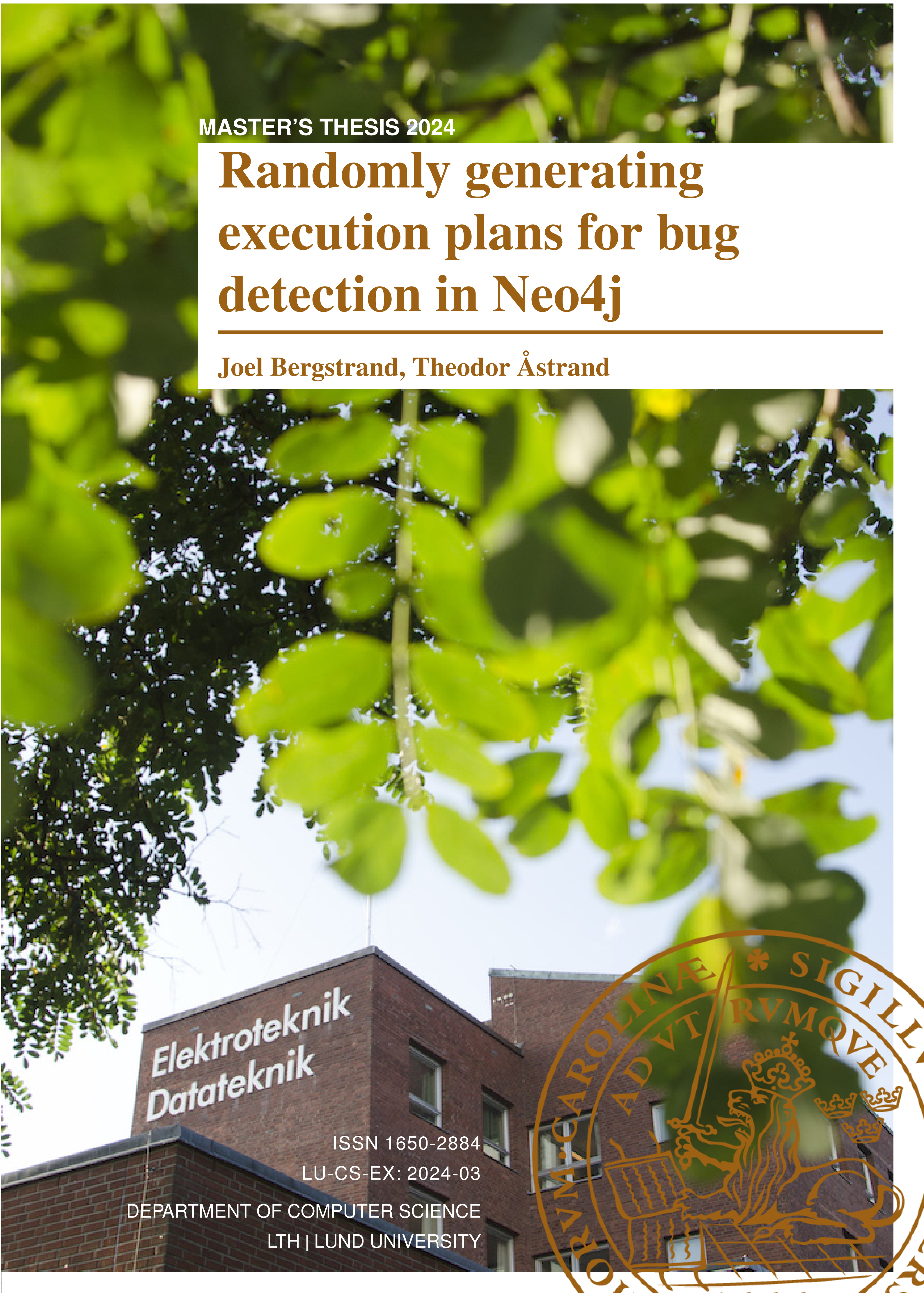

MASTER'S THESIS 2024

Randomly generating execution plans for bug detection in Neo4j

Joel Bergstrand, Theodor Åstrand

ISSN 1650-2884
LU-CS-EX: 2024-03

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-03

Randomly generating execution plans for bug detection in Neo4j

Slumpmässig generering av exekveringsplaner för felidentifiering i Neo4j

Joel Bergstrand, Theodor Åstrand

Randomly generating execution plans for bug detection in Neo4j

Joel Bergstrand
jo4082be-s@student.lu.se

Theodor Åstrand
th3522as-s@student.lu.se

January 22, 2024

Master's thesis work carried out at Neo4j Inc.

Supervisors: Love Leifland, love.leifland@neo4j.com
Niklas Fors, niklas.fors@cs.lth.se

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

Abstract

In recent years, Graph Database Management Systems(GDBMS) has increased in popularity for many use cases. One of the most popular GDBMS is Neo4j, which uses Cypher as a query language. With the increasing use of GDBMS in many business-critical applications, the need to test Neo4j and its competitors has become critical. One common practice for identifying bugs in a database system is using randomly generated tests, known as fuzz testing. Previously, this has been done by randomly generating queries, and several tools are currently available for this purpose.

When executing a Cypher query, the query goes through several processing steps to ensure a correct result returns quickly. One of the intermediate structures used in the query processing is the execution plan, which details how the runtime should solve the query.

In this thesis, we propose a novel approach to fuzz testing GDBMS by randomly generating execution plans. Our tool utilizes differential testing between different Neo4j runtimes, which allows for identifying incorrect results returned from one or more of the runtimes. These types of bugs are known as logic bugs. We can also identify situations when the Neo4j runtimes throw unexpected exceptions. The testing suite identified 20 bugs within the Neo4j, of which 11 were logic bugs. This approach to fuzz testing has proven helpful in identifying errors within the Neo4j runtimes, which previously received insufficient coverage by fuzz testing using queries. Other database management systems that utilize execution plans can benefit from the approach proposed by this thesis. The main drawbacks of this new approach are that it is not easily portable between different GDBMS and requires access to the query processing source code.

Keywords: graph database, fuzz testing, Cypher, execution plan, Neo4j

Acknowledgements

We want to thank Love Leifland for all the time he spent explaining the inner workings of Neo4j and supporting us throughout the work with our thesis. Additionally, we want to thank everyone at Neo4j who has provided us with insights and perspective. We also want to thank our supervisor at LTH, Niklas Fors, for his encouragement, supportive and helpful comments, and guidance throughout the process of writing this thesis.

Contents

Chapter 1
Introduction

1.1 Context

Database management systems (DBMS) are an essential part of any organization. Having a simple way to store and access data improves efficiency and decreases the complexity of many tasks. Relational databases have been the dominant solution for these purposes, but in recent years, graph databases have emerged as a more viable option for many use cases [2]. They are finding their applications in fraud detection, logistics, recommendation engines, and many other domains. Graph database management systems (GDBMS) store their data as a graph with nodes and relationships instead of several tables connected by primary and foreign keys linking them. Neo4j is one of these options and uses the query language Cypher. Several other graph databases also use Cypher, or versions of it, as their query language. Cypher can simplify some queries that would be complicated to construct in a language like SQL, and a GDBMS can compute them more efficiently [8]. The simplicity of Cypher is especially apparent when finding paths within the data. The two query languages contain some similarities in how their queries are processed [4] but differ in syntax. One of these similarities is that they use an intermediate structure during the processing before they can be executed, called an execution plan.

Software products are always flawed in their implementation, especially products under continuous development. Limiting the number of bugs is essential to the integrity of the software, both from a usability and security perspective. However, the challenge lies in finding bugs, which is a complex and time-consuming process. Finding bugs before they reach the consumer or a malicious actor that could exploit the bug is preferable. A wide variety of tools and practices for bug identification have materialized.

One of these practices is fuzz testing, which involves randomly generating input for the program to create many random tests. Through fuzz testing, it is possible to discover bugs and errors missed through manual testing. Furthermore, continuously utilizing random tests saves time as long as they are kept up to date and incorporate new features in the software.

Fuzz testing has been successfully used in many different scenarios, including relational DBMS [10] [11] as well as GDBMS [5] [3] [13]. In these examples, the fuzz testers create test cases by randomly generating valid queries as input to the databases. By randomly generating queries, the fuzz tester can produce many valid inputs to a database, providing good test coverage of the system. Furthermore, queries can be used interchangeably across different DBMS that use the same query language or with minor tweaks for different dialects of the same language.

Of particular interest for researchers in fuzz testing is finding bugs that return an incorrect value, commonly called logic bugs. Logic bugs are challenging to identify in a production environment as they require knowledge of the expected result of a query before running it on the GDBMS. Thus, unlike unexpected exceptions(error bugs), these are not commonly reported, which will crash the program. A logic bug will continue running generally without notifying the user and could corrupt their data. In recent research on using fuzz testing on Neo4j, only a few identified bugs have been logic bugs, even in papers with the explicit goal of finding them [5] [3].

1.2 Problem statement

Specifically for Neo4j, generating queries for fuzz testing has yet to prove to be as effective in identifying logic bugs as they have been at finding error bugs. One possible explanation for this limitation is that queries are ineffective in identifying the underlying issues that lead to logic bugs, as numerous intermediary steps exist between the query string and the possible root causes of logic bugs. Could this result from a lower likelihood of thoroughly investigating the later stages of query processing while extensively searching query syntax and parsing?

We believe there would be a potential advantage to test later in the query processing, in our case, randomly generating execution plans as the input, in fuzz testing of GDBMS. One benefit of using execution plans instead of queries is decreasing the space where we could find bugs. We cannot detect bugs in a query's syntax, parsing, or planning, limiting the search to the runtimes. One drawback of our approach is that the test cases we will generate might not correspond to an actual execution plan that the current implementation of Neo4j could plan. Thus, we will investigate the random generation of execution plans in Neo4j for use in fuzz testing of the GDBMS.

In this thesis, we will answer the following questions.

- How can we randomly generate semantically correct execution plans in Neo4j, and can these be used efficiently for fuzz testing?
- How does execution plan generation compare to query generation in the case of the identification of bugs, especially logic ones?

1.3 Contribution

During this project we have developed a testing suite capable of performing fuzz testing on logic execution plans in the Neo4j graph database. By testing directly on the execution plan layer, we have shown that a group of previously challenging bugs past the planning stage of the query processing within Neo4j are possible to identify. During our testing, we identified 20 bugs of varying importance in the Neo4j runtimes. The approach shows promise in effectively identifying logic bugs as they seem more likely to occur in the runtimes compared to syntactic or semantic bugs found higher up in the architecture. Our specific implementation is not portable to other databases. However, the practice of testing on intermediate structures applies to all Cypher-based databases and relational databases, such as SQL-based databases that use execution plans.

1.4 Contribution statement

During the development of the testing suite, we have pair-programmed all significant parts of the code, ensuring that both have complete knowledge of how the solution works. Pair programming has been an important method, allowing us to solve problems and discuss improvements with our supervisors. Writing the thesis has worked similarly, with one writing a section and then the other rewrites and improving on the initial text.

1.5 Outline

The structure of the thesis is as follows. In Chapter 2, we discuss the background needed to understand the project, particularly the Neo4j GDBMS, Cypher query processing, and the testing practices and tooling used in this thesis. Chapter 3 presents our approach to implementing differential testing using logic execution plans. Chapter 4 describes our process of generating valid logic execution plans, desired characteristics, and limitations. In Chapter 5, we present the test results and discuss their reasons. Chapter 6 discusses related works, focusing on other fuzz testers used on Neo4j that implement query generation and how they compare to our approach. Chapter 7 presents this thesis's conclusions and future work within the research area of execution plan generation.

Chapter 2

Background

2.1 Neo4j and Graph DBMS

What sets a graph database apart from relational databases are a few key distinctions. Most notably, instead of organizing data into multiple tables, graph databases store information within entities that collectively form a graph structure. These entities can represent nodes or relationships, connecting two nodes within the graph [1]. Nodes can be categorized using labels, and relationships always have a type and a direction. Furthermore, entities can contain properties, enabling the storage of data associated with them, which can later be used for computation or retrieved from the database. These properties are a combination of a property key and a corresponding value. The structure described here is specifically for the GDBMS Neo4j. For other GDBMS, the structure may vary, but the general outline is the same.

In Figure 2.1, we illustrate a simple graph created in Neo4j consisting of three nodes and their interconnecting relationships. The two blue nodes have the same label `Person`, and they both have a relationship of the type `FRIENDS_WITH` pointing at the other person. Having the same label does not imply that the nodes need the same properties. We can see Bob has one more property than Alice, which represents his occupation. Bob also has another relationship pointing at the third node. This one has the type `WORKS_AT` and a property representing the year he started. This third node has a different label from the other nodes to represent that it is a company. It also contains some differing properties. Note that the names *Alice*, *Bob*, and *Neo4j* written inside the nodes are only to make the representation more straightforward to understand and not stored in a Neo4j database.

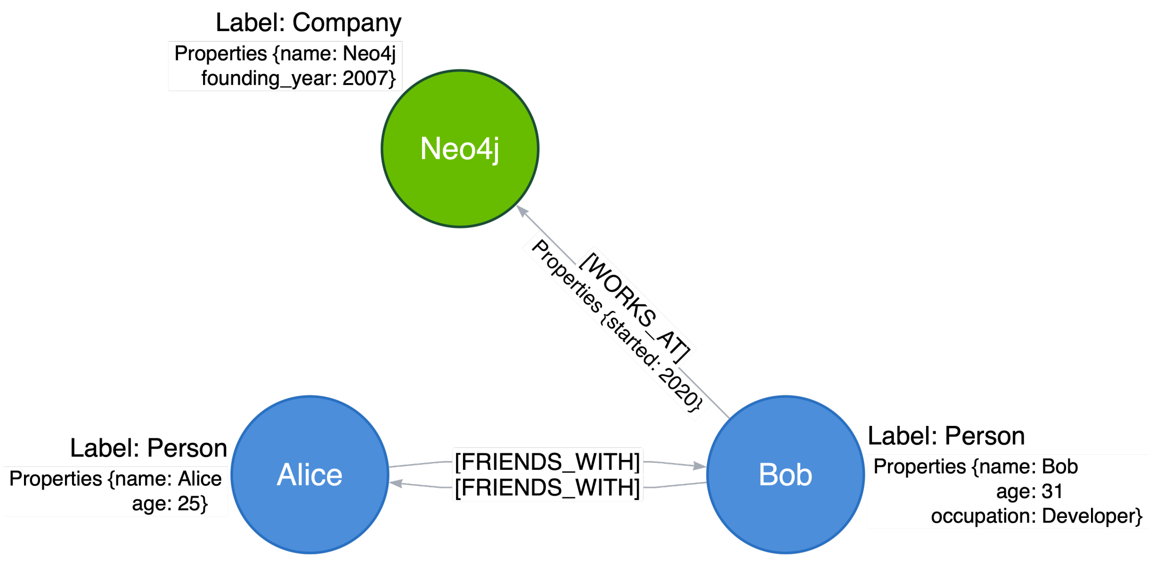

Figure 2.1: An example of a simple graph showing two nodes with the label Person, one node with the label Company, two relationships with type `FRIENDS_WITH` and one relationship with type `WORKS_AT`. All nodes and the `WORKS_AT` relationship have properties.

```
MATCH (:Person{name:Alice})-[:FRIENDS_WITH]->(p)-[:WORKS_AT]->(:Company
   {name:Neo4j})
RETURN p.name
```

Figure 2.2: An example of a Cypher query

2.2 Cypher

Neo4j has also developed Cypher as a dedicated query language for interacting with the Neo4j GDBMS. Notably, Cypher is not confined to its product alone, as several other graph databases across the industry have adopted dialects of the language. One of the primary advantages of Cypher lies in the simplicity and effectiveness of specific queries [2]. As all data is in one graph, it eliminates using a `JOIN` operator, which is necessary to query a relational database if accessing more than one table. Given the structure of a graph database, certain operations and computations can be executed efficiently [8]. The syntax focuses on common patterns in graphs, visually modeling the graph patterns when writing the query by representing nodes with parenthesis and relationships with arrows. Figure 2.2 shows an example of a Cypher query that incorporates labels and properties. In this query, we can see how the nodes Alice, Neo4j, and the one we are looking for, the variable `p`, are represented within parenthesis and the relationships between them with arrows with their type encased in brackets. This query will find every person in the graph named Alice and then the names of all their friends who work at Neo4j. In this case, the result would be only Bob.

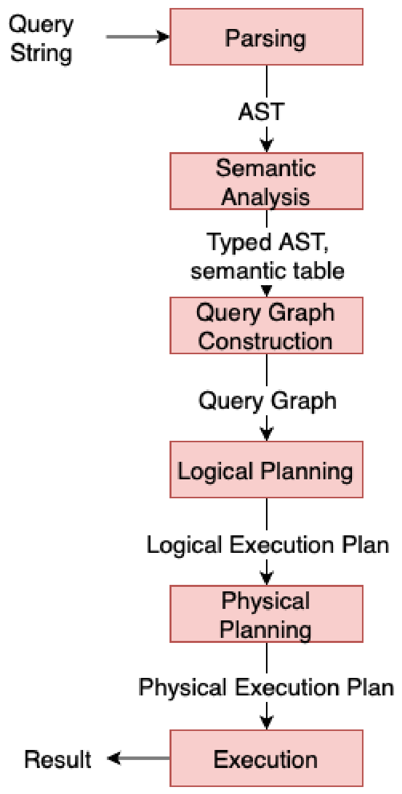

Figure 2.3: The stages of processing a Cypher query in the Neo4j GDBMS and the output at each stage.

2.3 Cypher Query Processing

To execute a Cypher query, Neo4j must first process the incoming query string. The process is done in several steps to ensure efficient and correct query execution. In Neo4j, the query processing is a six-step process from a given query string to a final result presented to the user, as shown in figure 2.3.

The first step is parsing the query into an Abstract Syntax Tree (AST) representing the query. This AST is then put through semantic analysis to check the variable types and scope of variables and optimizations such as naming anonymous pattern nodes, expanding aliases, and folding constants. After rewriting the AST, the next step is to construct another intermediate structure known as the query graph, an abstract high-level representation of the query, allowing for cost computation and performance optimization to a more significant degree than the AST can support. From the query graph, it is possible to generate multiple candidate logical execution plans through a step-by-step process in a bottom-up approach. By leveraging known statistics about the graph, the planner can now estimate and pick an efficient plan for executing out of these candidates. Neo4j uses several different runtimes that determine how to handle data during execution. This step also allows for several optimizations. During the next phase, the logical execution plan transforms into a physical execution plan corresponding to the chosen runtime. This physical execution plan can then be executed with the given runtime on a specific graph and will, if correct, generate a result. During the execution, the GDBMS uses variables containing rows of data.

2.3.1 Logical Execution Plans

Logical execution plans are binary trees composed of logical operators corresponding to the execution of a query [1]. The operators are responsible for data transformation in an execution plan. After the transformation or retrieval of data, the operator passes it to its parent. The operators can have different characteristics, such as the number of children (leaf operators, unary operators, and binary operators) if they are updating operators or eager operators. Neo4j has around 120 operators, which it uses when planning a logical execution plan.

Figure 2.4 was created by using the `EXPLAIN` keyword when running the query in figure 2.2 and shows the logical execution plan generated for that query. The plan is supposed to be read from the bottom up, looking at the columns Operator and Details.

- At Id 5 the operator `DirectedRelationshipTypeScan` will scan for every relationship in the database with the type `WORKS_AT`, giving the start node the variable name `p`.
- At Id 4, the operator is `Filter`, will only keep the relationships that point to a node that has the label `Company` and have the property `name` with a specific value. In the plan, this value is called a parameter and is represented by a $ before the name. In this case, the parameter `$autostring_1` has the value `Neo4j`.
- At Id 3, using the starting point of all the relationships we now have, the `Expand(All)` operator will find all relationships with the type `FRIENDS_WITH` that points to that same node.
- At Id 2, we again have a `Filter` operator. From the relationships we just found, we only keep those that originate from a node with a label `Person` and contain the property `name` of a with the same value as the parameter `$autostring_0`, which has the value `Alice`.
- At Id 1, the values of property `name` on the variable `p` is projected as a new variable with the name `p.name` using the `Projection` operator.
- Finally, the result is produced for the user using the `ProduceResults` operator.

```
>> EXPLAIN MATCH (:Person{name:Alice})-[:FRIENDS_WITH]->(p)-[:WORKS_AT]->(:Company{name:Neo4j})
   RETURN p.name


+-------------------------------+----+-------------------------------------------------+
| Operator                      | Id | Details                                         |
+-------------------------------+----+-------------------------------------------------+
| +ProduceResults               |  0 | `p.name`                                        |
| |                             +----+-------------------------------------------------+
| +Projection                   |  1 | p.name AS `p.name`                              |
| |                             +----+-------------------------------------------------+
| +Filter                       |  2 | (anon_0.name = $autostring_0 AND anon_0:Person) |
| |                             +----+-------------------------------------------------+
| +Expand(All)                  |  3 | (p)<-[anon_1:FRIENDS_WITH]-(anon_0)             |
| |                             +----+-------------------------------------------------+
| +Filter                       |  4 | anon_3:Company AND anon_3.name = $autostring_1  |
| |                             +----+-------------------------------------------------+
| +DirectedRelationshipTypeScan |  5 | (p)-[anon_2:WORKS_AT]->(anon_3)                 |
+-------------------------------+----+-------------------------------------------------+
```

Figure 2.4: The logical execution plan produced for the query.

Leaf Operators

Leaf operators are operators that do not have any children. According to some characteristics, most leaf operators fetch data directly from the database. In the plan shown in figure 2.4, there is one leaf operator, `DirectedRealtionshipTypeScan`. This operator will fetch all relationships and the nodes they connect that have the desired type. Another leaf operator is `AllNodesScan`. As the name suggests, this will return all nodes in the graph. Some leaf operators do not return entities but the number of entities such as `NodeCountFromCountStore`.

Unary Operators

Unary operators have one operator as their child passing data to them. These operators will transform before passing the data to their parent operator. The example, figure 2.4, contains a few of these, one being `Filter`, which will take the data from its child and pass the data that fulfills its predicate. Another unary operator in this example is `Expand`, which, given one or more nodes, will traverse all relationships connected to the node. There are also many variations of `Expand` to handle different types of traversals, such as when there are no matching relationships or when we want to traverse chains of relationships with a certain length. A final example of a unary operator is `Sort`, which will sort the incoming data in a given order.

Binary Operators

Similar to the unary operators, binary operators will apply a function to the data given by their children. Only the binary operators have two children. One example is the operator `Union`, which will concatenate the results from both of its children operators. Many of these operators are versions of the `Apply` operator. The standard format will take one row from the left-hand child at a time and use it as the argument for the right-hand child. The result will be the right-hand child for every row from the left-hand child.

2.3.2 Runtimes

The runtime decides how the logical execution plan should behave during execution, specifically how to handle data during execution [1]. There are currently three available runtimes within Neo4j: slotted, pipelined, and parallel. There is also a fourth, interpreted, which Neo4j maintains for internal use. There are several configurations available for the user to control how the runtimes transform the logical execution plan into a physical execution plan. The runtimes are all expected to provide the same results unless otherwise stated. The runtimes have different advantages and disadvantages and may vary in performance depending on the scenario.

Interpreted Runtime

The Interpreted runtime is now deprecated but still maintained runtime in Neo4j. It does not implement any of the optimizations that later introduced runtimes use. It is usually slower at solving queries than the more modern runtimes. It does not implement complicated optimizations, so it is less prone to errors.

Slotted Runtime

Slotted runtime produces a one-to-one mapping from the logical execution plan, where each logical operator maps to a corresponding physical operator and is processed individually starting at the bottom left-most operator. Each variable in the slotted runtime has a dedicated "slot" containing rows of data. This is where the data is stored intermediately during execution. The slotted runtime uses a pull-based process where each operator in the tree pulls rows of data from its child operator, causing data to move from the bottom to the top.

The Slotted runtime is an interpreted runtime, meaning it interprets the logical plan operator-by-operator. It has a comparatively short planning phase as it only needs to generate some of the code for the query before executing. On the other hand, it is generally slower than the newer runtimes as it is inefficient with its CPU usage.

Pipelined Runtime

The Pipelined runtime divides the logical execution plan into one or more pipelines, which allows more than one row of operators to execute at a time. The pipelines are a grouping of logical operators that can execute the same task. Each pipeline consumes a batch of rows, known as *morsels*, which buffers between the pipelines.

Unlike the Slotted runtime, the Pipelined runtime is a push-based execution model where data moves from child to parent. By storing the data in local variables, the data in Pipelined runtime allows for more efficient CPU usage. The Pipelined runtime can use either an interpreted or compiled runtime but uses the compiled variation as a default, referred to as with or without fusing. Compiled runtimes have a code generation phase, unlike the interpreted runtimes, which generally cause longer planning times but shorter executions.

Parallel Runtime

The Parallel runtime shares the same architecture as the Pipelined runtime but allows the parallelization of queries by allowing each pipeline task to execute on a separate thread. In general, the Parallel runtime produces more pipelines compared to the Pipelined runtime, as it is more efficient to have more tasks that run in parallel. When a pipeline receives a morsel, it can start computing the input if there is an available thread.

Several considerations need to be taken into account when using the parallel runtime. Firstly, running updating queries on the parallel runtime is currently not supported. Moreover, the order of the output is not guaranteed. The parallel runtime offers a significant speed-up on large graph-global queries. However, there is no single rule to determine if a query will run faster on the parallel runtime compared to the sequential variations.

Compiled Expressions

The setting "compiled expressions" controls how the runtime evaluates expressions in the query [1]. The default is that the runtime will use the compiled expression engine when needed, but it is also possible to force the runtime to use it always. Forcing the usage of the compiled expression engine should not alter the output.

2.4 Testing

2.4.1 Fuzz Testing

Fuzz testing is a technique of testing software through randomized correct input to the program. If the test is sufficiently random, the test will explore large parts of the program [6]. Running enough tests will probably result in the program crashing or finding differing results, thus finding a bug depending on what testing parameter the fuzz test uses to determine an incorrect response or the test's ground truth. There are several different ways to compare the performance of a fuzz tester, with some usual performance measures being the number of bugs found and the code coverage. A challenge with fuzz testing as an approach is identifying unique bugs, as it is usually possible for multiple generations to generate the same bug for different input variations.

2.4.2 Differential Testing

Establishing a ground truth to verify whether a program is running correctly is a foundational step in testing. Differential testing might be a helpful approach when it is difficult to establish a ground truth in a testing environment, exploiting the fact that if multiple programs are expected to solve the same problem and are deterministic, then they are expected to return the same result for the same input [9]. Differential testing cross-references different programs against each other. Thus, comparing results or behavior is utilized as the ground truth for establishing the correctness of the programs. Establishing which of the programs have returned the correct result can be difficult. The more programs in the comparison, the easier it will be to determine which are faulty.

2.4.3 ScalaCheck

ScalaCheck is a library used for property-based testing and automatic test data generation for Scala [7]. Generators generate test data in ScalaCheck, represented by the `org.scalacheck.Gen` class. You need to know how to use this class if you want ScalaCheck to generate data of types not supported by default.

This library allows for generating with conditions using the `suchThat` method, which will try to generate and afterward check if it conforms with the condition sent to the function. If the condition is not met, the generated data is discarded. A variation to this called `retryUntil` does the same thing but allows for the option to retry a set number of times. Two other central methods from this library are `oneOf` and `frequency`.

```scala
val numbers = Set(1, 2, 3, 4, 5, 6)

val generateNumber: Gen[Int] = for {
  number <- Gen.oneOf(numbers)
} yield number

val generateEvenNumber: Gen[Int] = for {
  number <- generateNumber.suchThat(_ % 2 == 0)
} yield number

val generateMostlyEven: Gen[Int] = for {
  number <- Gen.frequency(9 -> generateEvenNumber, 1 -> generateNumber)
} yield number


def printRandomNumbers(): Unit{
  val num1 = generateNumber(Gen.Parameters.default, Seed.random())
  val num2 = generateNumber(Gen.Parameters.default, Seed.random())
  println(s"First number: $num1")
  println(s"Second number, from the same generator: $num2")
}
```

Figure 2.5: An example of how generators can be created with ScalaCheck and used to generate numbers.

These will create generators that pick one option from a list of options. The first chooses at random, while the other has a set frequency of how often each option should be picked.

Figure 2.5 shows three examples of how we can create generators with the help of ScalaCheck. Note that these do not have the type `Int` but rather `Gen[Int]` and can be used to generate numbers. If called multiple times, the value will be random each time. The generator `generateEvenNumber` shows how `suchThat` can be used to generate with a condition, in this case, that the generated number should be even. Note, however, that `generateNumber` will only generate an even number half the time. When the number is not even, the generator `generateEvenNumber` will generate the value `None`. The last generator, `generateMostlyEven`, is an example of how the function `frequency` can weigh the options. In this case, the generator has a 90 percent chance of only being able to generate an even number (or None) and a 10 percent chance of generating a random number from the set.

Lastly, the figure shows an example of how a method can use these generators to generate random numbers. The method `printRandomNumbers` shows how one generator can be used multiple times to generate a number. Each time the generator is called, it will return a random value, `num1` and `num2` will not necessarily have the same value unless the seed is the same.

Chapter 3
Approach

Our method for identifying bugs relies on the idea that each runtime in Neo4j should return the same results when provided with the same query, which is the basis of our differential testing approach in our testing suite. As we have included the parallel runtime, we must consider that the result might be unordered when comparing the parallelized and sequential runtimes.

Algorithm 1 outlines the structure of our testing approach. Using a set of graphs, we create several execution plans for each graph. After execution on the different runtime configurations, we compare the results and can identify unexpected behavior. The reporting of bugs were done continuously during the development of the tool and the final product comprises over 2500 lines of code.

3.1 Test procedure

To enable the differential testing, we set up nine different runtimes configurations, as displayed in table 3.1. The interpreted runtime is our reference runtime, against which we compare all other configurations. All runtimes, except interpreted, have a configuration where we force the usage of the compiled expression engine. For the pipelined runtime, we run it with and without the option of fusing and with and without forcing compiled expression, giving us four total variations of the pipelined runtime.

After initializing our runtimes, we build the graph on which we will run tests. We give all the entities in the graph zero or more properties of the three types we have decided on: integers, strings, and boolean values. We track all entities and property labels and use them to construct the logical execution plan later. For our testing suite, we are using seven different graphs with varying complexity. Table 3.2 presents their configurations.

Using the information provided by the graphs, we can now generate a logical execution plan that incorporates relevant information from the plan. Each plan uses a specific seed that, combined with the graph, can recreate the test case.

Algorithm 1 An overview of our testing procedure.

```
Require: $G$ : Set of graphs
Require: $N_p$ : The number of plans to be generated for each graph
Require: $R$ : Set of runtimes
  for $g \in G$ do
      $propertySeed \leftarrow randomLong()$
      $addRandomProperties(g, propertySeed)$
      for $i = 1 \rightarrow N_p$ do
          $s \leftarrow randomSeed()$
          $logicalPlan \leftarrow generateLogicalPlan(s, g)$
          $referenceResult \leftarrow execute(interpreted, logicalPlan)$
          for $r \in R$ do
              $result \leftarrow execute(r, logicalPlan)$
              if $result$ did not complete then
                  $logBug(r, i, g)$                                  ▷ Error bug
              end if
              if $result \neq referenceResult$ then
                  $logBug(r, i, g)$                                  ▷ Logic bug
              end if
          end for
          $print(referenceResult, loggedBugs)$
      end for
  end for
```

Runtime	Compiled Expression	Fusing	Reference
Interpreted			✓
Slotted			
Slotted	✓		
Pipelined		✓	
Pipelined	✓	✓	
Pipelined			
Pipelined	✓		
Parallel		✓	
Parallel	✓	✓	

Table 3.1: Runtime configurations used in the test suite

The plan is then executed using the different runtime configurations, with all results stored and the results from the interpreted runtime marked as the reference. First, we check if all results completed execution without throwing an exception and within the time limit. If no problem occurs, the tool checks for differences between the results. To ensure ordering is no problem, we sort the results, including the lists that may be present in the results, and are then compared row by row to avoid false positives due to different ordering. The test is marked as passed if results do not differ and no exception occured.

When at least one runtime failed to execute, or a discrepancy is found in the result, the test will fail, and relevant information will sent to the log. In the case of a timeout or an arithmetic error, such as division by zero, the test will be ignored. When the test fails on an exception, the stack trace will be in the result.

When all the tests have completed, we manually check the failed test to see if the cause is differing results (logic bug) or an exception (error bug). If we have not previously reported the bug, or if determining its uniqueness is challenging, we save all the information concerning the test so that it can be recreated and reported later.

We cannot tell beforehand if a plan will terminate within a reasonable time frame. Because of this, we have implemented a time limit in the test suite, which will cancel the test if the test exceeds it. For most tests, this timeout has been one second per runtime configuration. The upper time limit for each test as follows $1s * nbrConfigurations + planGenerationTime$. Plans, including large expansions or path searches, are prone to quickly expand the number of rows, even on small plans, and are thus prone to cancellation. After all tests are done for a graph, we calculate some statistics. This includes the number of empty results, the number of canceled tests, and the average plan length.

3.2 Reproducing and ensuring uniqueness of bugs

When a test case is marked as failed, the test suite produces a reproducible case containing all necessary information. The test case can be run separately from the standard test suite to reproduce the error and confirm its validity. In some cases, we are manually able to simplify the plan slightly; this is done by removing one operator at a time from the plan until all operators that do not modify the result are removed.

Verifying the uniqueness of a bug is a process of discussion and comparison. With the error bugs, it is usually relatively easy to establish if the bug has already been recorded. It is usually more difficult with logic bugs as the plans can be pretty complex, and the root cause can be difficult to establish or appear in multiple operators due to underlying implementation. The final confirmation of a bug's uniqueness is impossible to establish until a fix for the bug in question is at hand. When rerunning the test case with the fix, it should now pass. Thus, after reporting, we keep track of the bug's progress.

3.3 Graph structure

The current test suite implementation uses seven different graphs as presented in table 3.2. The plans have different structures to broaden the possible test cases we can generate,

Graph	#Nodes	#Relationships
Bipartite with extra nodes	37	50
Empty	0	0
Two Nodes	2	0
Two Nodes and Relationship	2	1
Grid	25	40
Lollipop	3	3
Nested Star	40	39

Table 3.2: The graph configurations used in the test suite.

as the graph impacts which plans we can generate. Some graphs have simple structures, such as an empty graph, one with two nodes without a relationship, and one with a relationship connecting them. The more complex graphs were chosen from Neo4j's library of test graphs, including a bipartite graph, a grid, and others, providing the test suite with more complex graphs to run tests. Having graphs that were easy to create would simplify the process of recreating failing test cases and make it easier for developers at Neo4j to confirm bugs. The methods for creating all these graphs, except for bipartite with extra nodes, can be found in Neo4j's community edition [1].

When generating the properties for a graph, a random selection of the available properties was made and given to the entities. These properties were picked from three sets that we created. The sets contained properties of boolean, string, and integer types. The random generation of properties was also seeded to support recreating them if needed.

[1]https://github.com/neo4j/neo4j/blob/5.13/community/cypher/runtime-spec-suite/src/test/scala/org/neo4j/cypher/internal/runtime/spec/GraphCreation.scala

Chapter 4
Generation of Logical Execution Plans

For the random generation of plans needed to perform the fuzz testing, we have implemented a logical execution plan generator using ScalaCheck. This chapter describes the generated plans' general structure and an overview of the process.

4.1 Structure and desired characteristics

We strive to construct semantically correct plans that include interesting features. Over many tests, the variation of plans explored should be extensive. There are multiple interesting features that we are looking for in our plans. One factor is the interconnectedness and awareness of the plan by the operators. What we mean by this is that an operator should use previously generated variables, especially those generated close to the operator in question. This increases data dependencies within the plan, particularly when generating expressions incorporating variables. Furthermore, the generation should be able to generate any possible combination of operators over a large number of tests.

All of the plans we generate will be semantically correct and possible to execute, but we cannot verify if the generated plan will terminate within a reasonable time. Non-terminating plans are a problem as they are not possible to test. To counteract this, we rely on the number of operators in the plan to act as a rough limitation of the plan's complexity. The number of operators in a logical execution plan does not necessarily reflect the computational runtime. Nevertheless, limiting the number of operators will decrease the number of plans that pass the time limit.

4.2 Generation process

The first step of generation is the preprocessing of the graph information. We collect label names and relationship types and gather IDs of entities, as some of the operators we

	Implemented	Not in scope	Total	Lines of code
Leaf	17	33	50	212
Unary	23	22	45	370
Binary	14	4	19	138
Total	54	58	114	720

Table 4.1: Number of implemented operators per category as well as how many source lines of code their implementations consist of. The operators that are not within the scope of the testing suite are updating operators and operators leveraging indexes within the database. The missing binary operator is `Repeat(Trail)` due to complicated implementation.

implement require these as input. Thus, different graphs will cause different generations of plans as the available graph data differs. After the preprocessing step, we can start the generation process.

At the top of each logical execution plan is the operator `ProduceResult`, a unary operator, meaning we must generate a child operator. Figure 4.1 shows the function called every time an operator needs a child and will be called recursively by all unary and binary operators. When the plan reaches the operator limit, the `planGen` function will always assign leaf operators to complete the plan. The figure simplifies how the function works in the actual generator. In reality, there is the option to weigh the choice of operator type and specific operators, allowing us to tune the likelihood of each operator type depending on which has been picked. The three different methods picked in figure 4.2 only show one option per category, and table 4.1 shows the actual number of options within each category.

As all sub-trees of a plan are valid plans, the general idea is to pick an operator and then generate the children that fulfill the operator's requirements. When reaching the size limit and any operators still have unevaluated children, the plan will be completed by only assigning leaf operators. The ScalaCheck library allows us to put requirements on children, causing it to retry the generation if the requirements are unmet. We allow for 1000 attempts to generate the correct plan on the top level that meets the minimum requirement of operators. When the generation is complete, the plan is set as the child of the `ProduceResult` operator.

4.2.1 PlanState

During the plan generation phase, we utilize a `PlanState` object to ensure the correctness and that the plan conforms to our desired outcome at any given point during the generation of a plan. An overview of the `PlanState` object is presented in table 4.2. The attributes marked with *Control* are used when choosing the operator type when generating a child and are updated continuously during generation. The rest ensures the plan is correctly assembled and will run when executed. These values update during the generation of each operator. When we create each operator, the `opsCount` is incremented and given an id by `idGen`.

The `opsCount` value ensures the desired structure of our plans as shown in figure 4.1.

```scala
def planGen(planState: PlanState): Gen[LogicalPlan] = {
  if (planState.opsCount >= MAX_COUNT) {
    leafPlan(planState)
  } else {
    Gen.oneOf(
      leafPlan(planState.incOpsCount()),
      unaryPlan(planState.incOpsCount()),
      binaryPlan(planState.incOpsCount())
    )
  }
}
```

Figure 4.1: The function `planGen` from our generator. This method manages the number of operators in the plan and randomly picks the following type of operator if the limit has not yet been reached.

```scala
def leafPlan(planState: PlanState): Gen[LogicalPlan] = {
  Gen.oneOf(
    allNodesScan(planState),
    ...
  )
}
def unaryPlan(planState: PlanState): Gen[LogicalPlan] = {
  Gen.oneOf(
    projection(planState),
    ...
  )
}
def binaryPlan(planState: PlanState): Gen[LogicalPlan] = {
  Gen.oneOf(
    cartesianProduct(planState),
    ...
  )
}
```

Figure 4.2: The methods used when picking an operator within the three categories. These have been scaled down and only shows one option for each operator type.

Name	Type	Use	Correctness	Control
opsCount	Int	Number of operator generations in the plan		✓
idGen	IdGen	Generator for operator ids	✓	
varCount	Int	Tracks number of used variables	✓	
parameters	Set[Parameter]	Tracks used parameters including name and type expected	✓	
variables	Set[LogicalVariable]	Tracks the currently available variables	✓	
semanticTable	SemanticTable	Tracks types of variables in the plan	✓	
config	LogicalPlanConfig	Contains current frequencies for plan generation		✓

Table 4.2: The values stored in the PlanState object.

Using it, we can roughly control how many operators make up each plan. What we omit from this figure is how `config` can affect the choices made by the generator by controlling the frequency of individual operators. This may be updated statically before the generation as well as during the generation. For example, if we chose a binary operator, the likelihood of picking another afterward is lowered.

During generation, we create new variables used by the plan. The type of each variable kept track of in the `SemanticTable`. The variables are named sequentially, starting at zero. The set `variables` tracks which variables should be in scope during different stages of the plan. When generating expressions, we can use parameters containing random values. When we generate a parameter, it is tracked, together with their respective types, in the set `parameters` in the PlanState. Before executing the plan, we generate random values for the parameters with the expected types.

4.2.2 Operator implementations

All operator implementations follow the same pattern. They get a `PlanState` from their parent operator, and in the case of unary and binary operators, they call `planGen` once for each child. When doing this, some operators will have additional requirements on the plan succeeding it. For example, if we want to create a `ShortestPath` operator, the child must have two available nodes to calculate the path between them. The ScalaCheck library has a function called `suchThat`[7] that requires the child plan to contain specific characteristics. If the generated plan does not meet the requirements, ScalaCheck will discard the plan. In the case of operators with strict requirements on the contents of their children, it will be less likely to be planned. After the generation of the children, we

```
1 // Generated plan
2 .produceResults("var0", "var1", "var2")
3 .sort("var0 ASC", "var2 DESC")
4 .cartesianProduct()
5 .|.nodeCountFromCountStore("var2", Seq(Some("A"), Some("B"))
6 .projection("$param1 + 4 = 10 AS var1")
7 .allNodeScan("var0")
```

Figure 4.3: An example of a generated plan, generated with the data from the default graph. The left and right-hand side of a binary operator is separated with the symbol |. The plan generated three variables `var0`, `var1` and `var2` as well as a parameter `$param1`.

```
1 // Leaf Operator
2 def allNodesScan(planState: PlanState): Gen[AllNodesScan] = for {
3   variable <- newVariable(planState)
4   planState <- Gen.const(planState.newNode(variable))
5 } yield {
6   AllNodesScan(variable, planState.arguments)(planState.idGen)
7 }
```

Figure 4.4: The implementation of the operator `AllNodeScan`, an example of a leaf operator.

create the necessary variables and expressions before returning the complete operator. Figure 4.3 shows an example of a plan generated using our tool.

Figure 4.4 shows the implementation of the leaf operator `AllNodesScan`. This operator will return all nodes contained in the graph. Line 3 in this figure shows how a variable is created, and on line 4, we update the `planState` to contain this variable, and its content is of type Node. When the operator is created and returned on line 7, this variable is given as an input. In figure 4.3 on line 6, we can see how a `AllNodeScan` operator might look in a finished plan, `var0` is the variable that is created to store the resulting rows.

To give an example of the implementation of a unary operator, figure 4.5 shows the implementation of the operator `Projection`. This operator evaluates a particular expression and saves the result as a new variable. Since this is a unary operator, the first step is to call the `planGen` function to generate a child, as seen in line 4. After this, we generate an expression using the function `validExpression` that ensures the expression is valid within the current state. The method has a set of available symbols, the variables generated by the child plan. Like the previous operator, `Projection` needs to create a variable to store its results. However, we do not know the type of this variable. In line 7, the variable gets a type using a method that evaluates the type of the expression and gives the variable the same type. When creating the finished operator on line 10, the child is given as a parameter, connecting the operators. In figure 4.3 on line 5, we see an example of a `Projection` operator in a finished plan. In this case, the generated expression is `$param1 + 4 = 10`, and `var1` is the variable created to store it. Where `$param1`

```
// Unary Operator
def projection(planState: PlanState): Gen[Projection] = for {
  planState <- Gen.const(planState.updateConfig(new OneChildConfig))
  child <- planGen(planState)
  expr <- validExpression(child.availableSymbols.toSeq, planState, _.
    expression)
  variable <- newVariable(planState)
  planState <- Gen.const(planState.declareTypeAs(variable,
    getExpressionType(expr, planState)))
} yield {
  val map: Map[LogicalVariable, Expression] = Map(variable -> expr)
  Projection(child, map)(planState.idGen)
}

```

Figure 4.5: The implementation of `Projection`, an example of an unary operator.

```
// Binary Operator
def cartesianProduct(planState: PlanState): Gen[CartesianProduct] = for
    {
  left <- planGen(planState)
  right <- planGen(planState)
} yield {
  CartesianProduct(left, right)(planState.idGen)
}
```

Figure 4.6: The implementation of `Cartesian Product`, an example of a binary operator.

is a generated parameter, it is assigned a numeric type since it is part of an arithmetic expression.

The final type of operator to implement is the binary operator. Figure 4.6 shows the implementation of the operator `CartesianProduct`, which combines the rows from its two children. As with all binary operators, it generates its left child first and then the right-hand child second. Figure 4.3 provides an example of how this operator can be generated, as shown in line 3. In this example, the operator has the left-hand child `Projection` and on the right-hand side `NodeCountFromCountStore`. In the `ProduceResult` operator at the top, we can see how it contains the variables created by both left and right-hand children, `var0` and `var1` are created on the left-hand side of the `CartesianProduct` while `var2` is created on its right-hand side.

Some of the binary operators will modify the `PlanState` in between the generation of its children. In some cases, the need to do so is that the right-hand child should not be able to access all variables generated on the left-hand side. The need to update the PlanState is the case for the many different versions of the `Apply` operator. Binary operators may also generate variables and expressions when needed.

These were only one example of each operator type, but they all follow the same general structure. The complexity of the implementations of these operators varies wildly, and in general, the more complex the operators need, the more lines of code are needed for their implementations. Table 4.1 shows how many operators we implemented of each type and how many lines of code were written to implement them.

4.2.3 Expression generation

Many of the operators make use of expressions to transform the incoming rows. Neo4j supports a large number of different expressions that perform a variety of different tasks. Expression can be mathematical operations such as simple arithmetic, calculating the average from a list of values, calculating distance or duration, or predicate functions. An expression can incorporate properties from entities or parameters provided by the user. An expression can also include conditional statements and even sub-queries encompassing a complete Cypher query. Due to the large number of possible functions in expression and the implications of the combinations, we only implement a small subset of the possible expressions. Figure 4.3 shows an expression generated for the `Projection` operator. It is a predicate expression that will return `true` if the parameter `$param1` has the value 6.

The operators expect different expressions, some with specific requirements, while others can run with almost any combination of transformations. This requires us to tailor the expression generation to the specific operators to ensure valid and reasonable plans. The generators are heavily weighted towards desired behaviors, such as using available variables produced previously in the plan and utilizing properties available on the variables. These are kept track of in the `PlanState`. Importantly, we track the typing of variables in the `SemanticTable`. Without this weighting, the likelihood of the expression generator producing expressions showing these characteristics would be negligible.

As we want the logical execution plan to be interconnected, we need to type the results of the expression if possible. For this, we use the Neo4j semantic checking, which is usually run multiple times during the planning stage before the logical execution plan is generated. We can construct the necessary structures from the values we keep track of during generation. If an expression fails the semantic checking, a new one will be generated, retrying one hundred times to create a valid expression.

4.3 Generation limitations

We have made some limitations to the generation in the current version of the logical execution plan generator. We are currently not implementing any updating queries. This would require reworking the graph handling framework that implements rolling back changes to the graph after each test to keep the graph state coherent. Allowing for updating queries would also disallow us from testing the parallel runtime, as it currently does not allow updating queries. Furthermore, we are not handling any queries involving the indexing of values in the graph. Indexing would be a potential future expansion of the test suite. Finally, we are forcing ordering for some operators to ensure that the result from the sequential runtimes is the same as the result from the parallel runtime. This is done by sorting all variables before any operator relying on the order of the incoming data, such as `Top` and `Limit`.

Chapter 5
Evaluation

In this chapter, we evaluate the random generation of execution plans. The first section contains an evaluation and presentation of how the testing suite behaves and performance under different configurations. In the second section, the reported bugs are categorized and presented. In the third section, we highlight two examples of the bugs we have identified. In the last two sections, we further discuss the tool's usefulness and the generation's limitations.

5.1 Testing suite behavior

To better understand how the random generation works, we did several tests with different configurations on the limits of the plan size. The results are presented in table 5.1, giving us insight into how the results change depending on the size of the generated plans. The values resulting from generating 1000 plans on the bipartite with extra nodes graph on the same version of the tool. *Avg Ops* is the average number of operators present per plan. Observe that the number of operators in a plan can be larger than the maximum number due to the need of of adding additional operators if the plan is not complete by the point when the limit is reached. Unevaluated right-hand sides of binary operators are commonly what causes the need for additional opertors. The *Empty* column contains the number of tests without any resulting values. The *Canceled* column contains the number of plans timed out within the one-second timeout or included a valid error, e.g., division by zero. *Non-Comp.* is the sum of empty and canceled values showing the number of non-comparable tests. In the table, we can see that increasing the lower bound of the generation will increase the time it takes to generate a plan within the limitations. An explanation is that ScalaCheck has to retry the generations multiple times before returning a valid plan. Figure 5.1 presents the distribution of the number of operators in plans.

When observing the average length of the plans, the configuration with 10 as the upper limit can produce larger plans on average than the one with 25 operators as the upper limit. We also see that larger plans have a higher chance of not providing any rows in the result and causing a timeout. It is a balance between testing more complex structures but with a higher percentage of the plans being non-comparable and taking more time to generate. With shorter plans, we can generate more comparable plans in a shorter time, while the plans are less complex overall.

Min Ops	*Max Ops*	Time	Avg Ops	Empty	Canceled	Non Comp.
0	10	5m 59s	5.114	258	11	269
2	10	8m 31s	8.268	369	24	393
4	10	14m 26s	11.238	408	42	450
6	10	18m 57s	13.645	478	68	546
8	10	22m 6s	15.115	504	46	550
0	25	7m 23s	4.487	236	16	252
2	25	7m 40s	6.372	331	14	345
4	25	16m 57s	9.422	393	34	427
6	25	28m 59s	12.266	415	56	471
8	25	50m 5s	15.271	467	61	528

Table 5.1: The table shows how the test suite is impacted by adjusting the graph limiting values of *Min* and *Max* number of operators.

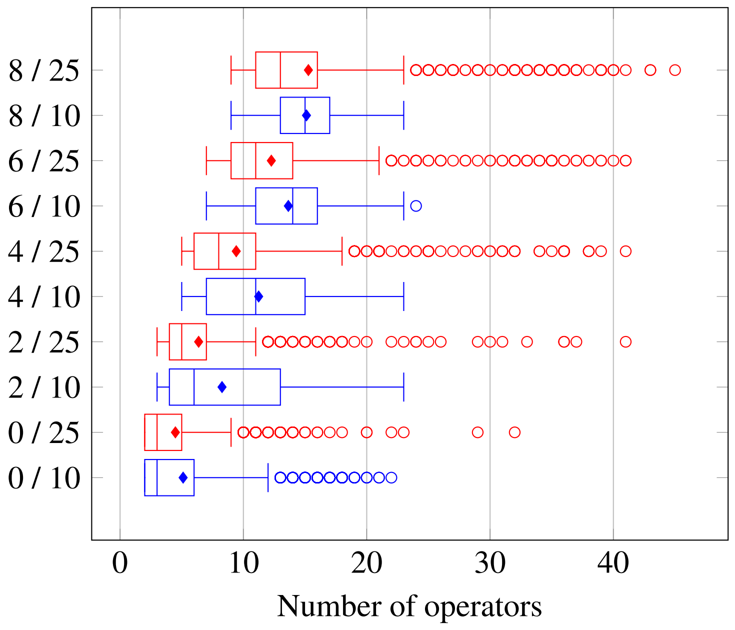

Figure 5.1: A boxplot showing how the number of operators per plan change depending on the generation minimum and maximum values. The diamond indicates the average number of operators. The values are from the data used in table 5.1

.

In figure 5.2, the distribution of generated operators is presented. Operators `Sort` and `ProduceResult` are excluded due to always being present or planned together with other operators due to the need for persevering ordering. It is interesting to see the distribution of the unary operators, with the most prevalent being the operators without any prerequisites, such as `Distinct`, `Optional`, and `Filter`. Note that `Expand` and its variations, as well as `Unwind`, are much less commonly planned due to requiring the existence of either nodes or lists present in the sub-plan of the child. The same pattern appears within the binary operators as `NodeHashJoin` and `Union` are rarely planned due to complicated requirements on their children.

5.2 Found Bugs

The random logical execution plan proved able to identify several bugs across the nine Neo4j runtime configurations used in the testing suite. The results are from tests that we ran during the development of the tool, as well as after the final version. Table 5.2 presents the found bug, totaling 20 bugs. Of the bugs, 11 were logic bugs causing the runtimes to provide different results, and 9 were error bugs causing the runtimes to throw an unexpected exception.

5.3 Highlighted Bugs

This section will present two bugs: one logic bug and one error bug. The shown logical execution plans are simplified versions of the original plans we generated. This process of minimizing the execution plans is done by hand.

5.3.1 An example of a logic bug

One logic bug we found was related to variable aliasing. When a query contains multiple variables that reference each other, Neo4j treats those variables as aliases, i.e., only one physical column is allocated for all of these variables. This is an optimization. We were able to identify a bug in this mechanism whereby an alias was incorrectly identified, i.e., two columns that were not equivalent were considered as aliases. This resulted in incorrect results being returned to the user. A simplified test case of the identified bug is displayed in figure 5.3. Looking at the example, one non-nullable variable, `n0`, was introduced first, followed by a nullable variable, `n2`, which referenced `n0`. Variables introduced below an Optional operator are always nullable. For some rows, the correct value for `n2` was `NULL` whereas `n0` could never be `NULL`, but this bug ensured that `n2` was never set to `NULL`.

5.3.2 An example of an error bug

One error bug we found concerned the unexpected cancellation of rows. The pipelined and parallel runtimes are push-based and work on batches of rows simultaneously. In contrast, slotted and interpreted are pull-based and work on a single row at a time. Push-based runtimes have performance advantages, but one drawback is that it is difficult to process a query lazily. One of the optimization tactics employed by pipelined to combat

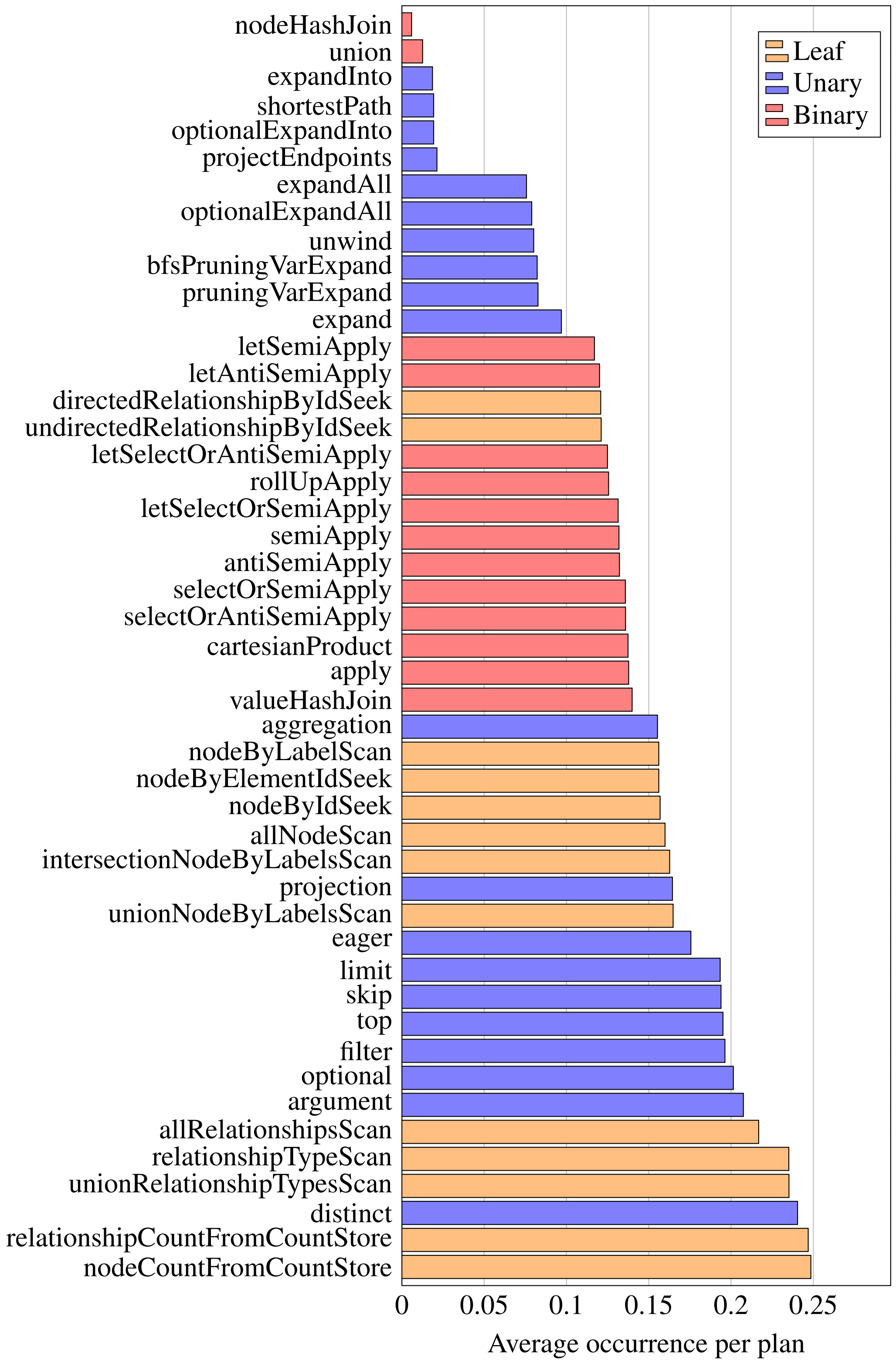

Figure 5.2: The plot shows the average number of occurrences per plan for each operator. The data is collected from 10000 generated plans. `ProduceResult` and `Sort` are omited from the plot and occur with a frequency of 1 and 1.55 per plan.

Description	Err/Log	Runtime	Fixed
NodeHashJoin should read correct slot from lhs	Error	S, Sc	Yes
Use RHS argument size slot allocation fix for primitive aggregations	Error	S, Sc	Yes
Null handling in LetSemiApply	Error	Sc, Pc, Pwc, PAc	Yes
Nested exception handling in byte code	Error	Sc, Pc, Pwc, PAc	Yes
Make sure slots are nullable after optional	Error	Sc, Pc, Pwc, PAc	Yes
Class cast exception in expand into	Error	P, Pc	Yes
ProjectEndpoints not filtering null rels	Logic	P, Pc, Pw, Pwc	Yes
OrderedUnion cancelled row bug	Logic	P, Pc, Pw, Pwc	Yes
ConditionalApplyBuffer only writing to one delegate	Logic	P, Pc, Pw, Pwc	Yes
Bug in OrderedUnion filter cancelled arguments	Logic	P, Pc, Pw, Pwc	Yes
Error in row cancellation on small morsel sizes	Logic	P, Pc, Pw, Pwc	Yes
Input cursors should not be reset	Logic	P, Pc, Pw, Pwc	Yes
Optional and Conditional-/Select-Apply handling of aliases	Logic	P, Pc, Pw, Pwc, PA, PAc	Yes
Race condition	Logic	PA, PAc	Yes
Account for NaN in StdDev	Logic	PA, PAc	Yes
Fused ProjectEndpoints producing null rows on optional input	Logic	PA, PAc	Yes
SlottedPipeOperator cancellation bug	Error	PA, PAc	Yes
Do not alias argument slots under Optional	Logic	S, Sc, P, Pc, Pw, Pwc, PA, PAc	Yes
Copy argument aliases to LHS slot config in cartesian product	Error	S, Sc, P, Pc, Pw, Pwc, PA, PAc	Yes
Bug in liveness analysis	Error	All	Yes

Table 5.2: This table presents the found, unique, confirmed bugs by the testing suite. In the Error/Logic column it is indicated what type of bug we categorized it as. In the runtime column we note in which runtimes the error occurred, the listing is not exhaustive but what we noted the bug as. I - interpreted, S - slotted, Sc - slotted with compiled expression, P - pipelined, Pc - pipelined with compiled expression, Pw - pipelined without fusion, Pwc - pipelined without fusion with compiled expression, PA - parallel, PAc - parallel with compiled expression

```

Logic bug

.produceResults("n0")
.apply()
.|.optional("n0")
.|.filter("false")
.|.projection("n0 AS n2")
.|.allNodeScan("n1")
.allNodeScan("n0")
.build()

```

Figure 5.3: A simplified logical execution plan for reproducing a logic bug. The expected result was 5 but we received -1.

this is to cancel upstream rows that have already been queued for later processing when the runtime recognizes it is safe to do so, e.g., if a `LIMIT` has already been reached. It is sufficient to say that pipelined operators must know this row cancellation mechanism and cannot process canceled rows. Thus, it is invalid for an operator to yield new output rows based on an already canceled input row. There was a bug in how `pruningVarExpand` handled canceled input rows, which could be reproduced by the plan presented in figure 5.4.

```

Error bug

.produceResults("r")
.semiApply()
.|.limit(1)
.|.pruningVarExpand("(c)<-[*1..1]-(d)")
.|.allNodeScan("c")
.relationshipTypeScan("(a)-[r:R]->(b)")
.build()

```

Figure 5.4: Simplified logical execution plan reproducing an error bug. The plan caused a NullPointerException.

5.4 Usefulness of the testing suite

Testing using logical execution plans in Neo4j proved helpful in identifying multiple previously unknown bugs in the Neo4j runtimes. Of the bugs reported, the distribution of bugs between error and logic bugs supports the idea that logic bugs are more commonly present in the later stages of query processing. The Neo4j runtimes are expected to be robust to semantically correct execution plans whether or not the plan can be planned in the current state of Neo4j parsing and planning stages. Thus, we see our bugs as crucial to the robustness of the Neo4j runtimes, which is emphasized further by the fact that the developers have fixed all the reported bugs shortly after reporting them.

From the results, we can see that bugs are present in all runtimes that have been tested, with most occurring in the pipelined configurations and configurations using the compiled expression engine. From speaking with the developers, we know that many of the bugs we have found have been in operators and combinations of operators that previously have been difficult to identify due to the limited test coverage reached through manual test writing. The developers at Neo4j intend to incorporate the tool into the routine testing procedures used during the development and maintenance of the runtimes.

After continuously running the testing suite and updating the Neo4j version to incorporate the fixes to previously identified bugs, we observed that the number of potential bugs detected by the tool decreased without further extensions to the testing suite. Our tool may have exhausted most of the bugs it can produce in the current version of Neo4j. There is still the possibility that there could be cases that produce bugs that have not been produced due to chance. We can assume that there are still bugs within the Neo4j runtimes. We believe that some of these can be reached with our tool if it is further improved to cover more operators and expressions or run for a more significant number of iterations.

5.5 Insights

The current implementation of the fuzz tester has proven successful at identifying bugs within the tested Neo4j runtimes. However, we have several insights from analyzing the generation of plans. Looking at table 5.1, we can observe that the number of non-comparable results increases with increasing average size of plans. This is expected as a larger plan is more likely to include combinations of operators with a high chance of canceling rows from the result. Generating larger plans is not, in itself, a desirable characteristic. However, a larger plan is more likely to include interesting data dependencies and combinations of operators that would not be possible to explore otherwise. Empty results are challenging to counteract without putting considerable limitations on the possible generations. One possible expansion would be to be more restrictive with how operators such as `LIMIT`, `SKIP`, and `Anti-Apply` operators as they are likely to cause empty results.

In figure 5.2, some further limitations of the fuzz tester are shown. The table shows that the tool is biased toward operators who do not have any requirements for their children. Thus, they are easy to plan. This limits the coverage of some operators, and it would be beneficial if we could plan these more often. A better spread of operator planning would allow for more varied combinations of operators, which would benefit the tool's coverage.

Studying figure 5.1, we find some odd behavior in the fuzz tester. When we increase the upper limit of the number of operators allowed in a plan, the plans are generally smaller, with a few significant outliers. This is counter-intuitive and is possibly an effect of the higher limit trying to create longer, complicated plans that are less likely to succeed, causing it to retry more often and be more likely to succeed only in making shorter plans. Nevertheless, a higher limit also allows for generating larger plans, which causes a few large outliers for the higher limit. Shorter plans will produce more comparable results and less variation of what can be planned on the right-hand side of the binary operators since they will reach their limit earlier and give leaf operators as a right-hand side to all unevaluated binary operators.

The uneven operator spread and the failure to generate large plans is, to a large degree, an effect of the generation architecture. With the plans constructed top-down and operators picked without knowing whether their children will meet the requirements, ScalaCheck will discard many potential plans that include more difficult operators. For a future exploration of execution plan generation, a bottom-up approach might be preferable to test. By starting from the bottom, the generation will always know what variables are available at every stage of the generation and can thus more intelligently choose relevant operators and expressions in parent operators than the current implementation can do.

Chapter 6

Related work

6.1 Random Generation of Semantically Valid Cypher Queries

Random Generation of Semantically Valid Cypher Queries is a previous thesis at Neo4j studying the generation of queries [3]. The project developed a fuzz tester using differential testing similar to ours, utilizing the idea that all runtimes should yield the same results. The implementation utilizes a context object to track the available values used in constructing the queries to increase data dependencies within the queries generated. Since they also generated updating queries, the parallel runtime had to be excluded from some tests, and the graph reset between tests. The project managed to identify 25 confirmed unique bugs, of which one was a logic bug located in the planning stage of the query processing and two error bugs located in a Neo4j runtime. The entire distribution of reported bugs is shown in table 6.1.

	Found
Parsing	3
Semantic analysis	7
Planning	13
Runtime	2
Total	25

Table 6.1: Locations in Neo4j processing layer of reported bugs by Random Generation of Semantically Valid Cypher Queries

6.2 GDsmith: Detecting Bugs in Cypher Graph Database Engines

GDsmith presents the first automated and portable testing tool for any graph database engine based on Cypher [5]. The tool is focusing on effectively identifying logic bugs. GDsmith generates data and queries using three open-source graph databases (Neo4j, RedisGraph, and Memgraph). As well as comparing the output from the different databases, they make a cross-version comparison. However, this is only for Neo4j(v.4.4.12 and v.3.5.0). To optimize the differential testing to allow them to find more logic bugs, they ensure that the generated queries give a high ratio of nonempty results. This is done by analyzing the generated data before creating the queries. The query is then created in two steps: firstly, a skeleton of the query is constructed, and then they generate patterns and expressions. When using these optimizations, GDSmith drastically increased the number of nonempty tests from 18% to 73%.

This study found 28 bugs across these three databases, 20 of which were logic bugs, and the rest were error bugs. Looking specifically at Neo4j, however, only two out of seven total bugs were logic.

Some of the limitations that they faced were quite similar to ours. One of these is dividing by zero, an exception they ignored just like we did. They also faced the issue with undefined behaviors regarding `NaN`, which sometimes threw an exception. All queries also had to be made deterministic, an issue we faced with the operators `Top` and `Skip`.

6.3 Dinkel: Fuzzing Graph Databases with Complex and Valid Cypher Queries

Dinkel is a tool created for fuzz testing on GDBSMs working to improve upon the work done in GDSmith and GDMeter [12]. The main contribution of Dinkel is the focus on increasing complexity within the queries generated through increasing data dependencies. This was done through accounting of more state information when constructing their queries. This practice was not used by previously published fuzz testers such as GDSmith and GDMeter. By tracking the query state during generation, Dinkel can know which variables, labels, types, and properties are available and can thus make intelligent use of them. As a result, the queries produced will utilize previously introduced values more significantly, allowing for increased data dependencies within the query. This is similar to how we have used the `PlanState` object in our implementation.

Dinkel was evaluated on Neo4j, RedisGraph, and Apache AGE, finding 53 unique, confirmed bugs. Of the reported bugs, 25 occurred in Neo4j, and all were error bugs. The testing suite does not test for logic bugs.

6.4 Comparison to execution plan generation

These three examples of query-generating fuzz testers show the usefulness of fuzz testing for GDBMS. With some differences in approach, they all successfully identified several previously unknown bugs with Neo4j and other GDBMS. Our approach shares several characteristics with these examples while limiting the scope of testing to the Neo4j runtimes.

Our results differ from that of the fuzz testers presented above, which is unsurprising as we can only search a subset of their search space while testing it more thoroughly. It is thus interesting to compare how the result from our approach compares to the more general one used by the query testers.

Lepik & Forsberg provide us with detailed documentation of at which stage of the query processing their identified bugs occurred, which is an interesting comparison. Out of their 25 identified bugs, only two were found within the Neo4j runtimes, while most were bugs within the parsing and planning stages of the query processing. It is interesting as their testing suite used differential testing between different Neo4j runtimes, similar to ours. This could suggest that query-based fuzz testing is insufficient for testing the later stages of query processing. We do not have access to the query processing stages in which the bugs identified by Dinkel or GDSmith occurred. Thus, we can not compare our results to their reach within the query processing.

An often-stated goal of papers presenting testing suites for GDBMS is finding logic bugs within a system. We were successful in finding a large number of logic bugs when testing on the runtimes directly. This differs from the results presented by Lepik & Forsberg as well as GDSmith, as they were only able to identify one and two logic bugs, respectively, in Neo4j. GDSmith was more successful at identifying logic bugs in other GDMBS than Neo4j. As far as we could tell, Dinkel did not utilize any method allowing them to identify logic bugs. By using execution plans for our fuzz testing, we can explore many situations for the Neo4j runtimes that might not be planned or only planned on a small number of occasions during normal operations of Neo4j. Every plan we generate would need to be individually investigated if one would like to determine whether or not an execution plan is currently possible to plan by writing a query and which configuration of Neo4j it would require. This is because there is no easy way to map from execution plans to queries, as any plan could correspond to many Cypher queries. Another possible explanation is that logic bugs are more often a consequence of errors within the runtimes of a GDMBS rather than the planning or parsing stages. We can conclude that testing the Neo4j runtimes has improved their robustness and correctness.

Compared to fuzz tester using queries, our approach has some apparent drawbacks. Firstly, unlike the three discussed testing suites, our approach does not provide portability between GDBMS. Both Dinkel and GDSmith ran tests on multiple GDBMS, and the tool provided by Lepik & Forsberg could be adapted to do so. This further allows for differential testing within a certain GDBMS and between multiple GDBMS using the same query language. Our approach is not easily modifiable between GDBMS as they use different underlying implementations to process the same query language. With that said, more tests using our approach on other GDBMS would be interesting as it would allow for a more general conclusion on the effectiveness of generating execution plans compared to queries.

Furthermore, to successfully implement an execution plan fuzz tester, the tester needs to be able to access the source code for the query processing of the GDBMS in question. Thus, our approach could be considered some kind of grey box testing. This limits the usefulness of this approach to database systems with open-source code in the case of external testing.

From this comparison, it is clear that how a testing suite is implemented will dictate what bugs it can identify within a system. Different tools will allow for coverage of different parts of GDBMS, and it is thus essential to have diversity in the approaches used for testing a system. Our tool has likely found bugs that the discussed query generators could not find, and most of the bugs they found were outside the scope of our tool. Execution plan generation is not a replacement for conventional testing using queries, as it would not provide sufficient coverage of the query processing in any GDBMS. However, it is an excellent complement to them as it can identify bugs that previously have been difficult to identify.

Chapter 7
Conclusion and Future Work

This thesis presents a novel approach to testing Neo4j through the random generation of execution plans. The work is a successful proof of concept of the approach's usefulness. Our testing tool found 20 bugs, of which 9 were error bugs, and 11 were logic bugs. Thus, we have covered parts of the query processing structure that were previously difficult to test using query fuzz testers exhaustively. Fuzz testing using execution plans is an effective testing approach within its limitations and complements conventional fuzz testers that randomly generate Cypher queries well. The approach requires comparatively more implementation, access, and knowledge of the system tested on than the query testers. However, it offers greater coverage in the later stages of the query processing. We will also note that the number of logic bugs identified within Neo4j is large compared to the number of error bugs identified, which could be of interest for future work aiming to identify logic bugs within database systems.

The practice of random generation of execution plans needs to be further explored, primarily through extending the practice to other graph database management systems and testing it on relational database management systems. Several implementation improvements and extensions could be made to provide more data dependencies within generated plans and increased coverage of the operators used. Possibly, many of the good practices and innovations used in generating queries could be adapted to execution plan generation.

References

[1] Cypher Manual. `https://neo4j.com/docs/cypher-manual/current/introduction/`. Accessed: 2023-11-21.

[2] Neo4j, Jun 2023. `https://neo4j.com/`. Accessed: 2023-11-21.

[3] Forsberg, Adam and Lepik, Andreas. Random Generation of Semantically Valid Cypher Queries, 2023. MSc Thesis, LU-CS-EX: 2023-07, Lunds University.

[4] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008.

[5] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 163–174, New York, NY, USA, 2023. Association for Computing Machinery.

[6] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Rickard Nilsson. ScalaCheck Documentation, 2021. `https://scalacheck.org/documentation.html`. Accessed: 2023-11-21.

[8] Vaios Patras, Petros Laskas, Kyriakos Koritsoglou, Ioannis Fudos, and Evaggelos Karvounis. A comparative evaluation of RDBMS and GDBMS for shortest path operations on pedestrian navigation data. In *2021 6th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pages 1–5, 2021.

[9] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and*

Privacy (SP), pages 615–632, Los Alamitos, CA, USA, may 2017. IEEE Computer Society.

[10] Manuel Rigger and Zhendong Su. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682. USENIX Association, November 2020.

[11] Andreas Seltenreich. Bug Squashing with SQLsmith, 2022. `https://github.com/anse1/sqlsmith`. Accessed: 2023-11-21.

[12] Dominic Wüst. Dinkel: Fuzzing Graph Databases with Complex and Valid Cypher Queries, 2023. B.S. Thesis, Advanced Software Technologies Lab ETH Zürich, `https://www.dwuest.com/dinkel_paper.pdf`. Accessed: 2023-11-27.

[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 2024-01-17

EXAMENSARBETE Randomly generating execution plans for bug detection in Neo4j
STUDENTER Joel Bergstrand, Theodor Åstrand
HANDLEDARE Niklas Fors (LTH), Love Leifland (Neo4j)
EXAMINATOR Görel Hedin (LTH)

Effektiv felidentifiering med hjälp av slumpgenerator

POPULÄRVETENSKAPLIG SAMMANFATTNING **Joel Bergstrand, Theodor Åstrand**

Att hitta och åtgärda fel i mjukvaruprodukter är något som utvecklare spenderar mycket tid på så att effektivt kunna identifiera fel är mycket värdefullt för ett mjukvaruföretag. Vi presenterar ett verktyg, som genom slumpmässig genering av exekveringsplaner kan testa grafdatabaser, specifikt Neo4j.

Att använda en databas för att lagra information är en självklarhet för alla typer av företag. En typ av databaser som blivit allt mer populär under de senaste åren är grafdatabaser. En grafdatabas struktureras som en graf där varje nod innehåller information och länkas samman av relationer. Ett populärt språk som används för att skapa och hämta ut data ur grafdatabaser är Cypher. Cypher har utvecklats för grafdatabasen Neo4j. Både språket och databasen expanderas och förbättras kontinuerligt och som alla mjukvaruprodukter så fungerar det inte alltid exakt som tänkt. Som användare förväntar man sig att databasen fungerar som utlovat, om detta inte är fallet går det då alltid att identifiera när det blivit fel?

Att säkerställa att resultaten från en databas är korrekta är ett ständigt pågående arbete men många fel är svårfångade och lyckas inte alltid identifieras innan de når användarna. Det är därför viktigt att ha en bra testningsmiljö för att hitta så många fel som möjligt, så tidigt som möjligt. För databaser som använder Cypher har en populär metod bland forskare varit att slumpmässigt generera en stor mängd korrekta frågor till databasen och på så sätt hitta fel.

Vi har tagit ett nytt angreppssätt för att hitta fel i grafdatabaser genom att istället slumpmässigt generera exekveringsplaner som sedan testas mot en databasinstans. En exekveringsplan är en datastruktur som en Cypher-fråga skrivs om till innan den kan hitta ett svar i databasen. Vi exekverar planen med olika konfigurationer som all förväntas ge samma resultatet och jämför sedan dessa. Genom denna process får vi två felkategorier, när en eller flera konfigurationer ger ett oväntat felmeddelande och när de olika konfigurationerna ger olika resultat tillbaka. Vi kan på så sätt effektivt identifiera fel i delar av databasen som tidigare varit svåra eller omöjliga att nå genom testning med slumpmässig generering av Cypher-frågor.

Detta angreppssätt för testning har visat sig vara mycket effektiv. Totalt hittade vi 20 fel i databasen, av dessa var 9 fel där databasen fick ett oväntat fel och 11 fel var tillfällen då vi fick olika resultat tillbaka som förväntades vara samma. Felen vi har identifierat skulle vara osannolika att identifiera med hjälp av slumpmässiga Cypher-frågor eller manuella tester. Vårt resultat visar att detta angreppssätt är effektivt och kan komplementera redan existerande testningsverktyg.