson, J. Webber, and E. Eifrem, *Graph databases.* O'Reilly, 2013.

[12] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *SIGMOD'18 Proceedings of the 2018 International Conference on Management of Data*. Houston, United States: ACM Press, Jun. 2018, p. 1433.

[13] Neo4j, "Neo4j cypher refcard 4.4," https://neo4j.com/docs/cypher-refcard/4.4/, accessed: 2022-02-16.

[14] Neo4j, "Neo4j cypher manual v4.4," https://neo4j.com/docs/cypher-manual/4.4/, accessed: 2022-02-16.

[15] Neo4j, "Cypher front end modules," https://github.com/neo4j/neo4j/tree/4.4/community/cypher/front-end.

[16] K. Wiegers and J. Beatty, *Software requirements, Third Edition.* Microsoft Press, 2013.

[17] S. Lauesen, *Software Requirements: Styles and Techniques.* Addison-Wesley, 2002.

[18] M. Glinz, "A glossary of requirements engineering terminology," International Requirement Engineering Board, 2020.

[19] ISO, "ISO/IEC 25010:2011," https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en, accessed: 2022-02-10.

[20] "Standard performance evaluation corporation," https://www.spec.org/spec/glossary/#benchmark, accessed: 2022-03-04.

[21] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 333–336.

[22] M. Ciglan, A. Averbuch, and L. Hluchy, "Benchmarking traversal operations over graph databases," in *2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012, pp. 186–189.

[23] R. Angles, J. B. Antal, A. Averbuch, P. Boncz, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J. L. L. Pey, N. Martínez *et al.*, "The LDBC social network benchmark," *arXiv preprint arXiv:2001.02299*, 2020.

[24] M. Lissandrini, M. Brugnara, and Y. Velegrakis, "Beyond macrobenchmarks: Microbenchmark-based graph database evaluation," *Proc. VLDB Endow.*, vol. 12, no. 4, p. 390–403, 2018.

[25] P. Runeson, E. Engström, and M.-A. Storey, "The design science paradigm as a frame for empirical software engineering," in *Contemporary Empirical Methods in Software Engineering.* Springer International Publishing, 2020, pp. 127–147.

[26] J. E. van Aken, "Management research based on the paradigm of the design sciences: The quest for field-tested and grounded technological rules." *Journal of Management Studies*, vol. 41, no. 2, pp. 219–246, 2004.

[27] D. V. Thiel, *Research Methods for Engineers.* Cambridge University Press, 2014.

[28] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, pp. 131–164, 2008.

[29] E. Magnusson and J. Marecek, *Doing Interview-based Qualitative Research: A Learner's Guide*. Cambridge University Press, 2015.

[30] M. Höst, B. Regnell, and P. Runeson, *Att genomföra examensarbete*. Studentlitteratur AB, 2006.

[31] M. Suárez-Cabal, C. Riva, J. Tuya, and R. Blanco, "Incremental test data generation for database queries." *Automated Software Engineering*, vol. 24, no. 4, pp. 719 – 755, 2017.

Appendices

Appendix A
Interview Guide

A.1 Interview Guide Developer

1. What is your role at the company?
2. How do you choose when to develop a new benchmark?
3. What is the process behind creating new benchmarks?
4. What types of benchmarks do you create?
5. In what way do the results of the benchmark suite contribute to the development of Neo4j?
6. What type of performance are you measuring with the benchmarks?
7. What kind of properties can a Cypher query be divided into?
8. What parts of a Cypher query are the most relevant as candidates for categorization?
9. Can benchmarks be categorized using the same method as for the query logs?
10. Do you have anything you want to add?

A.2 Interview Guide Manager

1. What is your role at the company?
2. In what way do the results of the benchmark suite contribute to the development of Neo4j?
3. What kind of results would you like to see from a categorization of AuraDB query logs?
4. What benchmark measurements are most important to you?
5. How do you think AuraDB user data could improve the benchmark suite?
6. Do you have anything you want to add?

A.3 Interview Guide Validation

1. What is your role at the company?
2. Would you say the categories represent the differences of Cypher queries?
3. Do you think the categorized queries could be used to improve the coverage of benchmarks?
4. Are there any specific categories you are missing?
5. Is there any additional information you would want from the queries that are interesting according to you?
6. Do you have anything else you want to add?

Appendix B
Consent Form

Consent Form Master Thesis at Neo4j

By signing this document, I understand and agree that

- I will participate in this interview.
- The interview will be recorded and transcribed to be analyzed in the master thesis.
- I can withdraw my participation from the master thesis at any time up until the thesis is published.
- After the master thesis is completed, the interview recording will be deleted if not requested to be kept by the participant.

Signature Participant **Date**

Signature Interviewer **Date**

Signature Interviewer **Date**

Appendix C
Example AST

Cypher query:

```
MATCH (a)-->(b) RETURN a
```

AST representation of the Cypher query above. Irrelevant data have been stripped.

```
Query(
    SingleQuery(
        List(
            Match(
                Pattern(
                    List(
                        EveryPath(
                            RelationshipChain(
                                NodePattern(Some(Variable(a))),
                                RelationshipPattern(OUTGOING),
                                NodePattern(Some(Variable(b)))
                            )
                        )
                    )
                )
            ),
            Return(
                ReturnItems(
                    List(UnaliasedReturnItem(Variable(a), a))
                )
            )
        )
    )
)
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-03

EXAMENSARBETE Categorization of Cypher Queries to Improve Benchmark Coverage for Graph Databases
STUDENTER David Johansson, Jonathan Paul
HANDLEDARE Simon Priisalu (Neo4j), Jens Wollert Ehlers (Neo4j), Per Runeson (LTH)
EXAMINATOR Björn Regnell (LTH)

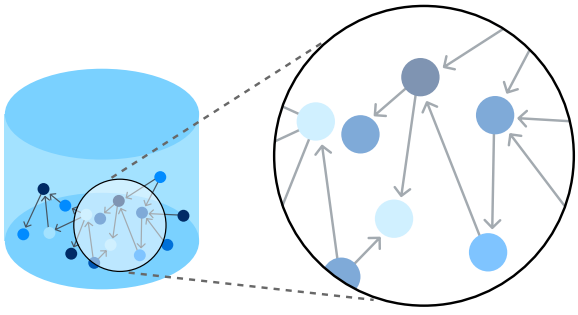
Testa mer effektivt med hjälp av användardata

POPULÄRVETENSKAPLIG SAMMANFATTNING **David Johansson, Jonathan Paul**

Benchmarks är en form av tester som används för att mäta och upptäcka försämringar av kvaliteten i ett system. Dessa kan ofta vara uppbyggda med konstruerad användardata. I det här arbetet har vi jämfört data från riktiga användare med benchmarks för att förbättra testerna för grafdatabasen utvecklad av Neo4j.

För att försäkra sig om att ett system bibehåller en hög kvalitet allt eftersom nya funktioner läggs till är benchmarks ett viktigt verktyg som hjälper utvecklare att se när problem i systemet dyker upp. Genom att upptäcka dessa problem innan de släpps till användare kan man förhindra att kvaliteten påverkas negativt. Användarupplevelsen är central för att bedöma kvaliteten och benchmarks behöver därför ta hänsyn till hur användaren nyttjar systemet. Ett sätt att göra det är genom att använda sig av användardata, som kan ge en bra indikation över vilka användningsområden som är vanliga i systemet.

Vi har i det här arbete jobbat tillsammans med det Malmöbaserade företaget Neo4j som tillhandahåller en grafdatabaslösning. En grafdatabas är uppbyggd av sammankopplade punkter av data, vilket illustreras i figur 1. För att en grafdatabas ska kunna hålla en hög kvalitet behöver den snabbt och effektivt hitta samband och relationer mellan olika punkter, vilket är anledningen till att benchmarks körs frekvent av utvecklarna på Neo4j. De existerande benchmarks som finns är skapade med hjälp av konstruerad användardata som exempelvis kan baseras på erfarenhet och utvecklarens kunskap av systemet. Neo4j samlar även in data från användare som vi i vårt arbete har jämfört med de redan existerande benchmarks som finns för grafdatabasen. Det gjorde vi genom att skapa ett verktyg som visar vilka funktioner av databasen som saknar benchmarks. Med verktyget kan vi öka täckningsgraden så att vi inkluderar användarområden som inte omfattas av de benchmarks som finns idag. På så sätt kan Neo4j öka relevansen av sina benchmarks och se till att de testar den funktionalitet som används av deras kunder. Det här leder i sin tur till att Neo4j kan bibehålla den höga kvaliteten som behövs för en snabb och effektiv databas.



Figur 1: Illustration av en grafdatabas[1].

Genom verktyget som vi utvecklat i arbetet kunde vi öka täckningsgraden av benchmarks med hela 32%, vilket genom fortsatt användning av verktyget kommer att öka ännu mer. Genom arbetet kunde även andra områden identifieras där en mer detaljerad datainsamling hade gett en bättre indikation över hur databasen används.