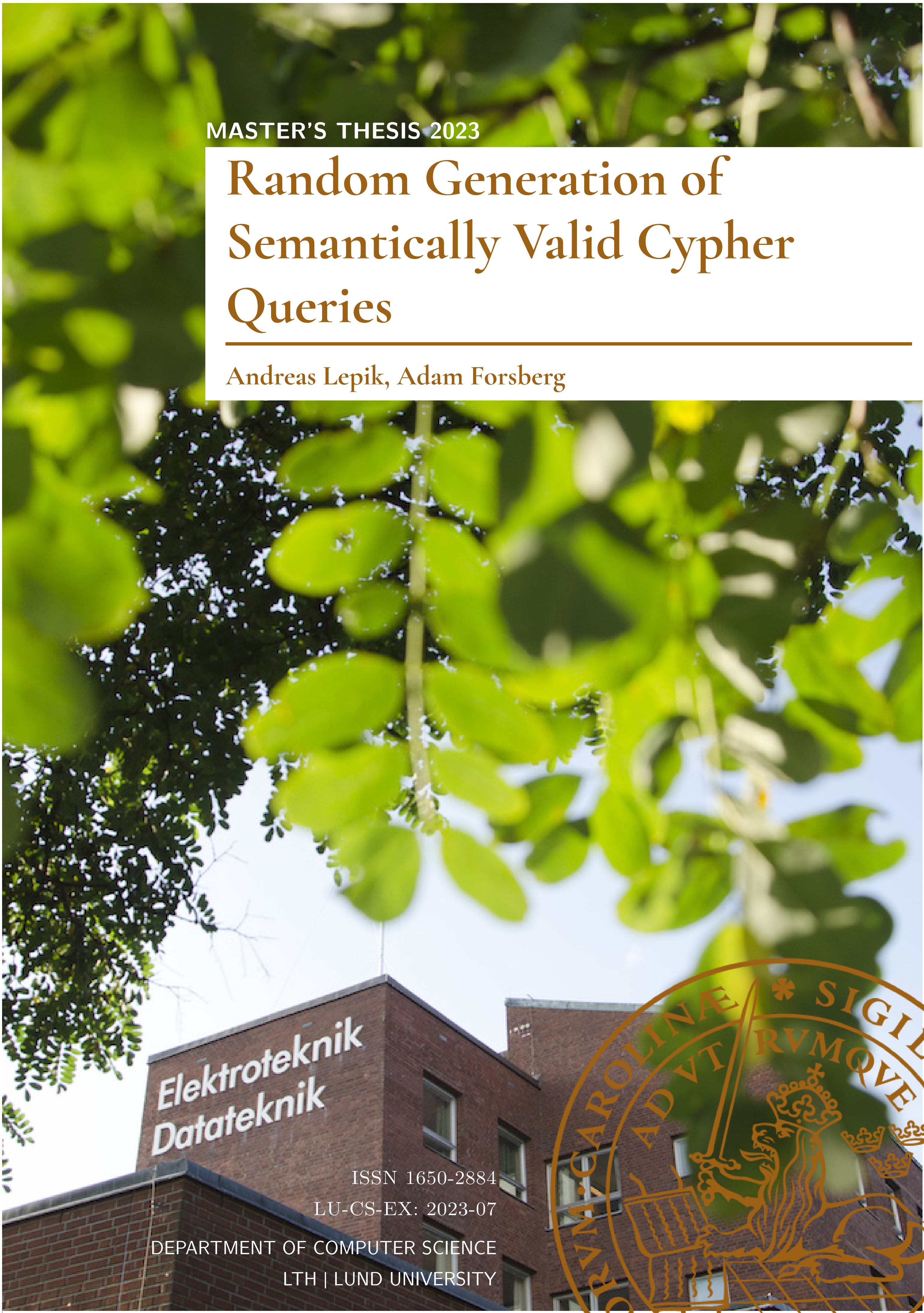


MASTER'S THESIS 2023

# Random Generation of Semantically Valid Cypher Queries

Andreas Lepik, Adam Forsberg

ISSN 1650-2884
LU-CS-EX: 2023-07
DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-07

# Random Generation of Semantically Valid Cypher Queries

Slumpmässig generering av semantiskt korrekta Cypher-frågor

**Andreas Lepik, Adam Forsberg**

# Random Generation of Semantically Valid Cypher Queries

Andreas Lepik
an8300le-s@student.lu.se

Adam Forsberg
ad0014fo-s@student.lu.se

March 28, 2023

Master's thesis work carried out at Neo4j Inc.

Supervisors: Tobias Johansson, tobias.johansson@neo4j.com
Niklas Fors, niklas.fors@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

## Abstract

Database management systems (DBMS) are integral tools at the center of many software applications, which means that these applications are deeply dependent on the correctness of their DBMS. In recent years, graph DBMSs have seen a significant rise in popularity, but they have not gotten the same amount of academic attention when it comes to testing as their relational counterparts. The most popular graph DBMS is called Neo4j and it has its own query language called Cypher.

In this thesis, we present a tool that generates random semantically correct Cypher queries. This query generator has a versatile set of use-cases and is built to be configurable, and in this thesis we have focused on using it for random testing of the Neo4j DBMS. Random testing of a DBMS means generating random but correct queries, executing them on the database and then checking whether the output is incorrect, which can be accomplished in a few different ways.

We found 25 confirmed bugs in Neo4j with our tool which suggests that it works well for the purpose of random testing of graph DBMSs. 21 of these bugs are already fixed, which suggests that the tool can find significant errors and not just irrelevant edge cases.

**Keywords**: Cypher, query generation, differential testing, property testing, graph database

# Acknowledgements

We would like to thank Tobias Johannson and the rest of the Cypher team at Neo4j for providing open-minded support and encouragement throughout the project.

We would also like to thank Niklas Fors for helping us navigate the tricky task of academic writing.

# Contents

**1 Introduction** **7**
1.1 Problem statement . . . 8
1.2 Contributions . . . 8
1.3 Contribution statement . . . 8
1.4 Outline . . . 9

**2 Background** **11**
2.1 Neo4j and property graph databases . . . 11
2.2 Cypher . . . 12
2.2.1 Reading clauses . . . 12
2.2.2 Projecting clauses . . . 13
2.2.3 Writing clauses . . . 14
2.2.4 Functions . . . 15
2.2.5 Other keywords and concepts . . . 15
2.3 Query processing . . . 16
2.4 Random Testing . . . 17
2.4.1 Error bugs and logical bugs . . . 18
2.5 ScalaCheck . . . 18

**3 Approach** **21**
3.1 Using the query generator for testing . . . 21
3.1.1 Pre-execution testing . . . 21
3.1.2 Execution testing . . . 22
3.1.3 Differential Testing . . . 22
3.2 Query generator . . . 23
3.2.1 Build AST with ScalaCheck . . . 23
3.2.2 Context . . . 25
3.2.3 Generating Expressions . . . 28
3.2.4 Fallback generators . . . 28
3.2.5 Graph schema . . . 29

3.2.6 Configuration . . . . . . . . . . 29
3.2.7 Static limitations . . . . . . . . . . 30

**4 Evaluation** **31**
4.1 Experimental setup . . . . . . . . . . 31
4.1.1 Updating queries with rollback . . . . . . . . . . 31
4.1.2 Comparing outputs and error handling . . . . . . . . . . 32
4.1.3 Performing the tests . . . . . . . . . . 32
4.2 Results . . . . . . . . . . 32
4.3 Highlighted Bugs . . . . . . . . . . 34

**5 Related work** **37**
5.1 GDSmith: Detecting Bugs in Graph Database Engines . . . . . . . . . . 37
5.2 Finding bugs in Gremlin-based graph database systems via Randomized differential testing . . . . . . . . . . 38

**6 Discussion** **41**
6.1 Usefulness . . . . . . . . . . 41
6.2 Generating Cypher ASTs and modularity . . . . . . . . . . 42
6.3 Practical value of random testing . . . . . . . . . . 42
6.4 Bug distribution . . . . . . . . . . 43
6.5 Non-empty results as a metric . . . . . . . . . . 43
6.6 Future Work . . . . . . . . . . 44

**7 Conclusions** **45**

**References** **47**

# Chapter 1
# Introduction

Database management systems (DBMS) are tools that are used to efficiently handle data and they are a core part of most modern software systems. While relational databases are the most common type, graph databases have been gaining significant popularity in recent years, with the most widely used graph database, Neo4j, being used by hundreds of Fortune 500 companies [1]. As the name suggests, graph databases represent the data as graphs. Because of this graph structure, traditional SQL-based query languages can not be used to query the graph databases, which is why Neo4j have developed their own query language called Cypher.

Bugs in DBMSs pose a serious threat to the applications that they are a part of. Since databases often act as an integrated part of a larger application, a bug in the DBMS might cause additional problems in other parts of the application. It is often taken for granted that the database in question always returns the correct results. This means that a logical bug, which results in the databases returning an incorrect result, is very hard to catch from a user point of view. There is also a possibility of the database crashing as a result of a bug. This too might lead to serious problems for the user, potentially making parts of the application useless or even crash.

One way of finding bugs in software systems is by using a technique called random testing (sometimes referred to as fuzz testing) which is an approach where randomly generated input is used to test properties of software systems. Extensive random testing has been done in order to improve the stability of compilers [8] and relational databases [9, 10, 4, 11], but finding logic bugs in property graph databases has not been given the same amount of attention [13]. Even though the random input produced in random tests may seem like nonsense that would not find any actually meaningful bugs, Marcozzi et al. present the conclusion that bugs found in random tests are just as relevant as bugs reported by users [8].

There is a variant of random testing called differential testing. The main idea behind differential testing is to give the same input to different implementations of the same system and compare the output. If the outputs are not equal, then that indicates that there is a bug in at least one of the instances. This technique allows you to easily compare correctness between optimizations and versions for example.

As the complexity of a system grows, the need for automated test tools naturally increases. Having to manually construct test cases is tedious and error-prone. When the potential input domain is large enough it is practically impossible. This situation is very much applicable to the Neo4j database. The complexity is constantly increased by new Cypher language features and internal optimizations. A query generator has the potential to be used in multiple areas, e.g. testing and benchmarking. Using an automated tool has multiple advantages when compared to manually writing queries. Using this computer-generated approach allows for producing both very large queries and also a very large quantity of queries. Having the tool randomly produce queries might also combine different language features in a way that is not commonly thought of by developers manually writing test queries.

## 1.1 Problem statement

The aim of this thesis is to implement a tool for automatically generating syntactically and semantically valid Cypher queries. Furthermore, effort will be directed at making the tool configurable, e.g. number of clauses, number of nested expressions, and type of clauses. We also implemented a test environment to demonstrate how the query generator can be used for random testing of graph databases.

## 1.2 Contributions

We have implemented a configurable open-source tool that can generate random Cypher queries. The tool has been used to improve the viability and robustness of the Neo4j DBMS and the Cypher query language through random testing. We have set up a property test suite that generates random but correct queries and feeds them as input to the Neo4j database. The suite contains two types of tests - non-executing tests and executing tests. The non-executing tests aim at testing the steps before actually executing the query, such as parsing, semantic analysis and planning. The execution tests aim at finding bugs when actually executing queries and are set up as differential tests.

During the project we found 25 unique bugs in the Neo4j DBMS confirmed by its developers. These 25 bugs consist of 24 bugs resulting in the DBMS wrongfully crashing (error bugs) and one bug resulting in an incorrect query result (logical bugs). This strongly suggests that the use of automated query generation can be highly beneficial.

However, the tool is not limited to Neo4j. The generated queries are fully capable of being used by any program or DBMS that supports Cypher. As mentioned above, an extensive query generator can also be useful in other testing scenarios, such as in performance benchmarking.

## 1.3 Contribution statement

The software presented in this thesis was almost exclusively written using pair programming. This means that every line of code has been reviewed and considered by both the authors.

The writing process of the thesis was a bit different. One author would write a first draft of each section, then the other would review and usually edit it. Most sections have had multiple changes done to them by both authors during the writing process.

## 1.4 Outline

This thesis is structured such that Chapter 2 contains necessary background information, primarily on graph databases, the Cypher query language and random testing. The information in this chapter aims at aiding the reader in understanding problems, concepts and discussions in the following chapters.

Chapter 3, describes our specific approach at solving the problem. In this section the general method is described, as well as certain implementation details concerning both the query generator and the test suite. This section highlights the main challenges, solutions and design choices made. This is followed by Chapter 4 which is an evaluation of the project. This entails experimental setup and results. It also contains a section highlighting a selection of the bugs found using our tool.

Chapter 5 summarizes related academic work.

Chapter 6 presents a discussion on our project and the test results. This includes a discussion of the usefulness of the tool and of its modular structure. It also contains an section on what kinds of bugs we found as well as suggestions on interesting related topics for future work.

In Chapter 7 we present our conclusions.

# Chapter 2

# Background

This first section aims to present relevant Cypher features and concepts that had to be taken into consideration when implementing the query generator tool. This can be viewed as a summary of the Neo4j Cypher Manual [2], with added details (that may be lacking in that manual) coming from our first hand experience with pushing the Cypher language to its limits during our implementation of the query generator. This section also describes the relevant concepts of random testing and differential testing.

## 2.1 Neo4j and property graph databases

Neo4j is a graph database, which means that data is structured and conceptualized as a graph rather than a set of tables. The base building blocks in graph databases are *nodes* and *relationships*. Nodes and relationships can in general be referred to as *entities*. Nodes are data points that can exist independently but relationships have to connect two nodes. All relationships in Neo4j are directed, meaning every relationship has a start node and an end node. Entities can have types used to categorize them, called *labels*. A relationship needs to have exactly one label, but nodes can have zero, one or multiple.

A simple graph can be seen in Figure 2.1. As we can see in the graph, each relationship has a label, in this case `WORKS_FOR`, `STUDIES_AT` or `FRIENDS_WITH`. The nodes are categorized by three different labels, `Company`, `Person` and `University`

All entities can store information in the form of *properties*. Properties are key-value pairs that are stored in the entities. In the example, the labels and properties of the nodes are shown in the boxes next to them. In this example there are no properties belonging to the relationships.

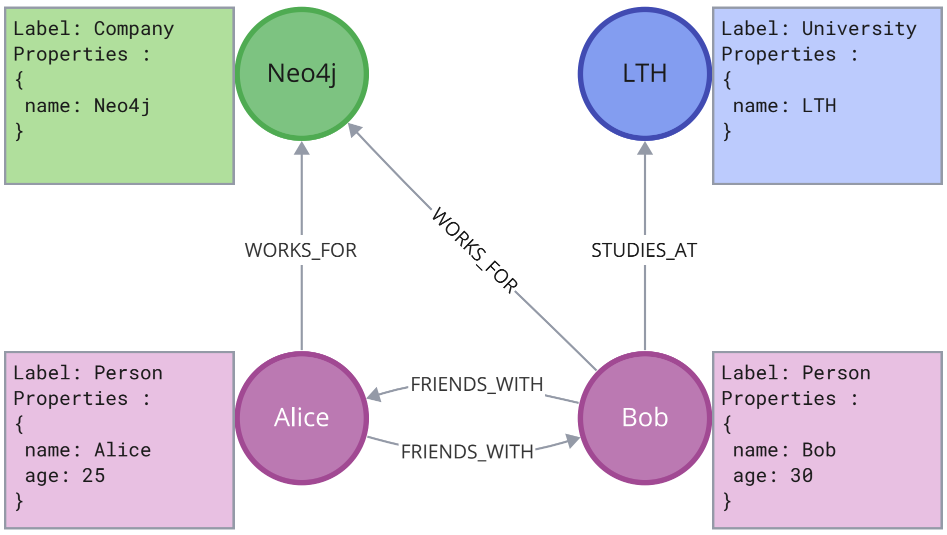


**Figure 2.1:** This is an example of how a graph can be represented in Neo4j. The label and properties of each node are shown in the box next to each node. In this example, no relationships have any properties, so only their label is written out.

## 2.2 Cypher

Cypher is the query language used in Neo4j to communicate with the database. Just like queries in SQL, a query in Cypher is made up of shorter sub-queries called clauses. The clauses are structured in sequence and each clause passes information to the next. A clause is expressed with a starting keyword followed by some body of text.

There are also non-functional clauses, such as administration clauses handling user privileges. Administration clauses are not part of our tool and will therefore not be explained.

### 2.2.1 Reading clauses

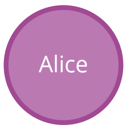
The `MATCH` clause is used to fetch data from the graph. It is used in the following way:

```
MATCH (p:Person)-[:WORKS_FOR]->(:Company)
RETURN p
```

The part after the `MATCH` keyword is called a *pattern* and describes what pattern of entities to look for in the graph. This style of expressing sub-graphs as patterns is a key feature of the Cypher query language. This particular pattern looks for any node `p` with a `Person` label that has a `WORKS_FOR` relationship to a node with a `Company` label. Nodes are expressed with round brackets and relationships are expressed as an arrow between them. The relationship arrows can have squared brackets containing more specific information inside them. The colons inside the entities are used to describe node or relationship labels, such as `Person`, `WORKS_FOR` and `Company` in the example above. The `p` in the first node is a variable declaration referring to a matching node. This variable is later referenced in the `RETURN` clause.

A `MATCH` clause can be followed by a `WHERE` sub-clause used to filter the result using some predicate.

```
MATCH (p:Person)-[:WORKS_FOR]->(:Company)
    WHERE p.name = "Alice"
RETURN p
```



The dot-syntax in the example above is used to refer to the property `name` of the node that the variable `p` refers to. `WHERE` sub-clauses can also be expressed within entities in patterns, meaning the following query is equivalent with the previous one:

```
MATCH (p:Person WHERE p.name = "Alice")-[:WORKS_FOR]->(:Company)
RETURN p
```

When trying to express properties that belong to entities in patterns, we can also use the curly bracket syntax. The following query will produce the same result as above:

```
MATCH (p:Person {name: "Alice"})-[:WORKS_FOR]->(:Company)
RETURN p
```

An alternative version of `MATCH` exists, namely `OPTIONAL MATCH`. This version allows users to specify optional parts of a matching pattern. If the `OPTIONAL MATCH` finds no match, it will return a result with `null` in its missing places rather than nothing at all.

### 2.2.2 Projecting clauses

Projecting clauses are clauses that return an output in the form of a sequence of rows, i.e. as a column. In fact, all variables in Cypher are treated as columns of values rather than single values. In the matching clauses above, the variable `p` might intuitively seem to represent only one node, but in fact it represents a sequence containing all the nodes in the graph that matches the pattern criteria. This is why the single variable `p` returned two nodes in the first matching query example. Viewed in table form, the output would look like this:

```
MATCH (p:Person)-[:WORKS_FOR]->(:Company)
RETURN p
```

| "p" |
|---|
| {"name":"Alice", "age":25}<br>{"name":"Bob", "age":30} |

The three projecting clauses that we will explain are `WITH`, `UNWIND` and `RETURN`.

The `WITH` clause is used to prepare the output of one clause to be used as the input of another one. It clears the previous variable scope and only passes on what is explicitly defined in the `WITH` clause.

The `UNWIND` clause is used to convert a list of values to a column of values.

The `RETURN` clause is a special clause that can only be placed last in the sequence of clauses in a query. It defines what data a query should return after it's done executing. The return clause can be followed by sub-clauses:

- `ORDER BY` sorts the results according to an expression. This sub-clause can also be used in a `WITH`-clause.
- `SKIP` *n* throws away the first n values of the results.
- `LIMIT` *n* only keeps n values and throws away the rest.

`WITH`, `UNWIND` and `RETURN` can all be used together with the `AS` sub-clause. The `AS` sub-clause allows the user to rename the returning expression and pass that renamed version to the next clause:

```
MATCH (p:Person)-[:WORKS_FOR]->(:Company)
    WITH p.name AS FirstName
RETURN FirstName
```

| "FirstName" |
|---|
| "Alice" |
| "Bob" |

### 2.2.3 Writing clauses

There are multiple clauses that write to the database. The `CREATE` and `DELETE` clauses modify the graph by using a pattern. The following query creates a new person Clara that works for Neo4j:

```
MATCH (c:Company {name: "Neo4j"})
CREATE (p:Person {name: "Clara"})-[:WORKS_FOR]->(c)
```

It is not allowed to delete a node with relationships tied to it, since this would cause dangling relationships that are missing a start or end node. To solve this one can start by deleting all connected relationships or instead use `DETACH DELETE` which automatically removes any connected relationships.

The SET clause is used to update node labels or entity properties.

The REMOVE clause is used to remove node labels or entity properties.

The FOREACH clause is used to update values based on a List or a column, commonly used together with the SET clause.

The MERGE clause uses a pattern, but works differently if the pattern exists in the graph or not. If it exists, it functions as MATCH. If it does not exist, it functions as a CREATE. There are special sub-clauses ON CREATE and ON MATCH that can be used to take different actions depending on the MERGE clause:

```
MATCH (c:Company)
MERGE (p:Person)-[:WORKS_FOR]->(c)
ON MATCH
    SET p.newly_hired = false
ON CREATE
    SET p.newly_hired = true
```

### 2.2.4 Functions

Cypher supports function calls as a way of performing various tasks that can not easily be expressed with the standard Cypher syntax. Users can define their own functions in Java code, but we will focus on the predefined functions already provided in the Cypher language. The syntax for calling functions is very familiar to that of many programming languages: the function name is followed by comma separated parameters wrapped in round brackets. Aggregating functions are functions that transform its input by reducing it column-wise. A good example is the count function:

```
MATCH (p:Person)
RETURN count(p) AS numberOfPeople
```

| "numberOfPeople" |
|---|
| 2 |

Many more types of functions exist in the Cypher language and in our query generator. There are for example many predicate functions and mathematical functions.

### 2.2.5 Other keywords and concepts

In this section, we will highlight some other concepts and keywords in Cypher that have been proven significant when developing the query generator. There are plenty of other details and keywords in Cypher that we will not cover in this section. These mostly work as one would expect, as is the case for boolean operators, mathematical operators, comparison operators, case expressions, etc.

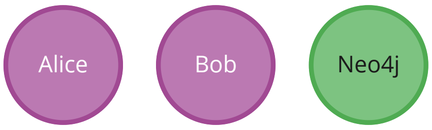
Cypher supports list comprehension and pattern comprehension as alternative syntax when defining lists. List comprehension uses another list as the base to iterate over when creating a new list. Pattern comprehension works the same, but it uses the column result from a pattern match as its base to iterate over. Lists are expressed with squared brackets. Below is an example of list comprehension.

```
WITH [element IN [1, 2, 3, 6] WHERE element < 4 | element^2] AS powerList
RETURN powerList
```

| "powerList" |
|---|
| [1, 4, 9] |

When matching on patterns, it is common to want to search for more complex node and relationship labels than just a simple single label. For this purpose Cypher supports what is called *label expressions*. It allows the user to express desired labels by combining conjunctions, disjunctions, negations and wildcard symbols (*).

```
MATCH (personOrCompany:Person|Company)<--(anyNode:*)
RETURN personOrCompany
```



Another powerful concept is variable length relationships in patterns. These can be used in pattern matching when we want to find a relationship connection between nodes that do not have to be of a specific length. An optional minimum length and an optional maximum length can be specified in the variable length relationship. Variable length relationships are only allowed in `MATCH` clauses and not in writing clauses, since they are ambiguous in what patterns they refer to.

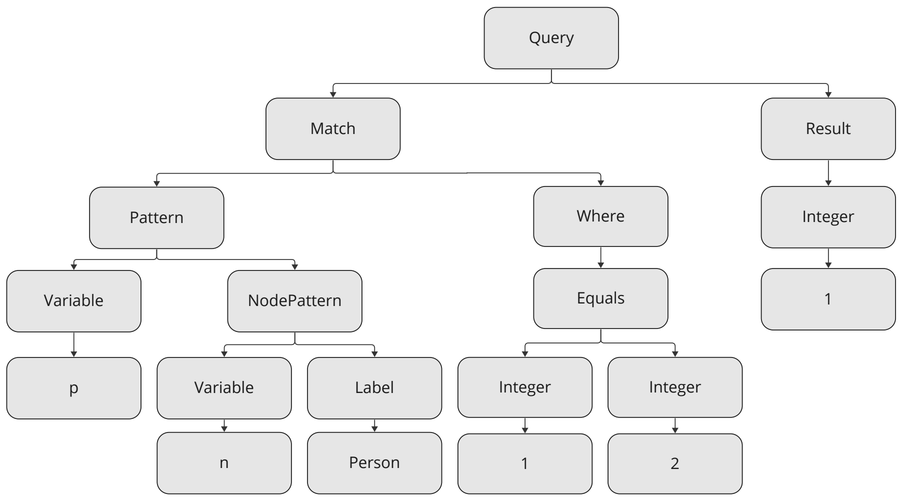
```
MATCH (p:Person)-[*2..3]->(nodeThatIsTwoOrThreeRelationshipsAway)
RETURN nodeThatIsTwoOrThreeRelationshipsAway
```

## 2.3 Query processing

The road from receiving a query to producing a result is long and involves several steps. Before executing, actions are taken to transform the query string to an actual executable plan. Looking at Neo4j specifically, these actions can be further divided into three distinct steps - namely *parsing*, *semantic analysis* and *planning*.

The role of parsing is to transform the query string to an *Abstract Syntax Tree* (AST). The AST acts as a tree representation of the query. Below is a simplified example of a query and a corresponding AST.

```
MATCH p = (n:Person)
    WHERE 1 == 2
RETURN 1
```



The next step, semantic analysis, verifies that variables and types are used correctly. This step also involves rewriting the AST to conform to a normalized structure, i.e. naming anonymous variables, expanding aliases and simplifying expressions.

The last step is to produce an actual execution plan of the query. There are two parts to this. First a *logical plan* is created. There are usually multiple working alternatives and producing the best one is no trivial task. To pick the best logical plan, the cost of each plan has to be calculated. However, this approach is too slow and instead an approximate cost based on graph statistics is used. The aim is to produce a working plan in a reasonable time.

Lastly, the logical plan is transformed into a *physical plan*. Since Neo4j offers multiple runtimes, this step involves taking the currently used runtime into consideration. Using one of the enterprise runtimes *parallel* or *pipelined* results in a very different physical plan compared to using the free open-source runtime *staged*.

## 2.4 Random Testing

Testing a program by providing random input and evaluating the result is called random testing. This technique is widely used in a variety of fields, such as compiler testing [8]. Using a program to generate test input greatly improves the number of tests compared to manually writing each input. Manually writing test input also risks not testing less obvious cases. The randomness hopes to mitigate this risk.

There are, however, different levels of randomness. On one end of the spectrum the input is a stream of completely random bits, on the other end the generated input is strongly guided by a model, i.e. a set of rules.

The terms used in the domain of random testing are applied quite broadly, making them quite hard to define. However, most seem to agree that the term fuzz testing describes a black-box technique with more or less completely random input. This technique is mostly used for simple stress testing of a program to see if it crashes when fed irregular input [6].

In other contexts, the term fuzz testing is used more synonymous with random testing. In this sense the term is used more freely to describe any test systems with random input, including those techniques that use a more guided input set [12]. These types of techniques with guided random input are sometimes referred to as gray-box fuzzing.

Another related term is property testing, which refers to generating random input according to a set of rules in order to test whether a system actually adheres to those rules or not. One can argue that this is a type of model-guided gray-box fuzz testing, but it is sometimes considered completely separate from fuzz testing.

Regardless of any strict definitions of these terms, from the perspective of this thesis, both well-guided and more erratic random testing has their benefits.

### 2.4.1 Error bugs and logical bugs

Since one aspect of our tool is to generate syntactically and semantically correct queries, it is very much guided by a model describing the syntax and semantics of the Cypher language. However, within this model, there is room for tweaking the randomness by configuring the tool. For example, if the user is only interested in queries using a certain type of Cypher clause, all other clauses can be turned off.

The possibility to tweak the tool to produce more or less bizarre queries is helpful when focusing on finding different types of bugs. In this thesis we distinguish between what is called error bugs and logical bugs. An error bug means that the program crashes when it is being fed a valid input. A logical bug, however, means that the returned value is incorrect.

Differential testing is a method for determining if the returned value is in fact incorrect. The basic idea is to feed the same input to multiple instances of a program that is expected to behave identically and compare the output. If the output differs one of the instances must behave wrongly. For example, different versions of the same DBMS can be fed the same input and expect to return the same output. It's also possible to compare different optimization settings of a DBMS against each other or to compare completely different DBMSs that support the same query language.

Since differential testing relies on difference in output, non-empty results are preferred. For this reason it might be useful to dial down the complexity and length of the generated queries since this is more likely to hit data in the graph. On the contrary, when looking for error bugs the more complex query the better.

## 2.5 ScalaCheck

A central and influential tool in the world of property testing is called QuickCheck [5]. It is designed to perform randomly generated property-based testing of programs. The user specifies what properties to test by defining them as Haskell functions. In our project we use a tool called ScalaCheck [3] which is inspired by QuickCheck, but written for Scala and Java.

In ScalaCheck there is a concept called *generators* that are used to generate random test input data. Technically speaking, a generator of type `Gen[T]` produces a value of type `T` when it is being evaluated. We can consider an example with a boolean generator returning `true` or `false`:

```
val generator: Gen[Boolean] = booleanGenerator()
val trueOrFalse: Boolean    = generator.evaluate()
```

The same generator can be evaluated multiple times and will then (most likely) produce different results. A powerful aspect of this generator concept is that it is possible to create custom generators. This opens up the possibility to invoke custom behavior with generators, such as returning `true` with a 90% chance instead of 50% in the example above. Custom generators also allow us to create generators for specific classes and objects, not only for primitive types. Custom generators are often implemented as functions with the `for-yield` syntax that return a `Gen` object. The boolean generator in the previous example can be implemented like this:

```
def booleanGenerator(): Gen[Boolean] = for {
    result <- oneOf(true, false)
} yield result
```

There are some very useful functions in the ScalaCheck library that can be used when defining custom generators. The function `oneOf`, as used in the example above, picks an option from a collection of choices. There is another function, called `frequency`, that works like `oneOf`, but it uses weights when randomly picking an option. A Boolean generator that returns `true` with a chance of 90% would look like this:

```
def booleanGenerator(): Gen[Boolean] = for {
    result <- frequency(9 -> true, 1 -> false)
} yield result
```

There is also a function `option` that produces a Scala `Option` object of its input, either producing `None` or, more likely, the input wrapped in a `Some` object when evaluated. It can be implemented like this:

```
def optionGenerator[T](input: T): Gen[Option[T]] = for {
    result <- frequency(1 -> None, 9 -> Some(input))
} yield result
```

# Chapter 3
# Approach

This chapter describes a selection of the most central concepts and design choices in our project. First the implemented test suite is explained. The different types of test are described, as well as what each specific test aims at. The next section describes the actual query generator used by the tests. This part goes into great detail about the use of ScalaCheck generators and aims at describing challenges and solutions. This includes the use of a *Context*, which role is partly to keep track of what variables are in scope in any given situation.

## 3.1 Using the query generator for testing

In order to test the Neo4j DBMS, we implemented a test suite with our query generator. The suite consists of different types of tests - targeting different parts of the system. The pre-execution tests target error bugs in parsing, semantic analysis and planning of a query. The execution tests target error bugs in the actual execution of queries. Lastly, the differential tests target logical bugs in various scenarios.

### 3.1.1 Pre-execution testing

Pre-execution testing focuses on the internal processes before executing a query, i.e. parsing, semantic analysis and planning the execution. For this reason the queries in this test are not actually executed. Avoiding execution of queries has multiple benefits. Since more complex queries tend to have longer execution times, this naturally puts a limit on the generated queries complexity. Avoiding execution therefore allows us to generate and test very much more complex queries. The complexity can easily be configured by for example increasing the number of clauses and maximum expression depth. Any query that produces a crash is flagged as a potential bug and examined. Since the chosen runtime affects the execution plan, each runtime - *Staged*, *Pipelined* and *Parallel* - is tested separately.

### 3.1.2 Execution testing

Another test is set up to specifically test the execution of queries. This allows for catching bugs that produce crashes in the execution phase. Compared to the pre-execution test, the complexity of the queries generated for this test is greatly decreased. Since executing very complex queries risks taking a huge amount of time, this trade off between complexity and execution time is needed.

### 3.1.3 Differential Testing

In contrast to the previous tests, which aim at detecting wrongful crashes, the differential tests aim at detecting logical bugs. As previously mentioned, this is done by feeding the same query to multiple equivalent databases, with some internal differences, and comparing the result. An outline of the algorithm can be found in Algorithm 1. In this specific example, the algorithm is performing differential testing by running the same query with different runtime settings on the same database instance. However, the same basic algorithm can also be used to describe other variations of differential tests.

**Algorithm 1** An outline of the random differential testing algorithm, using two different runtime settings.

```
Require: $GDB$: A Neo4j Graph Database instance
Require: $N_q$: The number of queries to generate
Require: $runtime1$: A Neo4j runtime setting
Require: $runtime2$: A different Neo4j runtime setting

for $i$ in $N_q$ do
    $query \leftarrow$ GenerateQuery( )
    $result1 \leftarrow GDB$.ExecuteWithRollback($query, runtime1$)
    $result2 \leftarrow GDB$.ExecuteWithRollback($query, runtime2$)
    if $result1$ crashed or $result2$ crashed then
        LogError($query, result1, result2$)        ▷ Queries should be semantically valid
    else if $result1 \neq result2$ then
        LogResultDiff($query, result1, result2$)
    end if
end for
```

#### Runtime Environments

One differential test makes use of the different runtime environments available in the Neo4j DBMS - *slotted*, *pipelined* and *parallel*. Slotted is free, open-source and generally slower than the other fast, but complex, enterprise versions. However, all runtimes are, of course, supposed to produce the same results.

The parallel runtime environment currently does not support queries that write to the underlying database. Therefore, to avoid missing potential bugs in the other runtime environments two different tests are set up. One test uses Parallel and Slotted, but is configured to not generate queries with writing-clauses. The other test uses Slotted and Pipelined and is freely generating all types of clauses.

### Optimization

Another type of differential testing involves toggling optimizations. One such optimization is to disable *eagerness*. Disabling eagerness essentially gives the planner more freedom in rearranging the order of operations. This, like many optimizations, increases the complexity and can therefore be error-prone. Similarly to the tests above, a query is fed to two databases - one with eagerness enabled, one with eagerness disabled. If the results differ it is flagged as a potential bug and investigated.

## 3.2 Query generator

In order to perform the random testing described in the previous section, a tool to generate random queries is needed. A central part of our query generating approach is that we first generate a correct Cypher AST and then convert that AST to a query in string form. This allows us to conceptualize the query in the same way as the Neo4j compiler does when interpreting it, by literally using the same Java classes.

A more detailed outline of how the query generation is structured in our random testing can be seen in Algorithm 2. As can be seen, the AST generator is created based on a *graph schema* and a specified configuration object. The graph schema contains information from the actual graph that the query is to execute on, such as which labels and properties exist. The generator can then be used to produce random ASTs, which is converted to a query. These concepts are explained in more detail in the following sections.

### 3.2.1 Build AST with ScalaCheck

The main tool we use to randomly generate ASTs are custom ScalaCheck generators. Essentially, each node type in a Cypher AST corresponds to its own *generator function*, i.e. a function that produces a generator of that node object. A generator function can call other generator functions in order to generate the required sub-trees. These nested generator function calls continue until reaching a generator function that produces a node without further calls. This can be thought of as reaching a *leaf* in the tree. A leaf can typically be a string literal or a variable reference. When we want to generate a query AST, we call the top generator function producing a generator for the root of the AST. This generator is used to produce actual Cypher ASTs.

Since what is allowed to be generated in a given moment is dependent on what has been generated before, a crucial feature of our tool is the possibility to pass information between generators. For example, a generator is not allowed to reference a variable that has not yet been declared. All necessary information is contained in what we call `Context`. The `Context` is described in further detail in section 3.2.2.

An example of how a typical generator function in the query generator looks can be found in Listing 3.1. This particular function's responsibility is to randomly generate a `Match` node in the AST and, consequently, all its necessary sub-trees. It achieves this by returning a

**Algorithm 2** An outline of the random differential testing algorithm, using two different runtime settings. This version goes into more detail about the query generation process than Algorithm 1. Line 1, 2, 4 and 5 are replacing the previous GenerateQuery function call.

```
Require: $GDB$: A Neo4j Graph Database instance
Require: $N_q$: The number of queries to generate
Require: $runtime1$: A Neo4j runtime setting
Require: $runtime2$: A different Neo4j runtime setting
Require: $Config$: An AST Generator Configuration object

  $GraphSchema \leftarrow$ LoadGraphSchema($GDB$)
  $ASTGenerator \leftarrow$ CreateASTGenerator($Config$, $GraphSchema$)
  for $i$ in $N_q$ do
      $AST \leftarrow ASTGenerator$.Evaluate( )
      $query \leftarrow$ RenderAsQueryString($AST$)
      $result1 \leftarrow GDB$.ExecuteWithRollback($query$, $runtime1$)
      $result2 \leftarrow GDB$.ExecuteWithRollback($query$, $runtime2$)
      if $result1$ crashed or $result2$ crashed then
          LogError($query$, $result1$, $result2$)
      else if $result1 \neq result2$ then
          LogResultDiff($query$, $result1$, $result2$)
      end if
  end for
```

generator of a `Match`-object, which is the same Java object actually used in the AST of the Cypher compiler when it is parsing a query.

Since a `Match` clause is allowed to declare and reference variables it is necessary to both receive a `Context` object as input and to return a new, possibly modified, `Context` for the next clause. This `Match` object requires a boolean to describe whether it is a so-called optional match or not, a `Pattern` node representing the match pattern and optionally a `Where` node containing a predicate.

The `optional` variable is a boolean and its generator function simply returns true or false with equal probability. This is one of the predefined generators in ScalaCheck. The `pattern` generator function is responsible for generating a match pattern. This function needs to take the `Context` as input and return a new `Context` since it can declare and refer to variables. The `where` generator function needs to take the modified `Context` as input since we want it to be able to refer to variables defined in the pattern. The `where` generator function is not allowed to declare new variables, which is why it does not return a new `Context`. The `Context` from the `pattern` generator function is returned by the match generator function. The `option` generator function produces a generator of a Scala `Option` of a given generator. Originally, this means is that the `where` generator function is returned with a 90% probability and a `None` object is returned with a 10% probability. However, in an attempt to increase the variety of the queries we changed the `option` generator probabilities to 50/50. This has the effect that half of the generated `MATCH` clauses will have `WHERE` sub-clauses.

```scala
def match(context: Context): Gen[(Match, Context)] = for {
    optional                <- boolean()
    (pattern, newContext)   <- pattern(context)
    where                   <- option(where(newContext))
} yield (Match(optional, pattern, where), newContext)
```

**Listing 3.1:** An example of an AST node generator function in the AST generator displayed in Scala code. This particular one is used to generate the AST nodes that represent `MATCH` clauses. The function has been slightly modified in order to be more readable.

## 3.2.2 Context

The domain of valid clauses and expressions is restricted and dependent on what clauses and expressions have been generated before. One example of this is the fact that previously declared variables should be available to the following clause. For this reason the query generation makes use of a so-called Context object to structure and update this information. We implemented the Context as an immutable data storage, which means that every time we want to change the Context, we create a new updated instance as exemplified in Section 3.2.1. This is a vital design decision since it allows us to easily keep track of how the Context is morphed and distributed throughout the generator functions.

### Variables

Keeping track of available variables constitutes one of the main challenges when generating queries. When using variables in a query, two main problems present themselves. First, avoid referencing undeclared variables or variables whose value is of an incompatible type. Second, avoid declaring a variable name that already exists.

To solve these problems the context holds all relevant information regarding variables. This object is passed to each generator that might need to declare or reference a variable. The Context object can be thought of as a set of tuples `(Name, Type)`. When a variable is declared, its name and type are simply added to the set. When a generator tries to use a variable reference it checks the set to find available variables of the requested type.

To clear the name space, all variables in the Context are simply deleted. As described in Section 2.2.2, `WITH` and `RETURN` are two such clauses. The code snippet below illustrates what variables are in the `Context` after each clause of the query.

```
MATCH (p:Person)                        {(p, Node)}
MATCH (c:Company)                       {(p, Node), (c, Node)}
WITH 123 AS myNumber                    {(myNumber, Integer)}
RETURN myNumber + 2 AS result           {(result, Integer)}
```

In the query above the node variables declared in the `MATCH` clauses are added to the Context. The `WITH` clause, however, clears the Context and declares a new variable of the type Integer, called `myNumber`. The `RETURN` clause generated an addition expression, which takes two numbers. The first one is a variable reference, but the second one is simply an integer literal. To use the variable the clause first asked the Context for a variable of the type Integer. Since a variable of that type existed in the context, i.e. `myNumber`, the request succeeds and the variable is used.

However, if the Context did not contain a variable of the requested type a new Integer-expression would simply be generated. The above example also demonstrates that the return type of the addition expression `myNumber + 2` is Integer. We know the return type of all generated expressions since the expressions are generated based on a type from the start. Generating expressions are explained in detail in section 3.2.3.

Removing variables from the Context is usually quite straightforward. In the case of the `WITH` clause above, the generator function simply discards the old Context and creates a new one with the new variables. A unique problem arises when we introduce modifying clauses, such as `DELETE`. Consider the following query:

```
MATCH (p:Person {name: "Alice"})
DELETE p
RETURN p.property
       ^
```

The node `p` is deleted, but the variable is still in scope from the perspective of the Cypher compiler. This means that the variable has to stay in our Context scope, since it would be semantically incorrect to overwrite it, but we also have to avoid using it, since using it is no longer meaningful. To solve this, we implemented a way to block variables in the Context from being used, without deleting them from the variable declaration scope.

Another layer of complexity is added when we consider the fact that when using `DELETE` and in particular `DETACH DELETE`, we run a high risk of deleting parts of the graph that is referenced in other variables than the one we are deleting. Consider the following example:

```
MATCH (p:Person)
MATCH (p2:Person {name: "Alice"})-[r]->()
DETACH DELETE p
RETURN p2.property, r
       ^           ^
```

In this example, both p2 and the relationship r are deleted, even though they are not mentioned in the DETACH DELETE clause. The solution for this is to block all relationships in the Context if any relationship is deleted and to block all relationships and all nodes in the Context if a node is deleted using DETACH DELETE. This might seem limiting, but it is a statically safe operation that we can do without having to actually evaluate each DELETE clause on the graph, which would be unreasonably complex.

## Expression Depth

Nested expressions greatly improves the coverage since many expression combinations will be generated. However, allowing expressions to generate more expressions in their sub-trees leads to a potential problem of near endless recursion. This is solved by having a variable in the context signaling the current expression depth. Every time an expression is to be generated, the current expression depth is compared to the configured maximum expression depth. If the maximum depth is reached a literal of the given type is created, allowing no further recursion. Otherwise the current expression depth variable is increased and a normal expression is generated without further constraints.

## Context Mode

Another feature of the Context is a variable used to describe if what is currently being generated adheres to certain extra rules regarding the syntax. For example, a CREATE clause does not allow undirected relationships or variable length relationships. Using a Context variable to signal these certain modes allows for flexible generator functions, more easily read code and avoids a lot of code duplication. The snippet below is used to illustrate the extra restriction put on patterns in CREATE, compared to MATCH.

```
MATCH (n)-[r]-(m)                   Legal
MATCH (n)-[r*]-(m)                  Legal
CREATE (n)-[r]->(m)                 Legal
CREATE (n)-[r]-(m)                  Illegal
CREATE (n)-[r*]->(m)                Illegal
```

## Order of clauses

It is not the case that all clauses are allowed to be followed by every other clause type. More specifically, a reading clause - such as MATCH, is not allowed to follow directly after a writing clause - such as CREATE. For this reason the Context contains a flag describing whether the previous clause was a writing clause. If so, this restricts what type of clause is allowed to be generated. This flag is enough to avoid generating any illegal order of clauses.

### 3.2.3 Generating Expressions

Generating expressions with correct types is a key part of our tool. To solve this, a top-level generator function `expression` is always used when generating expressions. This function takes a Cypher type as input, and generates an expression that evaluates to this type. Looking more closely, the `expression` function calls the generator whose role is to generate expressions of the given type. This generator, in turn, contains many more generators, each of which returns the given type. Going deeper down this generator-tree we eventually reach a leaf generator returning an AST-node. This simplified structure can be seen in figure 3.1.

```
def expression(exprType: CypherType, context: Context):
    Gen[(Expression)] = {
    if (context.exprDepth > MAX_EXPR_DEPTH)
        literal(exprType)
    else
        context.exprDepth += 1
        exprType match {
            case CTInteger => intExpression(context)
            case CTString => stringExpression(context)
            ...
        }
}

def intExpression(context : Context): Gen[Expression] = oneOf(
    additionExpression(context),
    variableReference(context),
    functionInvokation(context),
    ...
)

def additionExpression(context: Context): Gen[Expression] = for {
    leftExpression <- expression(CTInteger, context)
    rightExpression <- expression(CTInteger, context)
} yield AdditionExpression(leftExpression, rightExpression)
```

**Listing 3.2:** A simplification of the general structure when generating expressions. The functions has been modified in order to be more readable.

As can be seen in figure 3.1, this structure makes it convenient and reliable to guarantee type safety. This structure also makes it easy for expressions to generate more expressions. This can be seen in `additionExpression`. Allowing for nested expressions like $(1+(2+3))$.

### 3.2.4 Fallback generators

As mentioned before, our tool generally makes decisions from a top-down perspective in the tree. For example if an expression is needed, it first makes a decision on what type it should return and then generates the expression accordingly. This has the benefit of not having to evaluate expressions to know their types and it also avoids complicated rewrites of the tree based on those types. It does however sometimes lead to situations where the generator functions find themselves going down routes that are not always possible to conclude. An example could look like this: We need a string, so we generate a string expression, which generates a variable reference but there are no string variables declared yet. We solve this problem not by regenerating the entire expression but by using fallback generators. Whenever a generator has a chance to fail like this, we structure the generator function to wrap its return value in a Scala `Option` object that allows the function to return `None` if it failed. We can then manually set up conditions to trigger fallback generators to replace any failing generators and only regenerate what is absolutely necessary. In our example above, we could generate a string literal instead of the variable reference, since we know about the desired type.

### 3.2.5 Graph schema

In order to generate more meaningful and complex queries that to a greater extent references actual data in the graph a *graph schema* is used. The generator can use this schema to access information about the actual graph its generating queries for.

The graph schema is created by deriving some static information about the graph. The stored information includes all the existing node and relationship labels and all the property names and their types. We also store a lookup table from every label to all properties found on any entity with that label in the graph. This information allows us to perform quite sophisticated cross-referencing when we want to pick properties in `MATCH` patterns, for example. Consider the following example where we are generating a pattern and have decided to include a property in the node:

```
MATCH (p:Person|Company { })
                         ^
```

Here, we can look up all node properties that can belong to either `Person` or `Company`. One of the available properties are picked at random and the property type is used to generate the right-hand side expression based on that type:

```
MATCH (p:Person|Company {age: 2 * 15})
```

In a similar vein, this information is used to reference properties of variables based on a type. In the following example we know that is is safe to pick `age` property of `p1` when we need to generate an integer expression as part of the addition.

```
MATCH (p1:Person)
RETURN 1 + p1.age
           ^
```

### 3.2.6 Configuration

In order to diversify the use cases of the query generator, it has support for configuring different parameters that guide how the generated queries will look like. For example, if the generated queries are used on the parallel runtime, the tool can easily be configured to not generate any writing clauses. This is done by tweaking the frequency weights used in generator functions.

This is implemented by storing all the frequencies in a configuration class, where most frequencies have a default value of 1. This means that the query generator can be conveniently tweaked on a case-by-case basis by providing the generator a configuration instance. For example, to disable `LIMIT` clauses, the user would change the weight from its initial value 1 to 0:

```
ASTGeneratorConfig config = new ASTGeneratorConfig()
config.limit = 0
ASTGenerator generator = new ASTGenerator(config)
```

A similar configuration can be made in a special test-case where the query generator should not generate any `MATCH` clauses:

```
config.match = 0
```

Or if a use-case would want more variable references:

```
config.variableRef = 10
```

Other than modifying frequency weights, the configuration can also be used to determine the number of clauses in a query and the max expression depth. The configuration class is also extended to several different predefined default configurations based on common use cases. For example, if we would like to run queries in the parallel runtime we can use the predefined `ParallelConfig` instead of setting multiple parameters manually.

### 3.2.7 Static limitations

There are some parts of the Cypher language that have been proven to be problematic for the purpose of using the query generator for random testing. For example, the `LIMIT` clause is used to cut the output to a certain number of results. If there is no explicit and valid ordering to this result, then the Cypher implementation will not guarantee which parts of the results are cut out and which are preserved. What this means in practice is that the result is not deterministic and that it therefore can not be used for differential testing since two different outputs can both be correct. The same problem exists with the `SKIP` clause.

To solve problems like this, we have used the configuration object to limit the generated queries to only cover a meaningful sub-set of the Cypher language. One of these constraints is to never generate `LIMIT` or `SKIP` clauses in differential testing, they are simply too disadvantageous to be worth having in the generator. This works well with our Configuration object structure, since it allows us to enforce this static limitation only where it is required, so non-differential tests can produce queries containing `LIMIT` and `SKIP`.

Similar decisions have been made about less clear cut parts of the language, such as with the `DELETE` clause. If a `DELETE` is to be used without its `DETACH` prefix, it has to target a node without any relationships. To avoid this problem we have decided to always perform `DETACH DELETE` instead. This is a typical situation where we decided to limit some part of the generator in order to prioritize other aspects of the project.

# Chapter 4
# Evaluation

This chapter describes how we evaluated the query generator tool and the bugs found by using it. The first section presents how the tests are executed using rollback and how the results are compared in differential testing. The second section presents and categorizes the found bugs. The third section presents a selection of highlighted bugs along with short descriptions of their significance.

## 4.1 Experimental setup

In this section we will go over some important details about how we set up and used the query generator for random testing.

### 4.1.1 Updating queries with rollback

The fact that we wanted to include updating clauses such as `WRITE` or `DELETE` when testing did present some quite tricky problems. Simply executing the queries on the same database instance risks that some query creates thousands of entities or deletes the entire graph. This would make running tests very impractical for several reasons. For one the execution time could vary greatly which could be impractical. The greater problem is that the query results would not be identical every time a query is executed, so reproducing bugs could be impossible since the database state would not be saved.

The solution for this was to roll back the changes made for every query that was executed. We used Neo4j's transaction API to first start a transaction, then execute the query, evaluate the results and then roll back the changes made before closing the transaction. This means that the database state was reset after every query executed, which allowed us to use updating clauses in our query generator for our random tests.

### 4.1.2 Comparing outputs and error handling

When running differential tests, we want to compare two outputs that are supposed to be the same, but these comparisons present a problem. The default comparison function for comparing nodes from the graph is to compare their `id`, a number assigned to each node when created. The problem with this is that our query generator contains writing clauses, but there is no guarantee that the `ids` are assigned in a predictable order. This means that if two nodes are created in the same query on an empty graph, we can not know which node would get `id 0` and which one would get `id 1`. When comparing the output from such a query on two different Neo4j implementations, we might correctly get the same node in the two results, but with different `id` numbers, which would mean that we incorrectly would judge them as not the same. This problem is solved by overwriting the comparison function for nodes so that they compare their label and all their properties instead of comparing their `id` numbers. This solution does run the risk of our comparison function identifying two different nodes with identical labels and properties as the same, but we consider this trade-off to be reasonable since the incorrect `id` comparisons were quite common, while we believe that incorrect property comparisons should be very rare.

### 4.1.3 Performing the tests

While developing the query generator, we needed a way to test if it was functioning properly and produced semantically valid queries in the way we intended. We solved this by implementing a test suite early in the development process and let the tests function in both directions between the query generator and Neo4j. That is, we executed a lot of randomly generated queries on a Neo4j instance and when we encountered an error we would examine the query and the error and identify if the problem was originating from a wrongfully generated query or from a bug in the DBMS. In order to identify a problem like this, typically we would have to reduce the queries in order to get rid of irrelevant parts. This was done manually by removing parts of the query one step at a time and checking if the error would still occur between the changes. This workflow resulted in us finding the first bugs quite early in the development process and that we continuously found bugs throughout the project as we expanded the query generator and kept testing it.

When we implemented the first test suite, we opted for an execution test and a simple differential test that would not compare the outputs but only the output sizes. The reasoning behind this was that we wanted to focus on the query generator first as it was the main part of our project. The differential tests got more sophisticated later in the process. This perhaps had the unintended consequence that we ran more and better execution tests than differential tests early in the process.

## 4.2 Results

The semantically correct random query generator is a versatile tool, and as we have argued in our previous sections, it has use cases outside the realm of random testing. It will be available in the public Neo4j open source repository and can rightfully be viewed as a result in itself. However, this section mainly focuses on the results from using it in random testing.

The random testing can most effectively be judged by the bugs it found. These bugs are presented in Table 4.1 and 4.2. During our 15 weeks of project development time, we found a total of 25 confirmed unique bugs, where 21 of these were fixed within that same time span. The vast majority of the bugs were what we categorize as error bugs, only one logic bug was found because of conflicting outputs. Most bugs, around 50%, were found in the planning part of the query processing stack.

| Description | E/L | Error Location | Fixed |
|---|---|---|---|
| IN in list | Error | Parsing | ✓ |
| WITH list MERGE | Error | Parsing | ✓ |
| true = NOT false | Error | Parsing | - |
| EXISTS scope | Error | Semantic analysis | ✓ |
| nested EXISTS XOR | Error | Semantic analysis | ✓ |
| WHERE exists() | Error | Semantic analysis | ✓ |
| CREATE list compr | Error | Semantic analysis | ✓ |
| list compr scope | Error | Semantic analysis | ✓ |
| EXISTS scope | Error | Semantic analysis | ✓ |
| property EXISTS label | Error | Semantic analysis | - |
| CREATE list index EXISTS | Error | Planning | ✓ |
| variable length rel WHERE | Error | Planning | ✓ |
| MERGE SET EXISTS | Error | Planning | ✓ |
| MERGE CREATE EXISTS | Error | Planning | ✓ |
| MATCH EXISTS MATCH | Error | Planning | ✓ |
| variable length rel DISTINCT | Error | Planning | ✓ |
| CREATE list index EXISTS | Error | Planning | ✓ |
| EXISTS <> EXISTS | Error | Planning | ✓ |
| MATCH SET DELETE | Error | Planning | ✓ |
| FOREACH SET | Logic | Planning | ✓ |
| aggregating function EXISTS | Error | Planning | ✓ |
| WITH * WHERE | Error | Planning | - |
| deleted ref | Error | Planning | Blocked |
| MERGE var-lenght rel | Error | Runtime | ✓ |
| MATCH UNWIND MATCH | Error | Runtime | ✓ |

**Table 4.1:** This table presents all the individual unique bugs we found during our project. The columns show whether it was a logic or an error bug, where in the query processing stack the bug occurred and whether it was fixed during our project time or not. The blocked bug was blocked because it highlighted a part of the Cypher language that as of now is undefined.

| | Found | Fixed |
|---|---|---|
| Parsing | 3 | 2 |
| Semantic analysis | 7 | 6 |
| Planning | 13 | 11 |
| Runtime | 2 | 2 |
| Error bugs | 24 | 20 |
| Logic bugs | 1 | 1 |
| Total bugs | 25 | 21 |

**Table 4.2:** This table shows the number of bugs found and the number of bugs that were fixed during the project. All the fixed bugs are also counted in the Found column. It presents them both in the Error/ Logic bugs categorizations and in the bug types from the Cypher compiler stack. All these presented bugs were confirmed as bugs by the Cypher team at Neo4j.

## 4.3 Highlighted Bugs

This section presents a selection of reported bugs found by the tool. The goal of this selection is to highlight both the simplicity and the variety of the bugs found. The generated queries that originally triggered the bugs are usually quite long and complex. The queries shown below have been manually simplified and shortened as much as possible to only contain what is necessary to trigger the bug. The queries are presented exactly as they were reported to the Neo4j developers.

The following very short query produced a parsing error:

```
MATCH ()
  WHERE true = NOT false
RETURN 1
```

Worth noticing is that very slight modifications of the same query did not produce an error. For example, replacing `=` with `AND` or changing the `WHERE` condition to `NOT false = true` executes without error.

Like above, this query is also very simple and produces an error in the planning stage:

```
WITH *
  WHERE true AND true
RETURN 1
```

The simplicity suggests that it is quite likely to affect users, i.e. it is no strange corner-case that is unlikely to occur in a real situation. This particular query is just one example of multiple queries triggering the same bug. This adds to the likelihood of users being affected by it.

The following is an example that involves a relatively new Cypher language feature called variable length relationships:

```
MATCH ()-[*]-(n)
  WHERE false
RETURN 1
```

Apart from showing the range of language features triggering bugs, this particular bug is selected to illustrate the need for rigorous testing of new features. However, creating adequate tests can be both hard and time consuming, showing the value of automated query generation.

The following query found a parsing bug that was introduced on a newer version of the Neo4j DBMS:

```
MATCH ()
RETURN [true IN [true], false]
```

This particular query executes without error on an older version of the DBMS. This means that somewhere along the way the implementation changed and introduced a bug. This is another example of the value of using automated tests. It is not only handy when implementing new features, but also when trying to change or improve the implementation of previous features.

This query only triggered an error on one particular runtime setting:

```
MERGE ()-[r:L]-()
  ON CREATE SET r.prop = 1
```

This shows the variety of bugs found and highlights the versatility of the query generating tool.

The following query produces a logical bug:

```
MATCH (n:User)-[]->(:Movie)
FOREACH ( i IN [1] | SET n:User:Movie )
RETURN 1
```

This query returned completely different results depending on if eagerness was enabled or disabled. This type of bug can be very hard to find since they do not cause an error. In the worst scenarios these bugs can linger for a very long time before being caught, if ever.

# Chapter 5
# Related work

This chapter summaries two papers exploring related topic. Some similarities and differences are briefly highlighted.

## 5.1 GDSmith: Detecting Bugs in Graph Database Engines

Lin et al. present a tool for testing graph DBMSs, called *GDSmith*, and describe it as *"the first black-box approach for testing graph database engines"* [7]. They accomplish this by having a query generator that generates Cypher queries and executes them on three different open source graph DBMSs, including Neo4j Community Edition. Just as in our project, they look for both error bugs and logic bugs through random testing and differential testing. They chose Cypher rather than Gremlin as their query language of choice, since it is supported by ten graph DBMSs and there are translation tools from Cypher to Gremlin. *Gremlin* is another graph query language with a more functional approach to it's syntax. They performed differential testing both as cross-DBMS (comparing completely different DBMS, such as Neo4j and RedisGraph) and cross-version (comparing the same DBMS but different versions). According to their results, these methods seem to be equally efficient at finding logic bugs.

There are many similarities between GDSmith and our project, but there exists quite a few differences as well. Instead of running their queries on existing databases, they randomly generate their data. GDSmith also generates queries with a two-step approach. First they generate what they call a skeleton, which defines the clause and sub-clause ordering but skips generating expressions and variables. Then they fill in each gap in the skeleton query one by one by generating expressions and variables. This way, they can first focus on the clause-syntax when generating the skeleton, and then focus on upholding variable scope correctness and type safety during the second phase. Much like our implementation, they regenerate sub-trees (expressions) when they try to fetch a variable of a certain type, but can't find any and use a max-depth for expressions. They mention that they aim for optimal query sizes (number of clauses and max expression depth) that are long enough to not be trivial but short enough to hit data. To increase the data hit rate they present a quite novel query mutation strategy. The concept is that they save the generated queries that hit data and then alter them slightly hoping that the altered versions also will hit data. Much like our implementation, they have a similar Context object that they pass around in their query generation to keep track of variables. The query generator in GDSmith does not produce any writing clauses.

In their testing, they found 27 bugs across the three graph DBMSs, 14 of which were confirmed by the developers. 15 of the reported bugs were related to Neo4j, 6 being confirmed. In contrast to our result, two thirds of all the bugs that the GDSmith team found were logical bugs. Their tool is not openly available.

## 5.2 Finding bugs in Gremlin-based graph database systems via Randomized differential testing

Zheng et al. present an approach for finding logic bugs in graph databases that support the Gremlin query language [13]. Similar to our project, the goal is to find bugs through differential testing. They randomly generate syntactically correct queries and execute them on different database instances containing the same graphs and compare the results. Like GDSmith, they performed cross-DBMS differential testing and randomly generated their test data. This means that the generated queries start with updating clauses to add the test data, then continue with the actual query. They manually reduce their complex queries when finding a difference, and then investigate to see where the bug originates from and whether it is a new unique bug or one previously found. This is very similar to how we approached sorting through potential bugs.

Gremlin is a graph querying language built by Apache TinkerPop designed to traverse graphs using syntax similar to how data flows in functional programming languages. The reason they chose this as their query language is that it has broad cross-DBMS support. However, they did run into some issues connected to different DBMSs supporting different subsets of the Gremlin language. They bring up the lack of a SQL-like graph database standard as their reason for using Gremlin, but with the current development of GQL being closely tied to Neo4j, Cypher might be the best option in this regard in the future.

Due to the functional nature of the Gremlin query language, they had to be careful with the types being piped in between parts of the query. This is similar to how we had to type check the use of variables in our generator. They achieved this by building a static model of how a sequence in a query is allowed to be built, and then generated the queries by following the rules of that model. This is quite similar to our generator function approach, all though implemented a bit differently. They do not include updating queries in their testing, they only use them to initiate the randomly generated graphs. Just like GDSmith, they also use a query generating approach where they first generate the basic structure of the query and then fill in the gaps.

Using their described approach they found 18 bugs, where three of them were related to Neo4j. Even though their set goal was to find logic bugs, the vast majority of their bugs seem to be what we define as error bugs. This distribution is more similar to our results than GDSmith. The tool that they developed to implement their graph database testing approach, *Grand*, is available as open source software on their github [1].

[1]`https://github.com/tcse-iscas/Grand`

# Chapter 6
# Discussion

This chapter discusses the query generating tool as well as random testing and our results. It starts by discussing the value of a query generating tool and the implementation and is followed by a discussion around random testing and our results.

## 6.1 Usefulness

The amount of confirmed bugs strongly suggest that query generation combined with testing can be highly useful. However, one of the primary strengths of the query generation tool is its broad applicability. In this project we have mainly been focusing on applying it to various types of testing, but it is in no way limited to this. On the contrary the design of the tool makes it easy to both extend and configure for specific needs. For example, it could be valuable for aiding a test-first workflow when developing new language features. This new language feature can easily be implemented in the query generator and used to test the new feature before publishing it. Allowing easy use of such an approach can possibly eliminate the amount of bugs hiding in newly released versions. The same approach is of course possible if one is changing the implementation of a certain feature.

The flexibility of the tool also makes it useful in completely different scenarios, such as benchmarking. If a specific part of the language is less covered in benchmarking, the tool can easily be tweaked to generate random queries of that requested type. The configurability can also be used to avoid catching the same known bugs over and over again. If a certain bug is not interesting, perhaps because it is currently being fixed, the tool can be tweaked to avoid generating queries that trigger this particular bug.

A threat to keep in mind when discussing the usefulness of our tool and test suite involve false positives. If too much time is needed to sift through false positives the practical use is greatly decreased. If a query triggers an error, there exists a possibility of the query being wrong and the error actually being correct. This comes down to the correctness of the query generating tool. For this reason, queries are flagged as potential bugs and need to always be manually confirmed. The risk of the query generator being faulty can be mitigated by constantly updating and correcting the tool whenever an issue is found. This way the tool keeps getting better and more accurate.

Another case of false positives involve known limitations of the query generating tool. Because of the way the tool works, it never actually evaluates expressions. This makes it impossible for the tool to avoid generating queries with expressions that, when evaluated, result in illegally large numbers. These queries will rightfully cause an error when executing. It is possible to mitigate these types of errors to some extent by configuring the tool, but they are hard to completely eliminate without losing much of the tool's expressiveness. If too much time is needed to sift through false positives the practical use is greatly decreased.

## 6.2 Generating Cypher ASTs and modularity

The decision to build the query generator around Cypher ASTs has both benefits and drawbacks. The biggest gain from this is the extraordinary modularity of the system that comes from building it as generator functions of AST node types. If a new feature is to be introduced, all one has to do is define a new generator function for that AST node type and make sure it follows the structure of the other generator functions (such as correct Context handling). After this, it's very simple to introduce the new generator wherever it would fit as a sub-tree to existing node types in the AST. This ease of expansion is in our opinion key for the query generator to be useful and effective in the long term.

The fact that the AST built by the query generator uses the same classes as the Cypher compiler has the benefit that many Neo4j developers will have an easier time understanding and expanding the query generator since the AST node classes will be familiar to them. A potential drawback of this setup is that the query generator is now dependent on the AST node classes. If the structure of future versions of the Cypher AST changes, chances are that the query generator would have to be changed as well in order to function, which is not ideal. This could have the unintended effect that the developers need to continuously update the query generator as the Cypher compiler is updated. But if not, the query generator could just be tied to a Neo4j version in order to not require it to receive updates to function. It would also be possible to package the AST generator together with its dependencies from the Cypher compiler in order to allow it to function as a stand-alone program.

## 6.3 Practical value of random testing

A potential risk of using randomly generated queries for finding bugs, as in this project, is to only find bugs of little practical importance. The reason being that the randomly generated queries generally do not resemble user queries very well. However, our results stand in contrast to this and bugs from similar random testing have been concluded to actually be as important as user found bugs [8].

There are two reasons to suggest that this risk might not be a problem with our specific project. First, visually examining the bug-triggering queries shows that many of them are simple and look very reasonable. In addition to this, the main argument is that the Neo4j developers deem the reported bugs to be of such importance that most have already been prioritized and fixed, as seen in Table 4.2. This would not be the case if they saw little practical use in fixing the discovered bugs.

## 6.4 Bug distribution

As shown in table 4.2, a very large majority of the found bugs are error bugs. This is specially interesting in comparison to GDSmith, which found a majority of logical bugs. It's hard to determine what the reason for this is, but we suspect a few contributing factors. One such factor is simply the fact that we've run more tests aimed at error bugs. As described above, this has to do with the fact that our differential tests were not fully developed until relatively late in the project.

Another possible factor for these very different results lies in what was actually tested in the differential tests. In differential testing, it seems plausible that the greater the difference between the instances being compared, the greater the chance of getting different results. GDSmith performs cross-DBMS and cross-version differential testing, while we perform cross-runtime and cross-optimization differential testing. That is, GDSmith uses completely different DBMSs and versions while we focus on changes in settings on a single DBMS and version. Therefore it seems quite clear that the difference between the tested instances in GDSmith is greater than in our tests. This is a possible explanation for the different results, given that the assumption that greater difference means greater chance of different results holds true.

## 6.5 Non-empty results as a metric

A popular metric for evaluating a query generator when used for differential testing is *non-empty returns*. For example, GDSmith describes non-empty results as one of the primary challenges with random testing. It is also used as one of the key metrics for evaluating their tool [7]. The reasoning for using this metric is quite straightforward - empty results will never differ, which makes differential testing less effective.

However, we have chosen to not put much emphasis on this metric. It is important to remember that this metric is basically only interesting in differential testing. Many of the bugs found by our tests are error bugs. Finding such bugs do not benefit from the query generator aiming at non-empty returns, possibly quite the contrary.

We reason that it is actually a disadvantage to aim for non-empty results in many situations. Our tool can easily be modified and configured to only produce non-empty results. Always returning nodes and relationships matching a known pattern is simple. However, doing so would greatly decrease the expressiveness and language coverage of the generated queries. Therefore, losing variety is our main concern in focusing too much on non-empty returns.

Of course, producing only empty results makes differential testing useless, but , as suggested, focusing too much on non-empty results risks making the queries trivial and similar. Therefore, we have mainly been focusing on solutions that aim at increasing non-empty results without compromising language coverage and complexity. One such solution is making use of a graph schema when selecting labels. Instead of randomly picking a label name, that would most certainly result in no nodes or relationships matching that label, we pick a label that we know exists in the graph. This increases the chance of a non-empty result without losing complexity.

Finding the best sweet spot can be tricky and might depend on the graph. The configurability of the tool allows for tweaking the queries more or less towards non-empty results if needed.

## 6.6 Future Work

As presented in Section 4.3, the bugs we have found can often be expressed as very short queries, but we would often run quite large queries in order to cover as many scenarios as possible. This puts users of the test suite in a position where they manually have to reduce large queries in order to find the buggy part of the query. This process of manually distilling buggy parts of a query is often time consuming and repetitive, which indicates that it would be a good idea to automate the process. There is functionality in ScalaCheck called *shrinking* built for this purpose, i.e. automated test input minimization. Users can define their own shrinking methods that describe how the input data could be minimized and then ScalaCheck would automatically try and find the smallest input data that still triggers the error. Defining shrinking methods that automatically reduce the queries in this way would be a great way to improve the usefulness of the random testing tool. It would have the potential to significantly cut down on the most time consuming part of using the random testing suite.

Another potential improvement to our test suite would be to focus more on the differential testing. Since GDSmith presented way more logical bugs than us, it would be interesting to implement some of their ideas into our project. The query mutation is one such interesting concept, but it does not seem to be a widely used or tested technique as we have not found any other paper discussing the idea. Expanding our test suite to include other types of differential testing, such as cross-version or cross-DBMS, also has the potential to significantly improve the effectiveness of the differential testing. This is based on the idea mentioned above that we could reasonably assume that the bigger the difference between the tested instances, the bigger the chance of exposing an error. The downside of using more deviating implementations would be that while both might support the same query language, the supported syntax would in practice never overlap completely. This means that we would have to figure out a specific subset of the query language supported by both instances, for all pairs of instances that we would want to compare, which would not be an easy task. The upside in a scenario like this would be that when such a subset is defined, it would be very convenient to configure our query generator to express that sub-set. The tools configurability allows that specific language sub-set to be used only in that specific differential test, without reducing the capabilities of the query generator in general.

# Chapter 7
# Conclusions

In this project we have created an open-source tool for generating semantically valid Cypher queries. It uses a Context object that is being passed around during the query generation that allows it to correctly handle variables and other semantic details. The query generator covers a large part of the Cypher language syntax and is highly configurable to suit many situations. It is also implemented in a way that makes it easy to extend for future use with future versions of Cypher.

Furthermore, we have successfully used the tool for random testing of the Neo4j DBMS. This is partly done using differential testing of different runtime and optimization settings. During the project we found a total of 25 unique and confirmed bugs in multiple different parts of the Neo4j DBMS, i.e. related to parsing, semantic analysis, planning and runtime. The fact that the vast majority, 84%, of these bugs have already been fixed by the Neo4j developers indicates that these were meaningful errors in the Neo4j DBMS. 24 of the discovered bugs are error bugs and one is a logic bug. One possible reason for this quite surprising tilt against error bugs is the lack of cross-DBMS and cross-version testing performed in similar projects. However, the tool is perfectly suitable for this type of testing as well and it would be very interesting to see future work addressing this.

# References

[1] Neo4j. `https://neo4j.com`. Accessed: 2023-01-18.

[2] The neo4j cypher manual v5 - cypher manual. `https://neo4j.com/docs/cypher-manual/current/`. Accessed: 2023-01-09.

[3] Scalacheck. `https://scalacheck.org/index.html`. Accessed: 2023-01-25.

[4] Sqlsmith. `https://github.com/anse1/sqlsmith`. Accessed: 2023-01-09.

[5] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, sep 2000.

[6] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2123–2138. ACM, 2018.

[7] Wei Lin, Ziyue Hua, Luyao Ren, Zongyang Li, Lu Zhang, and Tao Xie. Gdsmith: Detecting bugs in graph database engines. *arXiv preprint arXiv:2206.08530*, 2022.

[8] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[9] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1140–1152. ACM, 2020.

[10] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[11] Donald R. Slutz. Massive stochastic testing of SQL. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 618–622. Morgan Kaufmann, 1998.

[12] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, jun 2011.

[13] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. Finding bugs in gremlin-based graph database systems via randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 302–313, New York, NY, USA, 2022. Association for Computing Machinery.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-03-15

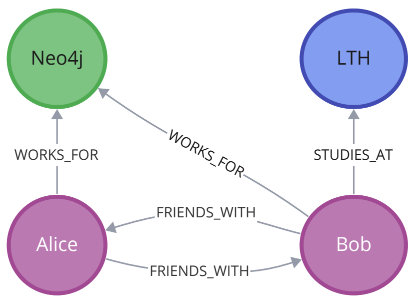
**EXAMENSARBETE** Random Generation of Semantically Valid Cypher Queries
**STUDENTER** Adam Forsberg, Andreas Lepik
**HANDLEDARE** Niklas Fors (LTH), Tobias Johansson (Neo4j)
**EXAMINATOR** Görel Hedin (LTH)

# Hitta buggar i grafdatabaser med slumpmässiga men korrekta frågor

POPULÄRVETENSKAPLIG SAMMANFATTNING **Adam Forsberg, Andreas Lepik**

I takt med att användningen av grafdatabaser i moderna applikationer ökar blir det allt viktigare att de fungerar korrekt. Vi presenterar ett program som använder generering av semantiskt korrekta Cypher frågor för att hitta buggar i Neo4js grafdatabas.

Grafdatabaser är en typ av databas som ökat i popularitet på senare år. Till skillnad från relationsdatabaser representerar dessa data som grafer med noder och relationer. Dessa grafdatabaser använder sig ofta av specialiserade språk för att kommunicera med databasen. Neo4j är marknadsledande inom grafdatabaser och har dessutom skapat och utvecklat frågespråket Cypher som används av flera andra grafdatabaser. Ett enkelt exempel på hur en grafdatabas och en Cypher-fråga kan se ut visas nedan.



```
MATCH (p:Person)-[:WORKS_FOR]->(:Company)
RETURN p
```

I vårt examensarbete har vi skapat ett verktyg för att slumpmässigt generera den här typen av korrekta Cypher frågor. Detta kräver att verktyget, förutom korrekt syntax, är medvetet om tillgängliga variabler och liknande semantisk information.

Verktyget är byggt för att vara väldigt konfigurerbart. Detta innebär att användare har möjlighet att styra slumpmässigheten och öka eller minska sannolikheten att olika delar av språket förekommer i de genererade frågorna. Detta bidrar till verktyget har många potentiella användningsområden.

I projektet har vi använt verktyget för testning av Neo4js grafdatabas. Detta har gjort genom att bl.a. skicka samma fråga till två ekvivalenta, men något olika, implementationer av Neo4js grafdatabas. Om resultaten skiljer sig har en bugg påträffats. Testningen resulterade i 25 unika bekräftade buggar inom flera områden. 21 av dessa buggar har redan åtgärdats, vilket antyder att buggarna bedömts vara meningsfulla.

En viktig egenskap hos verktyget är att det inte är begränsat till Neo4j, utan kan användas av alla databaser och applikationer som stödjer Cypher. Tack vare detta tillsammans med verktygets konfigurerbarhet och våra resultat hoppas vi att verktyget kommer fortsätta användas och utvecklas framöver.