MASTER'S THESIS 2024

Modeling Profiling Data in a Graph Database for Performance Analysis

Richard Lundberg, Marcus Rettig

ISSN 1650-2884
LU-CS-EX 2024-47

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX 2024-47

Modeling Profiling Data in a Graph Database for Performance Analysis

Modellering av profileringsdata i en grafdatabas för prestandaanalys

Richard Lundberg, Marcus Rettig

Modeling Profiling Data in a Graph Database for Performance Analysis

Richard Lundberg
richard.lundberg.7072@student.lu.se

Marcus Rettig
marcus.rettig.6146@student.lu.se

May 30, 2024

Master's thesis work carried out at Neo4j Inc.

Supervisors: Simon Priisalu, simon.priisalu@neo4j.com
Jaroslaw Palka, jaroslaw.palka@neo4j.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

Benchmarking is an important part of the development process for any mission-critical application. By inspecting profiling data, developers can identify bottlenecks and performance regressions before they reach the customers. Neo4j runs an extensive benchmarking suite on its database, resulting in a huge collection of profiling data collected each week. These profiles are commonly visualized individually as flame graphs which are inspected manually. Finding patterns and differences among multiple profiles is difficult to do manually, due to the size and complexity of the data.

We propose a framework for identifying bottlenecks and regressions by modeling the profiling data as call-stack trees in a graph database. We demonstrate the usefulness of the framework for cross-profile analysis such as time series analysis and aggregation-based methods. We conclude that there is much potential in this approach and our thesis can be used as a decision basis for organizations wanting to implement a similar framework.

Using a graph database to model profiling data has many advantages and is suitable for the tree-like structure of the data. It makes the data more accessible and facilitates flexible querying in which the user can ask questions about the data and perform non-trivial aggregation. It has already aided Neo4j in the process of pinpointing the cause of some performance issues. The main disadvantage is the complexity involved in importing large quantities of data.

Keywords: Bottleneck, Regression, Call-Stack Tree, Graph Database, Cypher, Neo4j

Acknowledgements

We would like to express our gratitude to Simon Priisalu and Jaroslaw Palka for their invaluable insights that helped us progress during the work process. We would also like to thank Jonas Skeppstedt for guiding us through the writing process. Finally, we would like to thank everyone at Neo4j who provided us with constructive feedback, and special thanks to Love Leifland and Alfred Clemedtson for our rewarding discussions throughout the thesis.

Contents

1 Introduction **7**
1.1 Problem Statement . . . 7
1.2 Contribution . . . 8
1.3 Contribution Statement . . . 8
1.4 Outline . . . 8

2 Background **9**
2.1 Sampling Profilers . . . 9
2.2 Flame Graphs . . . 10
2.3 Graph Databases and Neo4j . . . 11
2.4 Introduction to Cypher . . . 12
2.4.1 Basic Cypher . . . 12
2.4.2 Data Aggregation . . . 14
2.5 Statistical Concepts . . . 15
2.5.1 Average . . . 15
2.5.2 Standard Deviation . . . 15
2.5.3 Correlation . . . 15

3 Approach **17**
3.1 Work process . . . 17
3.2 Products . . . 18
3.3 Automation and Expert Knowledge . . . 18

4 Modeling and Importing Data **21**
4.1 Schema . . . 21
4.1.1 Modeling . . . 21
4.1.2 An example . . . 23
4.2 Flame Graphs to Trees . . . 23

5 Detecting Bottlenecks **27**
5.1 Code Path Potential 27
5.2 Subtree Potential 28
5.3 Bottleneck Correlation 34

6 Detecting Regressions **37**
6.1 Time Series Analysis 37
6.2 Regression Correlation 39
6.3 Expanding Algorithms 39
6.3.1 Greedy Selection 41
6.3.2 Custom Selection 41
6.3.3 Correlation Selection 42

7 Evaluation **43**
7.1 Flexibility of the Framework 43
7.2 Cypher vs Local Analysis 44
7.3 Changes to the Benchmarking Pipeline 44
7.4 Use Cases at Neo4j 45
7.5 Presentation of Data 46

8 Conclusion and Future Work **47**
8.1 Conclusions 47
8.2 Future Work 48

References **49**

Appendix A Cypher Queries **53**
A.1 Time Series Analysis 53
A.2 Correlation 54

Appendix B Program Output **57**

Chapter 1

Introduction

1.1 Problem Statement

Neo4j collects large amounts of profiling data each week but the massive size and complexity make manual examination difficult. Every time a benchmark is run, it results in high-level metrics such as *operations per second* and low-level metrics such as the number of samples collected at each stack frame. Neo4j already has a few automatic regression detection algorithms in place, but these only consider the high-level metrics. The low-level metrics are usually inspected manually after a regression or bottleneck has been detected — first by the Benchmarking team, then by the code owners — to pinpoint the problem. This works well for individual test runs because there are many great visualization tools for such data, particularly flame graphs. Recognizing patterns among metrics from numerous test runs is infeasible to do manually, however, and that is the problem we will tackle in our project.

Currently, low-level metrics can only be accessed for one test run at a time. This usually means downloading some files from cloud storage and opening them locally in some visualization tool. This is inconvenient to do manually if we for example want to compare some metric for a specific function among multiple test runs. Even if we automated this process it would still be inefficient because the full files would be downloaded, even though we're only interested in a single function. These problems could be solved by storing the results in a database instead of files. More specifically, we will use a Neo4j graph database for this purpose.

The following research questions will be answered in this thesis:

- Is a graph database appropriate for modeling call-stack trees for performance analysis?
- How can the database be modeled for the detection of bottlenecks and regressions?

1.2 Contribution

During the project, we have developed a framework for analyzing and exploring large sets of profiling data. We have shown that a graph database, such as Neo4j, can provide a solid foundation for storing and querying the call-stack tree data required for analysis. Further, we have implemented several analysis methods, all based on the same proposed database schema. The purpose of these implementations is primarily to demonstrate the possibilities of our framework, and to serve as inspiration for future work. An organization looking to adopt a similar framework may want to design other analysis methods tailor-made for their specific requirements.

1.3 Contribution Statement

The work has been divided as follows. We have continuously discussed all ideas and come up with them together. Most of the coding has also been carried out together through pair programming. However, Richard has focused more on the implementation of correlation and expanding algorithms while Marcus focused more on the subtree potential and time series analysis. The writing was divided evenly, where one author wrote a section and the other reviewed it.

1.4 Outline

The thesis is structured as follows. Chapter 2 provides sufficient background knowledge to understand graph databases and related concepts used in the remainder of the thesis. Chapter 3 describes the taken approach to answer the research questions and provides an introduction to chapter 4, 5 and 6. Chapter 4 describes the process of modeling an appropriate schema and importing data. Chapter 5 and 6 further explain our implementations for detecting bottlenecks and regressions respectively. Chapter 7 presents an evaluation of the proposed framework by discussing its practical use. Chapter 8 sumarizes our conclusions and most important insights and suggests some potential future work.

Chapter 2

Background

2.1 Sampling Profilers

A sampling profiler is a tool that captures snapshots of which functions are running at a given frequency. Frequency-based sampling creates a statistical analysis of the executing program. Statistical profilers are less intrusive and have lower overhead than event-based deterministic profilers that sample all methods and influence the execution time by themselves. Functions that run quickly may however with variability fall in between two sampling time points, causing the final profile to become less accurate [5].

Neo4j uses the async-profiler in their benchmarks, which is an open-source project available on GitHub. The async-profiler is proficient in tracing multiple events such as CPU cycles, cache misses and allocations in Java Heap etc. In this thesis, we focus on traces sampled with CPU cycles. The profiler may be run with a multitude of options, among others, allowing for the tracking of compile modes and producing a flame graph of the call traces [2], which is the topic of the next section. CPU time profiling can be misleading since it only captures the time that the CPU was busy in each context. It fails to capture many events, such as I/O operations and sleeping or otherwise blocked threads. A more comprehensive way of sampling is *wall-clock time* profiling which samples all threads equally, regardless of their status [4]. The async-profiler is capable of such profiling but it is currently only utilized in a small subset of benchmarking at Neo4j. Sampling with CPU time places more focus on the actual functions and disregards circumstantial events. The wall-clock time profiling normally contains large amounts of superfluous data that can drown out the relevant parts, especially when visually inspecting it.

2.2 Flame Graphs

A flame graph is a visual representation of stack traces, intending to simplify the inspection of CPU usage. The x-axis of the flame graph shows the stack profile population sorted alphabetically and the y-axis shows stack depth, building with the root function from zero and expanding upwards with branching child functions. Since the graph lacks the time dimension, equivalent stack traces are aggregated along the x-axis. Each function is visualized as a bar and often color-coded with warm colors, resulting in a resemblance with flames, hence the name. The length along the x-axis corresponds to the number of samples recorded within this function or any of its children, often resulting in the main function spanning the entire base of the graph. A flame graph may be converted into a collapsed file format that represents each trace as a line in a text file with corresponding samples.

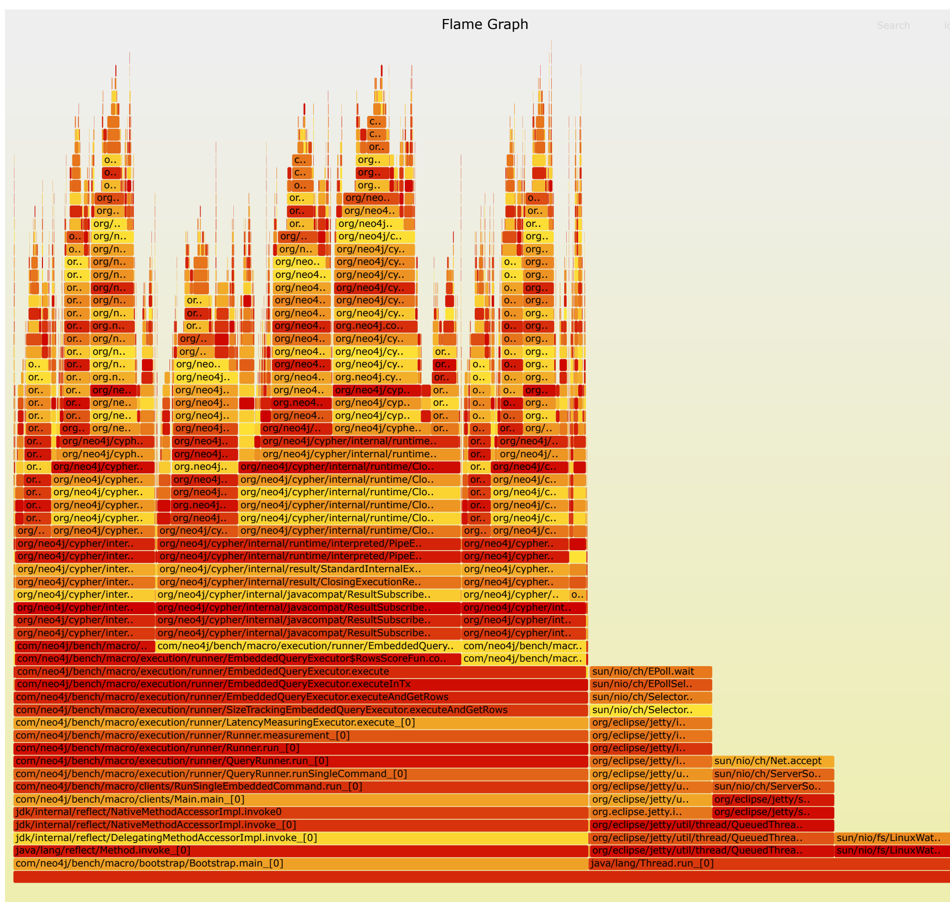


Figure 2.1: An example flame graph from benchmarking a part of the community edition of Cypher. We can determine that a main and at least one secondary thread has been started which causes a division right from the start. Higher up in the graph, the bars further divide into subtrees and diminish in size for each split, representing the total amassed samples in each subtree.

Flame graphs may also be compared by extracting the difference between two given graphs, denoted as a *differential flame graph*. The purpose of the differential flame graph is to visualize

how two separate runs of the same, or preferably at least similar programs have differed in their resource consumption. The bars of these comparative flame graphs typically display a red shade for functions that increased and blue shades for those that decreased [3].

Tornetta [7] presents some algebraic properties of flame graphs and suggests, for regression detection, comparing a statistically significant sample of flame graphs before the regression with a sample after the regression has occurred, and subsequently examining the distributions between the two sets. They propose a two-sample Hotelling T^2 test, which is a multivariate probability distribution suitable for sample statistics.

In figure 2.1 we see an example of a flame graph produced by profiling a specific query in the community edition of Cypher. The flame graph is normally interactive and hovering a bar would reveal the number of samples.

2.3 Graph Databases and Neo4j

The core product of Neo4j is its native graph database. In contrast to a relational database, a graph database stores nodes and relationships instead of tables or documents. In Neo4j's graph database, nodes represent entities of a domain and can be classified with labels and described by properties. A relationship is always directed and connects a source and target node. Relationships must be classified by a label that describes their type and can, just like nodes, be described by properties. The advantage of graph databases compared to relational databases reveals itself when querying for complex connections between entities. Since the relationships are pre-established, the connections can be instantly traversed without further calculation during runtime [9]. Figure 2.2 is a simple example of how a small company with a couple of employees and interconnecting relationships may be stored. Neo4j also has a built-in web interface that offers the possibility of visually inspecting and interacting with a graph.

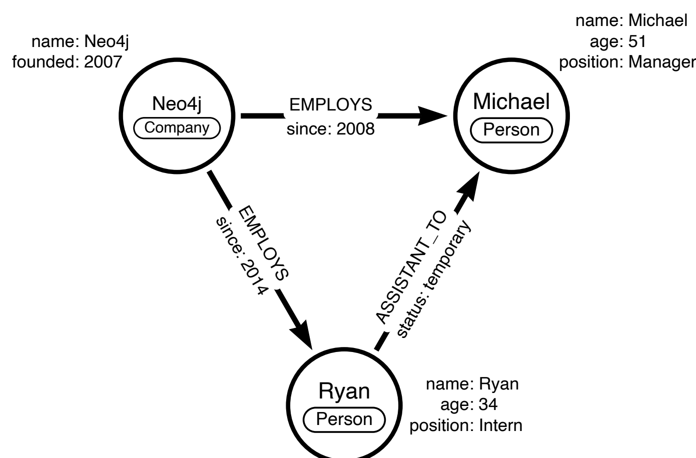


Figure 2.2: An example of a graph in Neo4j displaying the nodes and relationships of a small company. Each node is classified by the label 'Company' or 'Person' and described by individual properties such as name and age. The entities are connected by directed relationships with labels, e.g. `EMPLOYS`, and their respective properties.

2.4 Introduction to Cypher

Cypher is a query language, designed by Neo4j specifically for retrieving graph data from a graph database such as Neo4j. It is a *declarative* language, allowing the user to "focus on what to retrieve from [the] graph, rather than how to retrieve it" [8]. Cypher is used extensively throughout our project. This section is intended to give the reader sufficient background knowledge to be able to understand the rest of the report. For more in-depth details about Cypher we refer to the Cypher Manual [8].

2.4.1 Basic Cypher

Most Cypher queries include at least one `MATCH` statement, which is used to find nodes and relations in a graph matching a specified pattern. For example,

```
MATCH (p:Person { name: 'Michael' })
RETURN p
```

matches and returns all `Person` nodes where the name property is `'Michael'`. It is also possible to match relationships, for example,

```
MATCH (p:Person { name: 'Michael' })<-[:ASSISTANT_TO]-(a:Person)
RETURN p, a
```

matches all `ASSISTANT_TO` relationships from any person to any person named `'Michael'`, and returns all pairs of nodes with such a relationship. Note that the syntax resembles two nodes with a left-pointing arc between them. Here, `p` and `a` are variables bound to the matched nodes. When matching on multiple properties it may be more readable to specify the properties using the `WHERE` keyword. This also allows for more complex expressions. For example,

```
MATCH (p:Person)<-[:ASSISTANT_TO]-(a:Person)
WHERE p.name = 'Michael' AND a.age > 30
RETURN p, a
```

matches all `ASSISTANT_TO` relationships from any person over the age of 30 to any person named Michael and returns all pairs of nodes with such a relationship.

It should be noted that the above queries may return the same node twice if they have multiple matching relationships. We can use the `DISTINCT` keyword to exclude duplicates before returning data. For example, to return all persons who are the assistant to someone named Michael, the following query could be used:

```
MATCH (:Person { name: 'Michael' })<-[:ASSISTANT_TO]-(a:Person)
RETURN DISTINCT a
```

Another way to do the same thing is to use the `WHERE` keyword together with an inner query:

```
MATCH (a:Person)
WHERE EXISTS ((a)-[:ASSISTANT_TO]->(:Person { name: 'Michael' }))
RETURN a
```

That would match any person who has a `ASSISTANT_TO` relationship to some person named Michael. It is also possible to order and limit the results. For example,

```
MATCH (p:Person)
RETURN p
ORDER BY p.age DESC
LIMIT 10
```

would return the ten oldest people in the database. Here the `DESC` keyword is used to specify that the values are ordered in descending order. The `LIMIT` keyword is then used to select the first ten records.

Before we can match something, we first need to populate the database. This can be achieved through either the `CREATE` or `MERGE` statements. `CREATE` is the simplest, and to create the node with the name Michael that we previously matched, we would write the following code:

```
CREATE (p:Person { name: 'Michael' })
RETURN p
```

Similarly, a larger pattern with nodes and relationships can be created all at once in the same query:

```
MATCH (p:Person { name: 'Michael' })
CREATE (a:Person { name: 'Ryan' })-[:ASSISTANT_TO]->(p)
RETURN p, a
```

In this query, the `Person` node with the name-property `'Michael'` is already present in the database from the previous query and we match it before the `CREATE` statement. This syntax displays how to connect new data to existing nodes.

The `MERGE` statement works similarly but will only create components that are not already present in the database without having to match them first. For example:

```
MERGE (p:Person { name: 'Michael' })
RETURN p
```

In contrast to the `CREATE` clause, this query would not create a duplicate node since we are using `MERGE` and the `Person` node with the name `'Michael'` is already present in the database.

Another useful clause is `UNWIND` which transforms a list back into individual rows. For example:

```
UNWIND ['Michael', 'Ryan'] AS firstName
CREATE (p:Person { name: firstName })
RETURN p
```

It is also possible to pass arguments to Cypher from a given programming language. Arguments are prefixed with the '$'-sign. For example, in Python:

```
driver.execute_query(
    """
    UNWIND $names AS firstName
    CREATE (p:Person { name: firstName })
    RETURN p
```

```
        """,
        {'names': ['Michael', 'Ryan']},
    )
```

The above queries will each create two separate nodes with the names 'Michael' and 'Ryan'. This syntax is especially useful when populating a database with millions of nodes, each with individual property values. Passing arguments allows us to separate the query and the data, meaning that the same static query string can be reused for variable data.

2.4.2 Data Aggregation

Aggregating data is the process of performing calculations on multiple values to produce fewer values — many times a single value. This is an important feature for databases for at least two reasons. First, because the calculations are performed on the same machine that stores the data it can significantly reduce the amount of data that has to be transferred over the network. Second, because data is stored in a well-known format, the query language (e.g. Cypher) can generate an optimized execution plan for running the query efficiently, often without extra effort from the programmer.

Cypher has many built-in functions for data aggregation. A simple example is the following query which returns the average age of all `Person` nodes in the database:

```
MATCH (p:Person)
RETURN avg(p.age)
```

Another useful built-in aggregating function is `collect` which is used to create an aggregated list. The `collect` list is often used in conjunction with grouping keys to create groups of data. For example, the following query returns a list of names for each unique value of `age` that exists in the database:

```
MATCH (p:Person)
RETURN p.age AS age, collect(p.name) AS names
// Example return value:
// { age: 46, names: ['Michael', 'Phyllis'] }
// { age: 81, names: ['Creed'] }
```

The above query also uses the `AS` keyword to rename the values included in the returned records. Note that, in Cypher, all values that are not the result of an aggregation expression will be used as grouping keys. This is in contrast to SQL where grouping keys are expressed explicitly using `GROUP BY`.

In Cypher, it is also possible to perform aggregation in several steps. One way is to use the `WITH` keyword to create a chain of calculations where the result of each part of the query is passed on to the next part. For example, the following query calculates the average age among the five oldest people in the database:

```
MATCH (p:Person)
WITH p ORDER BY p.age DESC LIMIT 5
RETURN avg(p.age)
```

The query first matched all people in the database. Then the five oldest people are selected

and passed to the next step. Finally, the average age is calculated and returned.

2.5 Statistical Concepts

This section contains a short introduction to some basic concepts in statistics and time series analysis and the notation used in the report.

2.5.1 Average

We denote the average of n observations $x_1, x_2, \ldots, x_n$ as

$$\text{avg}(x_1, x_2, \ldots, x_n) = \frac{x_1 + x_2 + \cdots + x_n}{n}.$$

2.5.2 Standard Deviation

We denote the standard deviation of n observations $x_1, x_2, \ldots, x_n$ as

$$\text{std}(x_1, \ldots, x_n) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \text{avg}(x_1, \ldots, x_n))^2}.$$

We often need to account for missing observations in the calculations. If an observation is missing at a certain time point it means that the sampling profiler sampled it zero times and the observed value is implicitly zero. Given n observations and m missing observations the standard deviation can be calculated as

$$\text{std}(x_1, \ldots, x_{n+m}) = \sqrt{\frac{1}{n+m-1}\left(m \cdot \text{avg}(x_1, \ldots, x_{n+m})^2 + \sum_{i=1}^{n}(x_i - \text{avg}(x_1, \ldots, x_{n+m}))^2\right)}.$$

The standard deviation is the square root of the variance and is a measure of how much the observations are expected to deviate from their average.

2.5.3 Correlation

A common measurement for linear correlation between two sets of data is the Pearson correlation coefficient, denoted

$$\text{pcc}(\{x_1, \ldots, x_n\}, \{y_1, \ldots, y_n\}) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}},$$

where

$$\bar{x} = \text{avg}(x_1, x_2, \ldots, x_n),$$
$$\bar{y} = \text{avg}(y_1, y_2, \ldots, y_n).$$

Chapter 3

Approach

Our approach combines several methods that help make sense of large quantities of profiling data.

3.1 Work process

We employ an iterative work process, where we continuously improve our product based on new learnings and experience. The tools we build are intended to make it easier for engineers to identify bottlenecks and regressions, so an important part of the process is to collect feedback from the engineers at Neo4j — our users.

The scope of our work is limited to intuitive methods that produce predictable results that can be easily understood by us and by our users. Specifically, we focus on statistical methods and data aggregation. These methods work well for big data sets, too big to fit in memory, because many calculations can be performed directly in the database.

Our approach relies heavily on the use of a Neo4j database to store data. We built a Java program that automates the process of importing flame graph data into such a database. Given a specific benchmark, it downloads the profile recordings and imports them into the database. We made a few iterations of this program to support new requirements as they arose. We also put effort into making this program performant to support our iterative work process.

The rest of our work was carried out iteratively. We started with simple methods that only consider single functions. We realized early — by looking at the results and feedback from our users — that single functions are often hard to improve by themselves. In most cases, it is more important to know in which calling contexts these functions perform badly. For this

reason, later iterations of our work focus on methods that consider calling contexts consisting of multiple functions. The final iterations focus on presenting our results in an intuitive way to our users.

3.2 Products

Addressing our problem statement, there are two fundamental problems to solve: bottleneck and regression detection. Bottleneck detection, in this context, refers to the process of discovering problematic functions in the code base that are negatively impacting the performance of the executing program. Bottleneck detection implies searching for a pattern that is performing poorly in a space of functions over multiple benchmarks. Regression detection, on the other hand, looks at variations over time for a single benchmark. This analysis requires that we have a time series of previous executions with normal behavior and a regressing execution with worse performance. The aim is then to identify the cause of this regression, whether it be a change in the code base or perhaps a bug.

We apply different approaches to tackle the two issues, although they share some similarities. A common ground to build from is the profiling data that we require to analyze the test runs. As previously mentioned, Neo4j utilizes the async-profiler in their benchmarking process. The results of the sampling it performs are represented as flame graphs, which can be folded into a collapsed format that we subsequently import into a Neo4j graph database. The data is stringed together in the database as a call-stack tree with both sampling properties from the profiler and calculated properties for easier querying. With regression detection, we perform a time series analysis for each function to see if it has deviated from its expected behavior. With bottleneck detection, we instead calculate a *potential* improvement if the function could be reduced to zero samples. We then expand on promising functions to consider a larger context.

Another approach that applies to both scenarios, is looking at the correlation between the samples of a function and the execution time of the entire program. An important discovery we made during this process is that there exist some "usual suspects" that appear both when looking for bottlenecks and regressions. The intuition is that a small worsening in the performance of a bottleneck function will have a large impact during a regression. Performing a regression detection may therefore expose the bottleneck functions of a given program.

We focus on identifying suboptimal functions that take large fractions of the total execution time. According to Amdahl's law, the overall improvement of a system gained by optimizing a single part is limited by the fraction of time that is spent on that part [6]. This relationship is commonly used in the context of parallel computation but can be generalized so that it applies to a wider domain of optimization.

3.3 Automation and Expert Knowledge

Early on in the process, we realized that building a completely automated system without human interaction would be infeasible because expert knowledge often is required to make the final decision. The nature of the profiling data can be quite deceiving, which forces us to

apply a broader selection to avoid excluding false negatives. This makes it significantly more difficult to determine which node is the underlying issue and ultimately requires inspecting the actual code. Knowledge of the code base will quickly expose the false positives and help identify the real candidate functions for further inspection. Here is where expansions come into play. Even without a lot of previous experience with the code base, looking at expanded stack traces will give hints about where the real issue resides. For this purpose, we developed expanding algorithms that attempt to capture the most relevant branches connected to the initial candidate functions. Each initial selection is paired with a corresponding expanding algorithm that looks at the same properties but in a larger context. The system is however kept modular so that any one initial selection can be expanded with any algorithm, although it may not always be suitable. For example, one could produce an initial list of candidate functions from the bottleneck detection and then explore its surroundings with regression expansion. Normally it makes more sense to expand with the same process as with the initial selection.

The output is presented in the order of importance deemed by our algorithm. Less important data is displayed progressively further down. As much data as possible is kept and output to avoid filtering away false negatives. The number of presented candidates is determined by a user-specified argument.

Chapter 4

Modeling and Importing Data

This section further describes the process of importing profiling data to a Neo4j database.

4.1 Schema

When importing huge amounts of data, a suitable graph structure is crucial for efficient query performance. It is additionally important that the graph is intuitive as the database will be accessible to many employees and understandability can expedite the process of identifying bottlenecks or regressions. A clear structure can furthermore assist the developer in the task of finding problematic functions when visualizing the graph.

4.1.1 Modeling

To reinforce the structure of our schema, we initially outlined our core use cases for the database. We formulated these use cases by discussing the needs of people who work daily with benchmarking at Neo4j.

Our core use cases are as follows:

- Find all benchmarks where a specific function took more than a given percentage of samples.
- Acquire test run, benchmark name and benchmark group from any given function.
- Acquire an entire stack trace tree for a specific test run with an absolute number of samples.

Secondly, we decided on what data to aggregate and decided on a "Calling context tree" which resembles the format of a flame graph and keeps enough valuable information for a statistical analysis.

The structure of our database was influenced by Neo4j's result store database where the company currently stores high-level information about previously run benchmarks. The result store contains information required to access specific benchmarking data from cloud storage, which we use to populate our database. Aligning our model with the structure of the result store was appropriate to retain homogeneity and for cross-database querying. This also makes a future merge of the two databases possible. From our core use cases, we produced a high-level abstraction of our model that served as a basis for our code implementation seen in figure 4.1.

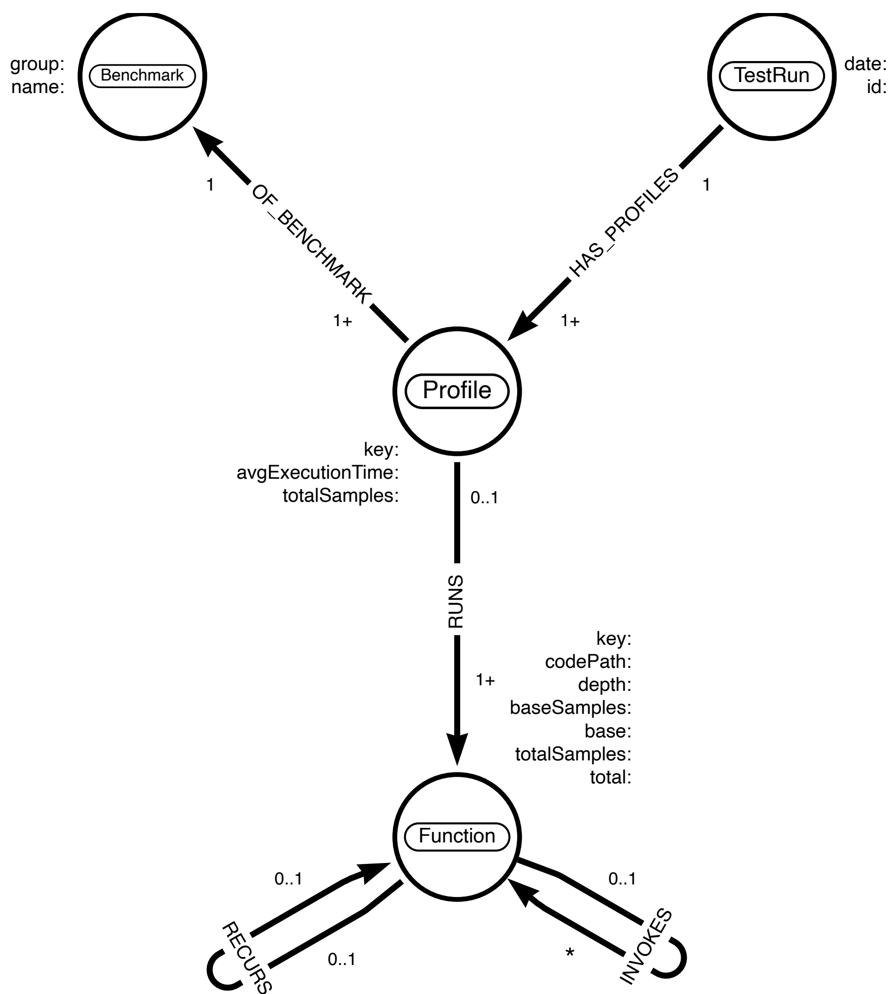


Figure 4.1: Model Schema. A relationship cardinality is read independent of the direction of the arrow e.g., a benchmark maps to at least one profile and a profile maps to a single benchmark

A profile is uniquely identified by a benchmark and a test run. A test run and a benchmark may however map to many unshared profiles. The database successively grows into a call-stack forest connected through test runs and benchmarks with each new flame graph import. While creating this schema there were a few design choices to consider, such as the direction

of different types of relationships, and how to handle multiple calls to the same method in different contexts. The direction of the relationships proved a trivial problem as Neo4j asserts in their graph modeling guidelines: "Although you must store a relationship in a particular direction, Neo4j has equal traversal performance in either direction..." [9]. The issue with multiple calls in different contexts to the same function meant that the database would need to enforce some unique constraint to define a function node other than simply the name of the function — also known as the *code path*. After consulting the Neo4j documentation we deduced that creating a unique key for each node would be suitable to solidify the contrast between multiple nodes with identical code paths.

With time, our demand for identifying recursive relationships developed. A recursive relationship may stretch from an ancestor to a far-removed descendant with the same code path, but never past its first recurrence in each branch.

With this structuring, the core use cases are easily fulfilled with just a few lines of Cypher code, and the model lends itself nicely to powerful computations in trivial execution time.

Each function node has several properties. The most important ones are explained in the following list:

- `key`: A unique identifier for the node.
- `codePath`: The qualified name of the function.
- `depth`: The number of invocation edges between this node and the root node.
- `baseSamples`: The number of samples collected for the trace consisting of the node and all of its parents.
- `totalSamples`: The sum of base samples for this node and all of its descendants.
- `base`: The normalized number of base samples, calculated by dividing the base samples of the node by the total samples of the root node.
- `total`: The normalized number of total samples, calculated by dividing the total samples of the node by the total samples of the root node.

4.1.2 An example

In figure 4.2, we illustrate a simplified example of a single call-stack tree in our database. $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ have both a `RECURS` and an `INVOKES` relationship since they are representations of the same code path being called at different depths. Note that A_1 and A_2 also have a `RECURS` relationship even though they are not immediate neighbors.

4.2 Flame Graphs to Trees

We begin the process of importing the flame graphs to Neo4j by converting the flame graph files to a collapsed format. We then parse these files, line by line, generating nodes with corresponding properties. The trees are incrementally generated in memory before importing them into the database. This entire process is multi-threaded, with each thread handling one

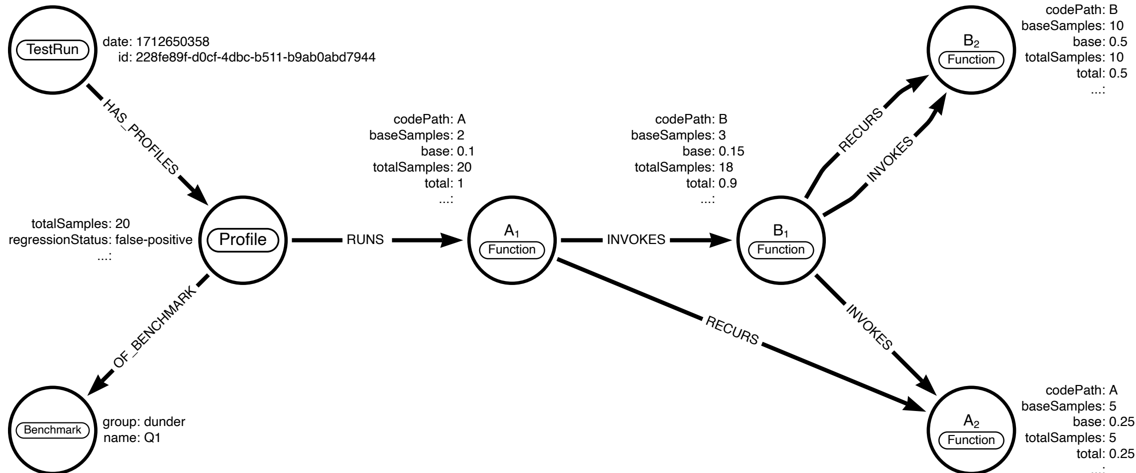


Figure 4.2: An example of how a test run with a single profile may be populated in our database.

tree at a time, preserving thread safety as each tree of function nodes is entirely independent of others. At last, we push the profile node, rooting the tree, and possibly create the benchmark and test run nodes with the `MERGE` operation. Any of these latter two nodes may already be present in the database or multiple threads may be racing to create them. To handle this race condition, we enforce a uniqueness constraint on the database which will cause all but one of these transactions to fail. Luckily, Neo4j is ACID compliant [10], guaranteeing data validity even in this eventuality.

Importing no more than a given week of benchmarking data means creating tens of millions of function nodes and yet even larger amounts of relationships. Populating the database at a sufficient speed is critical for the usefulness of our program. Any given day of benchmarking data generally constitutes too much information to push in a single transaction and will throw an out-of-memory error as the heap space runs out on the RAM of the hosting server. Even pushing just a single profile all at once is infeasible because the query would become too large. A basic approach to handle these problems is to populate the database in increments, starting with the root node

```
CREATE (fn:Function { codePath: "root" })
```

and later matching the root again and creating any children together with the relationships between them using the `MERGE` operation in a separate query:

```
MATCH (fn:Function { codePath: "root" })
MERGE (fn)-[:INVOKES]->(childFn:Function { codePath: "child" })
```

This last query will then be repeated for each new node. However, the efficiency of this approach proved entirely unsatisfactory for our purposes as `MERGE` acquires locks on all matched entities before creating the missing elements of a pattern and all internal nodes are matched once for each of its children. In the final version of our importer, we utilize the `UNWIND` operation to first `CREATE` all of the nodes for a profile:

```
UNWIND $functions AS f
CREATE (fn:Function { codePath: f.codePath})
```

Then in the second phase, once again making use of `UNWIND` and `CREATE`, we import all of the relationships between the nodes:

```
UNWIND $invocations AS i
MATCH (a:Function { codePath: i.fromCodePath })
MATCH (b:Function { codePath: i.toCodePath })
CREATE (a)-[:INVOKES]->(b)
```

With this implementation, we avoid the unnecessary round-trips of the previous solution as we only need one query for functions and one for invocations for each profile. Finally, we merge the profile, benchmark and test run nodes together with their respective relationships. All these phases are contained within a single transaction and thanks to the atomicity property, we can rest assured that if one phase fails, they all fail and are subsequently rolled back. A failed transaction is retried with exponential backoff [10].

For more efficient data retrieval, we impose several indexes on the database, creating copies of selected primary data [8]. These indexes significantly reduce access time and speed up the import of relationships since these require initially matching the nodes to be connected.

Chapter 5

Detecting Bottlenecks

Software bottlenecks are segments of the code base that negatively affect the performance of the system as a whole. Subsequently, improving just a few bottlenecks can significantly improve the overall performance of the system. The Pareto principle (also known as the "80/20 rule") states that focusing on the top 20% of causes will affect 80% of the consequences. In other words, optimizing a few vital functions will yield substantial improvements. To identify these vital bottlenecks, we propose two different approaches: *potential* and *correlation*.

5.1 Code Path Potential

Given a single code path, we define the *code path potential* as the maximum system performance improvement in percent that can be achieved by optimizing the given code path. Intuitively, this is a measure of how much the system performance would improve if we could eliminate this code path.

Given a selection of profiles, we want to calculate *code path potential* for each unique code path in this selection, and then pick the n code paths with the highest potential. This is achieved by aggregating the number of base samples for each code path and then dividing by the total number of base samples of all function nodes in the selection of profiles. We performed the aggregation step in Cypher as follows.

```
MATCH
    (f:Function)
WHERE
    f.profileKey IN $profile_keys
WITH
    f.codePath AS codePath,
```

```
        sum(f.baseSamples) AS potential
    RETURN
        codePath,
        potential
    ORDER BY
        potential DESC
```

With the aggregated data, we calculate the final percentual potential in Python:

```
df["potential %"] = 100.0 * df["potential"] / df["potential"].sum()
```

The result is a list of code paths ordered by their potential to improve the program. For a particular selection of profiles we got the following results:

Code Path	p_0
org/neo4j/kernel/impl/store/RecordPageLocationCalculator.offsetForId	7.83
org/neo4j/kernel/impl/store/format/BaseRecordFormat.longFromIntAndMod	2.65
org/neo4j/kernel/impl/store/format/BaseOneByteHeaderRecordFormat.has	1.28
org/neo4j/kernel/impl/store/record/RelationshipRecord.initialize	0.99
org/neo4j/kernel/impl/store/format/BaseOneByteHeaderRecordFormat.isInUse	0.85

This would suggest that the `offsetForId` has the highest potential. It could theoretically improve the system performance by almost eight percent. This is easier said than done, however. Looking at the source code[1]

```
int offsetForId(long id, int recordSize, int recordsPerPage) {
    return (int) (id % recordsPerPage) * recordSize;
}
```

one quickly concludes that this function by itself leaves little room for improvement. Further noticing that the `offsetForId` function does not call any other functions, implying that it is a leaf node in the flame graph. Many of the code paths that we find when looking at them individually are indeed simple functions close to the leaf set. The reason these simple functions show up in our results is that they are called many times by other functions. In these cases, it would be more interesting to compare larger calling contexts instead of single code paths. Such methods are explored in later sections.

5.2 Subtree Potential

Similar to how we defined *code path potential* earlier, we define *subtree potential* as the maximum system performance improvement in percent that can be achieved by optimizing a given subtree containing multiple connected code path nodes. Intuitively, this is a measure of how much the overall system performance would improve if we could make all code paths included in the subtree run in zero seconds.

In the end, we want to find a small selection of nodes so that a programmer familiar with the code can quickly analyze the results. We therefore choose to limit our comparison to subtrees

[1] Publically available on Github

of a given size. We further limit our search to subtrees of a specific shape to reduce the search space. It might be possible to search the entire search space using constraint programming which could be explored in future research.

Given a node u, we define the full n-level subtree rooted in u as the subtree containing u and all its descendants reachable at a distance n from u. The search space will be limited to such subtrees.

Precalculating potential for each node

Given a node u, the subtree potential of degree n, defined $p_n(u)$, is the possible improvement that can be achieved by improving u or any descendent with a maximum distance n from u.

For degree 0, only u is considered with no descendants so $p_0(u)$ is simply equal to the base samples of u, that is,

$$p_0(u) = \text{base}(u).$$

For degree 1 we consider u and any immediate children of u so $p_1(u)$ is the base samples of u, plus the sum of base samples for all immediate children of u, that is,

$$\begin{aligned} p_1(u) &= \text{base}(u) + \sum_{v \in \text{children(u)}} \text{base}(v) = \\ &= \text{base}(u) + \sum_{v \in \text{children(u)}} p_0(v). \end{aligned}$$

Similarly, for degree 2 we get,

$$p_2(u) = \text{base}(u) + \sum_{v_1 \in \text{children}(u)} \left(\text{base}(v_1) + \sum_{v_2 \in \text{children}(v_1)} \text{base}(v_2) \right),$$

and we see that the parenthesis is equivalent to $p_1(v_1)$. Inserting this we get the simplified expression

$$p_2(u) = p_0(u) + \sum_{v \in \text{children(u)}} p_1(v).$$

Higher degrees follow the same pattern, so the general case can be neatly expressed recursively as:

$$\begin{aligned} p_0(u) &= \text{base}(u), \\ p_n(u) &= p_0(u) + \sum_{v \in \text{children(u)}} p_{n-1}(v). \end{aligned}$$

By precalculating $p_0, p_1, \ldots, p_n$, in that order, we can calculate p_n for all nodes in an initial forest in $O(n \cdot |V|)$, where $|V|$ denotes the number of nodes in the initial forest. This is sufficiently fast for our purposes, especially since we are mostly interested in relatively small subtrees.

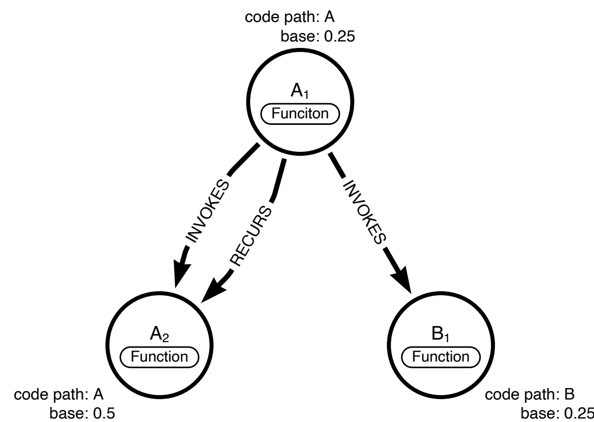
Aggregating potential for subtrees

Let C_i denote the set of all nodes with a given code path i. Under the assumption that no subtree contains any recursions, the potential for the full n-level subtree rooted in i is the sum of all the nodes reachable by a maximum distance n from any node in C_i. Because we already precomputed the potential for each node, the aggregated potential is conveniently and efficiently computed with:

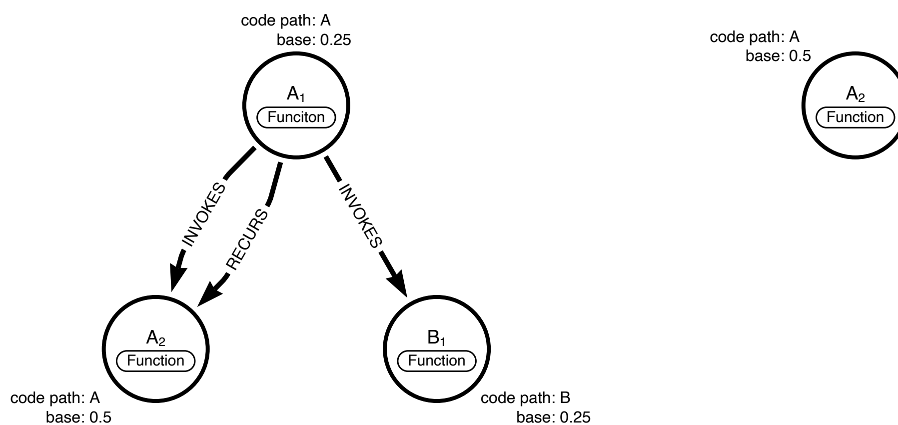
$$P'_n(C_i) = \sum_{c \in C_i} p_n(c).$$

The recursion problem

The problem with the simple aggregation $P'_n(C_i)$ is, as stated, that it only works under the assumption that there are no recursions. To illustrate why, let us consider the following example:



This simple program only has two functions (code paths) A and B. Let us now calculate $P'_1(C_A)$. We start by finding all full 1-level subtrees rooted in A, which would be:



Performing the aggregation we get

$$\begin{aligned} p_1(A_1) &= \text{base}(A_1) + \text{base}(A_2) + \text{base}(B_1) = 1.0, \\ p_1(A_2) &= \text{base}(A_2) = 0.5, \\ P'_1(C_A) &= p_1(A_1) + p_1(A_2) = 1.5. \end{aligned}$$

That is, the program can be improved by 150%. Something went wrong because we can impossibly improve the program by more than 100%. The problem is that the A_2 node was included twice in the sum. Once in $p_1(A_1)$ and once in $p_1(A_2)$.

Accounting for recursions

Let us now correct the aggregation to also account for recursions. Because we are only considering full n-level subtrees we can precalculate a correction term for each subtree. This term will compensate for everything that would otherwise be counted multiple times. In the example above we would like to exclude the A_2 node from the left subtree, rooted in A_1, so the correction term corresponding to the first-degree subtree potential is $c_1(A_1) = \text{base}(A_2)$. The right subtree, rooted in A_2, contains no recursions so its correction term is $c_1(A_2) = 0$. The first-degree aggregated subtree potential is then calculated as follows:

$$\begin{aligned}
p_1(A_1) &= \text{base}(A_1) + \text{base}(A_2) + \text{base}(B_1) = 1.0, \\
p_1(A_2) &= \text{base}(A_2) = 0.5, \\
c_1(A_1) &= \text{base}(A_2) = 0.5, \\
c_1(A_2) &= 0, \\
P'_1(C_A) &= (p_1(A_1) - c_1(A_1)) + (p_1(A_2) - c_1(A_2)) = 1.0.
\end{aligned}$$

For the general case, the process is similar. Let $S_n(u)$ denote the full n-level subtree rooted in u. For each immediate recursion edge $u \rightarrow v$ the nodes in $S_n(u) \cap S_n(v)$ would be counted twice. The correction term for u is therefore the sum of all nodes in these intersections, which will be subtracted when aggregating the result. Because of the tree structure, all such intersections are mutually exclusive, implying that no node can be subtracted twice. We also note that each intersection is itself a full subtree of degree strictly less than n. This implies that the sum of the nodes in $S_n(u) \cap S_n(v)$ has already been calculated as the potential of degree $(n - \text{distance}(u, v))$ for u, where $\text{distance}(u, v)$ is the number of invocation edges on the path between u and v. These principles are illustrated in an example in figure 5.1. Formally the correction term can be written as

$$c_n(u) = \sum_{u \rightarrow v \in R_n} p_{(n - \text{distance}(u,v))}(v)$$

where R_n is the set of recursion edges $u \rightarrow v$ such that $\text{distance}(u, v) \leq n$.

The corrected aggregated subtree potential of degree n for code path A becomes

$$P_n(A) = \sum_{A_i \in A} p_n(A_i) - c_n(A_i)$$

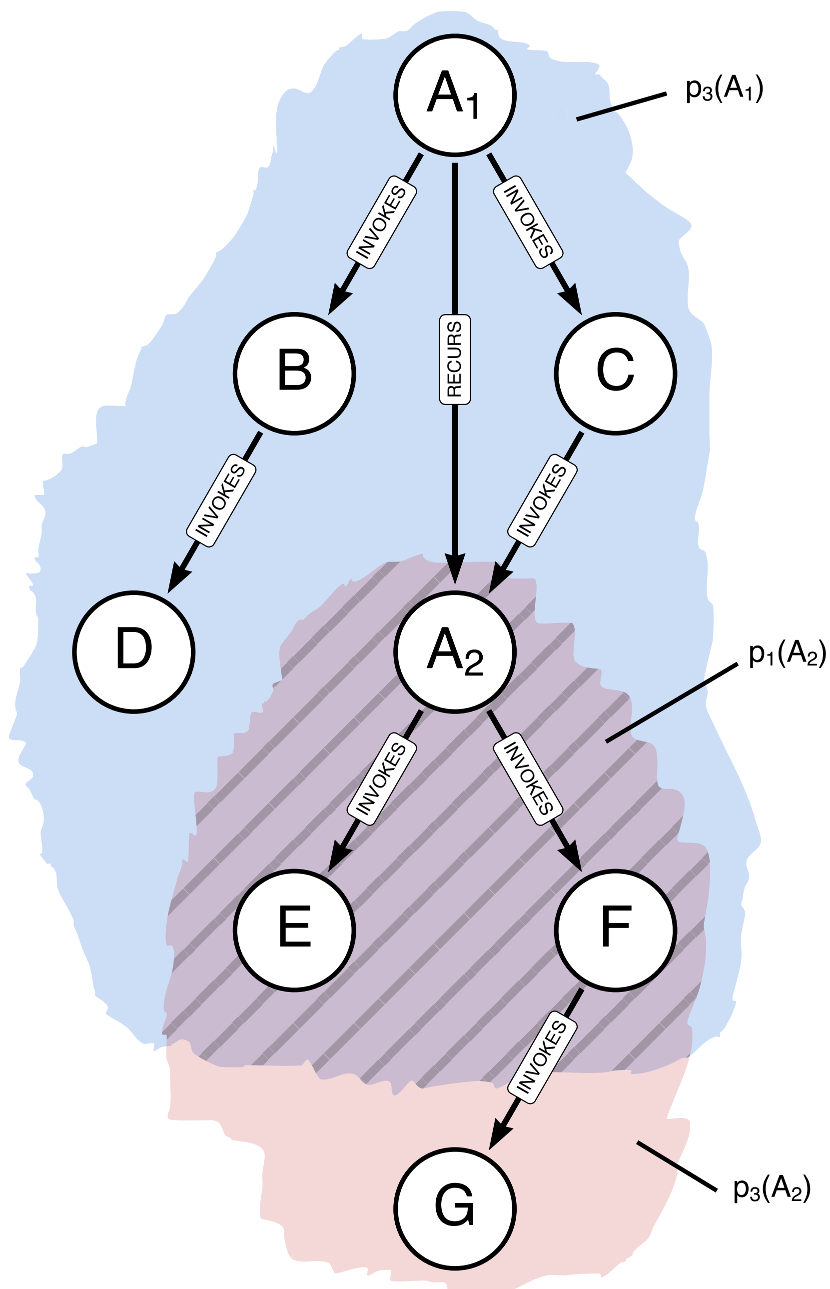


Figure 5.1: When aggregating full 3-level subtrees rooted in code path A, some nodes are included twice because the subtrees intersect due to recursions. The aggregation is corrected by subtracting the intersecting nodes from the final result. In this particular example, the correction term for the intersecting nodes is $c_3(A_1) = p_{3-2}(A_2) = p_1(A_2)$. The aggregated result is $P_3(A) = p_3(A_1) + p_3(A_2) - p_1(A_2)$.

where

$$\begin{aligned} p_0(u) &= \text{base}(u), \\ p_n(u) &= \text{base}(u) + \sum_{v \in \text{children(u)}} p_{n-1}(v), \\ c_0(u) &= 0, \\ c_n(u) &= \sum_{u \to v \in R_n} p_{(n-\text{distance}(u,v))}(v). \end{aligned}$$

Cypher Implementation

One of the benefits of storing the data in a graph database is that the above aggregations can be expressed declaratively with Cypher. The first step is to precalculate the potential and correction terms for each node in some forest from the database. These values will be stored in two array properties, p and c respectively. We begin by initializing p_0 and c_0 which is done by the following Cypher query:

```
MATCH
    (f:Function)
SET
    f.p = [f.base],
    f.c = [0.0]
```

This query is self-explanatory. In the real implementation we also have a few `WHERE` statements to limit the functions to a selection of profiles. The remaining potential values are calculated by running the following query multiple times with incremental values for n:

```
MATCH
    (u:Function)
OPTIONAL MATCH
    (u)-[:INVOKES]->(v:Function)
WITH
    u,
    u.base + sum(v.p[$n-1]) AS potential
SET
    u.p = u.p[..$n] + [potential]
```

This query first matches all functions u in the graph, and their immediate children v. The potential is calculated according to the above recursive definition and finally appended to the array property p. Similarly, the correction term is calculated with the following Cypher query:

```
MATCH
    (u:Function)
OPTIONAL MATCH
    (u)-[r:RECURS]->(v:Function)
WHERE
    r.distance <= $n
WITH
    u,
```

```
        sum(v.p[$n - r.distance]) AS correction
    SET
        u.c = u.c[..$n] + [correction]
```

This query matches, for all functions u, the functions v whose subtrees would intersect with the subtree rooted in u. The correction term is calculated as the sum of the nodes in these intersections according to the above recursive definition.

The final step is to perform the aggregation.

```
    MATCH
        (f:Function)
    WITH
        f.codePath AS codePath,
        sum(f.p[$n] - f.c[$n]) AS potential
    ORDER BY
        potential DESC
```

This calculates $P_n(C_i)$ for each code path i and returns an ordered list containing the code path with the highest potential at the top.

5.3 Bottleneck Correlation

A simple intuition about bottlenecks is that they limit the performance of the executing program, giving rise to an inverse dependence between throughput and time spent in the bottleneck. Going back to the Pareto principle, we can assume that there exist a vital few functions that are inhibiting the performance of a program. In the simplest case, the program is structured as a single branch where no node possesses a relationship to more than a single child. Here, the performance of the program would be entirely decided by the base samples collected throughout the branch, most commonly amassed in the leaf node. Over time, this leaf node would display an almost perfect correlation with the execution time, varying only due to randomness in sampling. This correlation implies that the trace ending in the leaf node could be a "problematic" part of the code. Unfortunately, introducing more complexity in the tree structure will diminish the correlation between bottlenecks and execution time, as various functions contribute to the limiting factor. The principle that a limiting function exhibits a relationship to the execution time still holds. In many cases this results in a higher correlation than most trivial functions. An important distinction here is that correlation does not imply causation, meaning that just because a function correlates it does not imply that it is a bottleneck. It does however highlight branches of promise that can be subject to further investigation. Note that we make an assumption here, that the most problematic bottlenecks will exhibit a somewhat linear correlation with the execution time as they have a very direct impact. This could be tweaked in the future by instead looking at some non-linear measure of correlation, e.g. Spearman's correlation coefficient.

A perk of looking at correlation is the flexibility in analyzing diverse data. Since each function will be aggregated separately for each profiling, the function will only be considered in the executions where it is present. Therefore, one can include various benchmarks from different programs without impacting the result of a function only present in a few pro-

grams.

An aspect to keep in mind when looking at correlation for a given function over multiple benchmarks is the various environments it appears in. Some benchmarks may run relatively few other functions, while some benchmarks run sizeable amounts of other time-consuming functions. This attribute will affect the position and progression of the sampling points. A separate correlation must be calculated for every benchmark to assure fairness. Finally, an average correlation can be calculated to represent the overall pattern. Below, in figure 5.2, is an example of two different benchmarks with corresponding sampling points for a function. If we ignore the diversity in the benchmarks and attempt to fit a correlation on all of these sample points, it would become more or less a random distribution. Instead, we handle these benchmarks individually and calculate a mean correlation to find the most promising candidates. It would also be possible to use another aggregation method such as max depending on the purpose of our search. All sample points have been normalized, again to adjust for fairness. Without normalization, a longer-running benchmark would command more importance.

In appendix A.2, we provide the Cypher code for calculating the Pearson correlation coefficient for each unique function in a set of benchmarks.

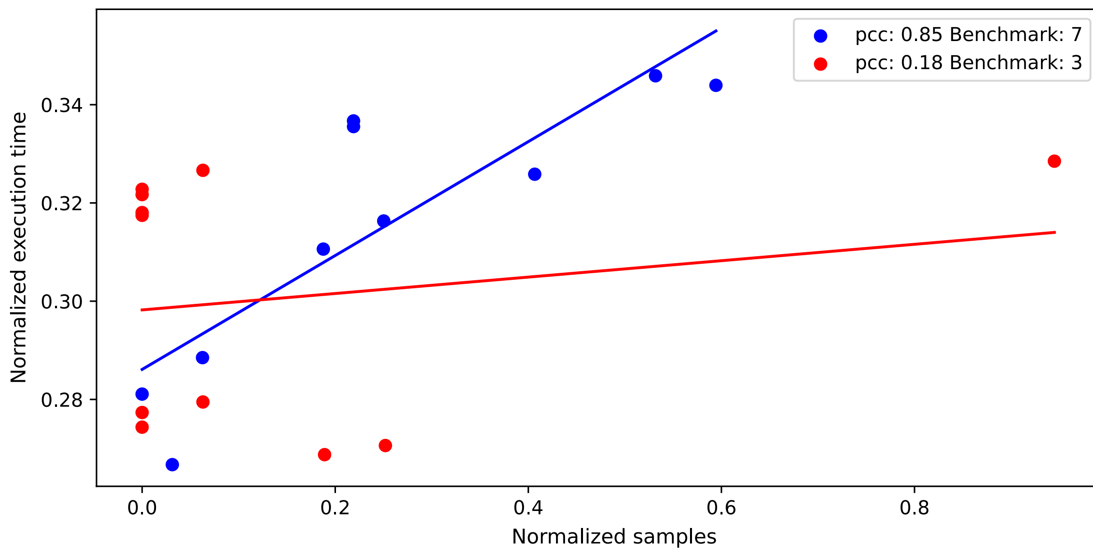


Figure 5.2: An example of how the samples of a function on the x-axis may correlate with the execution time on the y-axis. The function has a significantly higher correlation in benchmark 7 than in benchmark 3. The score(f) = $(0.85 + 0.18)/2$ in this case, will be compared with other functions in any of the specified benchmarks. The lines in the image are for visualization purposes.

The correlation score calculated for a given function f that is present in m specified benchmarks is calculated through:

$$\text{score}(f) = \text{avg}(\text{pcc}(X_1, Y_1), \ldots, \text{pcc}(X_m, Y_m)),$$

where X_i is the set of observed samples for f per profile in benchmark $i = 1, \ldots, m$ and Y_i is the corresponding execution time. Note that the number of profiles per benchmark n_i may vary, but the benchmark is only included if $n_i \geq 2$. This threshold can optionally be set to a greater number by the user.

Chapter 6

Detecting Regressions

In this chapter, we explore methods for regression analysis. This entails identifying a set of candidate functions or contexts that may be the root cause of a given regression.

6.1 Time Series Analysis

One way to detect regressions is to look at how the profile results change over time. Neo4j already do this to detect specific benchmarks that take longer to run than usual, by comparing the current execution time with previous mean values. This is however only performed on the overall execution time of the profile and gives no information about which particular functions may have caused the regression. Instead, this is done by manually inspecting the flame graphs generated by the profiler. These flame graphs contain all the data necessary to perform automated analysis on a function level, which is the topic of this section.

The basis of this analysis is the intuition that individual functions should be sampled approximately the same number of times every time the benchmark is run, within some error margin. We calculate the simple moving average and moving standard deviation for each function in a benchmark, based on previous observations of this function in the same benchmark. Each function is then assigned a score depending on how many standard deviations it deviates from its previous average. That is:

$$\text{score}_n(x_0) = \frac{x_0 - \text{avg}(x_1, x_2, \ldots, x_n)}{\text{std}(x_1, x_2, \ldots, x_n)}$$

where n is the number of previous observations to consider, x_0 is the number of total samples at the current point in time T, x_1 is the value at time $T - 1$, and so on. The value $n \geq 2$

can be chosen arbitrarily. However, we found that $n = 10$ works well for most of our data. We use total samples in our implementation to capture information from a larger calling context. It would also be possible to use base samples. However, in practice, we found that this produced a result biased towards the leaf set. Finally, it would be possible to use the precalculated corrected *potential* values from 5.2 which would act as a middle-ground. If either total samples or potential is used, it is important to note that we also have to account for recursions to avoid counting the same value twice. For total samples, this means excluding all nodes with an incoming recursion edge. In Cypher this is done by adding a `WHERE` clause as follows:

```
MATCH (f:Function)
WHERE NOT EXISTS (()-[:RECURS]->(f))
```

The full query is available in appendix A.1.

The score will be high for functions that are usually stable and have increased by a large amount compared to previous observations. This is illustrated in figure 6.1. By calculating the score for each function and presenting it in an ordered list, we get an overview of candidate functions that are worth exploring further.

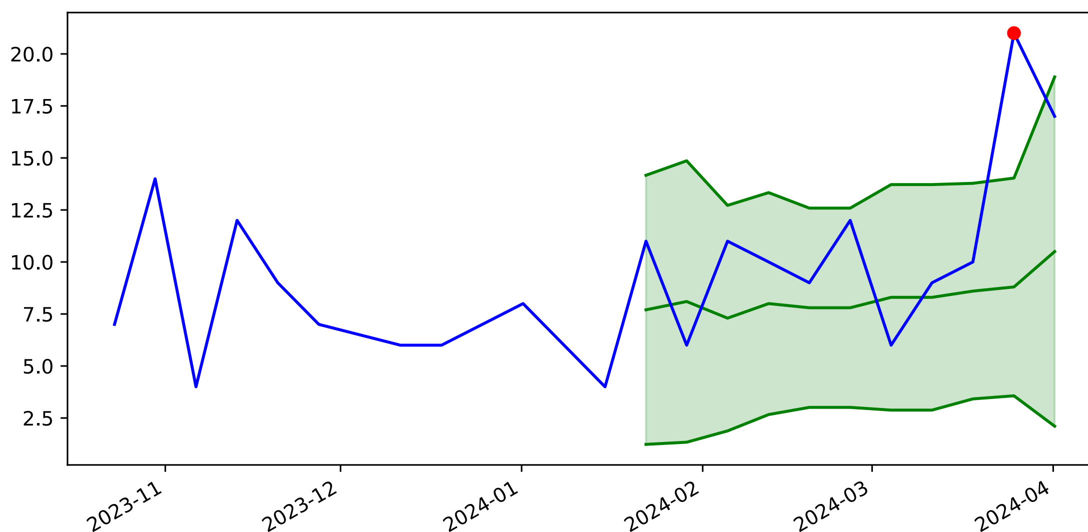


Figure 6.1: Bollinger plot illustrating the scoring function for a specific function. The blue line shows how the observed value changes over time. The middle green line is the moving average over ten previous observations. The green area contains all values that would deviate less than two standard deviations from the moving average. The regression point, marked in red, has a score of 4.7 standard deviations and is therefore outside this area. From experience, we found that two standard deviations usually capture the variation expected from sampling noise but any value could be used.

6.2 Regression Correlation

Another way to identify candidates is to look at the correlation between the aggregated samples of a function and the total execution time of the benchmarked program. The same principles as when detecting bottlenecks in section 5.3 apply. A function that fluctuates in tune with the total execution time is likely to have an inverse dependence on the throughput.

Specifically for regressions, a dramatic change in samples will overpower all the other sampling time points and somewhat overlook their discrepancies, which will skew the results to single out the functions that have high extreme values during the time point of the regression.

The advantage of correlation, as opposed to time series analysis, is that the trees can come from various benchmark groups and settings. A common situation for the benchmarking team is when reviewing the past week of regressions, various programs of different types have dropped significantly in performance. The suspicion is that some common code or library has changed, which impacts all of them. With time series analysis, one would have to single out each benchmark, perform the detection and manually attempt to find commonalities between the identified candidates. With correlation, on the other hand, each function will only be compared to the respective execution time of its running program. Our correlation implementation benefits from large quantities of data, as the results become more reliable, and is capable of potentially identifying the common functions that cause the regression in any number of benchmarks. Note however that a given function that only appears in a single benchmark and has a substantial correlation over time, may end up higher on the candidate list than any function that appears in all the specified benchmarks.

6.3 Expanding Algorithms

Once a set of suspect candidates has been identified, the contexts of their appearances draw focus. An increase in total samples, causing a function to show up as a candidate for the time series analysis, may be due to worse performance by itself or it may be exhibiting abnormal behavior as a symptom of a proximate issue:

- An ancestor has increased the number of invocations of the path that includes this candidate.
- One or multiple children have increased in samples, meaning there has been a code change that negatively impacts their execution time.

However, we can not trivially expand every possible parent and child path since the search would explode in time complexity, especially when looking at large call-stack forests with many occurrences of the code path in different contexts in each tree.

As the path is expanded, multiple branches may be considered with each step. We choose which branches to pursue or discard, based on the behavior of the initial guess u in each considered option. The expansion is carried out in a recursive manner where the prospectiveness metric of the node u is continuously updated as the path grows.

The prospectiveness of a child path: $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n$ is calculated by looking at the historical sampling of $u|u \rightarrow v_1|v_1 \rightarrow v_2|\dots|v_{n-1} \rightarrow v_n$, meaning u in the context of calling v_1, which in turn calls v_2 and so forth. For example, in figure 4.2 if we wished to expand the children of A_1 with time series analysis and we have ten previous trees with the same structure but varying sample properties at each node in the database, the score of the path 'A_1/B_1' would be calculated through a standard score of 'A_1/B_1' in the regression against equivalent branches in the previous trees. The question of whether the path would be worth exploring could be reformulated as "Given that A_1 calls B_1, how much has A_1 deviated from its normal behavior?".

Algorithm 1 Expansion

```
Require: branch: Some branch rooted in u
  procedure EXPAND(branch)
      if prospectiveness(branch) ≤ some threshold then                ▷ Stop criterion
          return ∅
      else if max depth reached then
          return branch
      candidates ← selectExpansions(any extensions of branch)   ▷ Selecion Algorithm
      if candidates = ∅ then
          return branch
      expansions ← ∅
      for c ∈ candidates do
          expansions ← expansions ∪ EXPAND(branch ∪ c)
      return expansions
```

In Algorithm 1, the procedure for expanding out from an initial node is shown. This algorithm was inspired by the *zooming* algorithm by Ammons et al. [1], but differs by offering more flexibility. Both the stop criterion and selection algorithm are customizable, keeping the expansion relevant for both time series analysis and correlation, as well as any future insights. Note that the stop criterion here is only a simple example of how it could be implemented. The selection algorithm, as previously mentioned, keeps the expansion from blowing up in time complexity, its purpose is to ignore branches that appear irrelevant to the cause of the detected abnormality or correlation in the initial node and instead focus on those branches that have a high promise of explaining it.

Max depth is specified by the user and restricts the search, which produces quicker results and keeps the algorithm from trailing off to distant relatives of u if the stop criterion should be too relaxed. A max breadth can also be implemented in the selection method to narrow down the search. An argument for these parameters is that the stop criterion may not be easy to formulate so that it fits all benchmarking data. Capping the search is easy enough and can be quickly adjusted if the results are found to be unhelpful or too packed with information to make out the relevant parts.

6.3.1 Greedy Selection

With our most basic selection algorithm pertaining to time series analysis, we make the following assumption: When traversing the tree either up or down, starting from a given initial node u, as soon as we come across a child or parent node v_n that causes u in this context to receive a negative score, the current path can be terminated and the results discarded. A negative score implies that u in the context of being called or calling the path up until and including the node v_n, has accumulated fewer samples in the regression tree than what is expected from its past behavior. The name *greedy* however, comes from the mechanic that it greedily selects to pursue the $n \leq$ max breadth most promising candidates. This mechanic only takes score into consideration and fails to consider other properties that might indicate that this could be an interesting candidate.

6.3.2 Custom Selection

Algorithm 1 can easily be utilized with a *custom selection* method. *Selection* is implemented as an abstract class in Python and provides all necessary functions for Expansion to work. *Greedy selection* leaves a lot to be desired, which can efficiently and modularly be implemented with a *custom selection*.

As an example of the power of this structure, we implemented a *custom selection* called *informed selection*. *Informed selection* originates from *greedy* and applies to regression detection with time series analysis. It makes the same initial ordering based on the deviation score of each branch, but then it broadens its perspective and considers some additional factors:

Git change recency walks the git repository and collects data about the latest commit for each file containing a specific code path. It then produces a weight w_t for each code path by calculating an exponential decay of the recency of the latest commit time t:

$$w_t = e^{-t*\lambda}, \quad \text{where } \lambda = \frac{\ln 2}{t_{1/2}}.$$

The half-life $t_{1/2}$ was set to 15 days, as regressions caused by code committed further back should have been detected earlier.

Compile modes look at the mode of compilation for each considered candidate extension. Perhaps one of the children of the branch has consistently been compiled with Just-In-Time, but during the regression behaved abnormally by being Interpreted. The mode of compilation is determined by the JVM and can heavily influence the execution time. The benchmarking team at Neo4j has previously observed significant changes caused by irregular compile modes.

Depth of the branches looks at the distribution of depths where the branch has appeared in previous trees. It considers how many new depths have been encountered in the regression tree and promotes those branches that have behaved abnormally. For example, this aspect catches if a method has been called an extra time in a recursion, compared to previous behavior.

6.3.3 Correlation Selection

Similarly to *greedy selection*, *correlation selection* performs a simple mechanism and chooses the top n potential extensions. In contrast to *greedy* however, it possesses no deviation score to evaluate. Instead, it looks at the measure of how much a given branch correlates with the execution time of the entire program. Just like with the initial correlation selection, an average Pearson correlation coefficient is calculated over all of the specified benchmarks.

The default threshold for the stop criterion when exploring a subtree is trivially set to 0, meaning that we stop exploring as soon as we consider expanding with a candidate that causes the current branch to have a non-positive linear correlation with the execution time. Essentially, this would imply that the samples and execution times possess no linear relationship whatsoever. A more influential parameter is the user-provided max breadth and max depth.

Another interesting means of exploring could be to reverse the usual selection and instead look at negative correlations. A negative correlation could imply that spending more time on this particular function results in faster execution. There could be many explanations as to why such a relationship would appear. One explanation is that a neighboring function is a bottleneck or regression and during faster executions, this candidate function has been given, proportionally, more time. Another explanation that we identified during testing was that some functions involved in query optimization had a negative correlation, as more time spent in those methods results in overall faster execution.

Chapter 7

Evaluation

In this chapter, we will evaluate the usefulness of our proposed framework. The goal of this evaluation is to provide organizations looking to implement something similar with insights to help them in their decision-making process. We will weigh the pros and cons of our framework and alternative solutions which may or may not be more appropriate depending on the specific requirements of the organization.

7.1 Flexibility of the Framework

The database schema, proposed in 4.1, proved both flexible and efficient for a variety of analysis methods. Following is a summary of the ones that we implemented to demonstrate what is possible:

- The *subtree potential* implementation in section 5.2 demonstrates that schema can be used for non-trivial data aggregation methods. This implementation also highlights why a separate relationship for recursions is necessary; without it, the query would be both more complex and less efficient.
- The implementation in section 6.1 demonstrates how time series analysis methods can be run directly on the database level. It is also possible to query time series data to perform local analysis or to create visualizations: This is in fact what we did to create figure 6.1. This implementation also makes use of the recursion relationship.
- The correlation based implementations in sections 5.3 and 6.2 demonstrates how statistical methods are supported by the framework.
- The *expanding algorithms* implemented in section 6.3 demonstrates how search algorithms are supported by the framework. In these implementations, we make multiple

queries to the database to iteratively expand a subtree. In many cases, it is probably more efficient to query a larger part of the graph and perform the search locally, which is also possible.

7.2 Cypher vs Local Analysis

When implementing our methods we tried to put as much logic as possible in Cypher queries. This turned out to be a good decision for the following reasons:

- **Expressiveness:** Cypher's declarative syntax allowed us to focus on *what* we wanted to calculate, rather than exactly *how* to calculate it. We found that many of our Cypher queries showed a close resemblance to the mathematical expressions that we wanted to implement. The resulting queries are concise, yet efficient because the Cypher planner performs many optimizations without any additional effort from our part.

- **Memory limitations:** Neo4j's extensive benchmarking suite collects tens of gigabytes of data each week. Performing aggregation on large data sets locally presents a few challenges if the data does not fit in memory. The benefit of using a Neo4j database for these purposes is that it will automatically read chunks of data from a disk into a page cache to avoid storing too much data in memory while at the same time allowing fast random access to frequently used data.

- **Portability:** Because our work was explorative, we used Python for our implementations to allow us to iterate quickly with a short feedback loop. However, because most of the calculations are done with Cypher queries it will be easy for the Benchmarking team at Neo4j to port our methods into Java code when bringing this into production.

One drawback of using a database compared to local analysis is that some algorithms may be slower if they make many queries to the database. However, in most cases, it is possible to fetch all the required data from the database into memory before running the algorithm. This also has the benefit that it is possible to specify exactly which data is needed which could greatly reduce the amount of data that needs to be transferred over the network.

7.3 Changes to the Benchmarking Pipeline

Keeping the database up to date with the latest profiling data may add some complexity to the benchmarking pipeline. In some methods, such as time series analysis, only profiles from a single benchmark are considered and it is feasible to import the required data before processing begins. In other methods, such as aggregation methods and queries searching for a specific benchmark, it is required that all relevant data be present in the database before performing the analysis. A few weeks of data may result in hundreds of millions of nodes and the import takes several hours to run. In these cases, it is better to import new profiling data incrementally as it becomes available by running the importer after each benchmark. It is also a good idea to periodically prune the database of old data to prevent it from growing indefinitely — one is often only interested in the most recent runs of each benchmark anyway. Maintaining an up-to-date database adds a step to the benchmarking pipeline but the

advantage is that analysis can be run on demand in a matter of seconds.

7.4 Use Cases at Neo4j

This section describes a few use cases that we identified by collecting feedback and ideas from the engineers at Neo4j. Our database importer tool will be integrated into Neo4j's Benchmarking pipeline to support some of these use cases. These use cases are all made possible by the use of our database schema.

Queries Targeting Multiple Flame Graphs

Several engineers expressed the desire to be able to run queries on data from multiple flame graphs. One such query that we heard multiple times was: "I want to find all benchmarks where a given function took more than x % of the total samples for the benchmark". This was also one of the core use cases which we used to design the database schema. This query will be one of the first use cases that is implemented in Neo4j's benchmarking user interface. Similar queries are easy to express in Cypher, and the Benchmarking team can now conveniently run these queries against our database.

Analyzing Subrees and Partial Stack Traces

In our early implementations, we discovered that looking at single functions can be misleading because the seemingly worst functions are often simple ones that are called many times. On multiple occasions, we also received feedback from engineers that single functions are rarely a problem by themselves, and rather it is the context in which they are called that is of interest. Our database schema allows much flexibility for analyzing calling contexts where several functions are involved. Cypher's built-in ability to match paths between function nodes makes it easy to perform aggregations on partial stack traces instead of single nodes. The pre-calculated *potential* properties, described in section 5.2, make it possible to query subtrees of a specific shape. The `RECURS` relationship makes it convenient to handle the special cases that arise when a function calls itself.

Assigning Regressions to the Correct Team

Identifying a set of suspect candidate functions also helps with deducing what people and resources to deploy. It is commonly not the benchmarking team that tackles the detected issues, and coordinating efforts from the wrong team is both costly and time-consuming. Even when our program does not produce the exact function that caused a regression, it can indicate an appropriate team to assign. Looking at the code path of a candidate function often reveals the name of the team that maintains it, in which case, assigning the correct team becomes a trivial task.

7.5 Presentation of Data

Presenting the data in an interpretable manner is a crucial requirement of our product. Through several iterations of providing developers with our output, and receiving constructive feedback, we developed an interactive output format, in which, the user can selectively display more specific details of the results. Appendix B is an example of such an output produced from running the regression detection program with time series analysis. Other algorithms produce a similar output but with corresponding metrics. The output is highly customizable through user-provided arguments and aims to convey dense information intuitively. In contrast, the very first iteration consisted of a single large list of candidates without any explanation or guidance, which was received as confusing and overwhelming. Just as the output format was developed iteratively, it will likely continue to evolve as more teams adopt the framework and progressively expose new requirements.

The output as it is presented today, has already been of some use to the company when identifying causes of regressions and assigning the correct team.

Chapter 8

Conclusion and Future Work

8.1 Conclusions

In this thesis, we proposed a framework for exploring large sets of profiling data by making queries against a Neo4j graph database. Our proof of concept application successfully demonstrates the possibilities of the framework by implementing a data-importing tool and a few different analysis methods. Our importing tool will be integrated into Neo4j's benchmarking pipeline and some new user interfaces that make queries based on our schema are planned.

A Neo4j graph database is a good choice for modeling call-stack trees for performance analysis. It allows the analyst to write powerful aggregation queries for large sets of data that would not fit in memory. The built-in ability to pattern-match on paths between multiple nodes in the graph is very useful for analyzing calling contexts.

We proposed a simple yet powerful database schema, and our implementations show that this schema can be used for both time series analysis and aggregation methods. An organization may use our framework to build a set of analysis tools tailor-made for their specific requirements. We received much feedback from engineers at Neo4j, and the consensus is that high flexibility in the initial selection, e.g. the ability to filter on a specific set of benchmarks, is crucial. Another important component is that the result presented to the user is easy to interpret and does not overwhelm the user.

One potential downside of the framework is the added complexity of importing data because an extra step has to be integrated into the benchmarking pipeline. On the other hand, using a database reduces the complexity when handling datasets too large to fit in memory. In conclusion, the framework may be excessive for small sets of profiling data but provides many benefits otherwise.

8.2 Future Work

Our database can used to query data needed for local analysis methods such as machine learning. Such methods are out of scope for this thesis but could be explored further in future research. For example, Xie et al. [11] propose a visual analytics framework for anomaly detection in call-stack structures. They utilize a One-Class Support Vector Machine with a custom *stack2vec* embedding that could be applied to the call-stack trees in our database.

Another suggestion for future work is to implement a reverse functionality to what we do today, which would collect data from our database to produce a flame graph. This would allow for the creation of flame graphs from filtered or aggregated data. For example, it would be interesting to produce an average flame graph from historical data and subsequently create a differential flame graph between the current flame graph and this historical average.

Yet another future topic could be to integrate heap allocation for our selection methods, or as a completely new feature. The async-profiler is already capable of collecting such measurements and heap allocation could potentially expose previously shrouded issues. With small modifications, our database schema could be adjusted for such analyses.

The effectiveness of our implemented methods may be amplified by the use of wall-clock time profiling instead of CPU time profiling, which has a larger coverage. A future endeavor to explore such differences would be beneficial to deciding what mode of profiling to run.

References

[1] G. Ammons, J. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 172–196, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[2] async profiler. async-profiler repository. `https://github.com/async-profiler/async-profiler`. Accessed: March 12, 2024.

[3] B. Gregg. The flame graph : This visualization of software execution is a new necessity for performance profiling and debugging. *Queue - Cloud Debugging*, 14(2):91 – 110, 2016.

[4] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Addison-Wesley professional computing series. Addison-Wesley, 2 edition, 2021. pp. 187 - 189.

[5] IBM. Sample-based profiling. `https://www.ibm.com/docs/en/mon-diag-tools?topic=perspective-sample-based-profiling`. Accessed: April 2, 2024.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[7] G. N. Tornetta. On the algebraic properties of flame graphs, 2023.

[8] Neo4j, Inc. Cypher manual. `https://neo4j.com/docs/cypher-manual/5/introduction/`. Accessed: April 3, 2024.

[9] Neo4j, Inc. Neo4j documentation. `https://neo4j.com/docs/`. Accessed: April 5, 2024.

[10] Neo4j, Inc. Neo4j python driver manual 5. `https://neo4j.com/docs/python-manual/current/transactions/`. Accessed: April 8, 2024.

[11] C. Xie, W. Xu, and K. Mueller. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE Trans. Visual. Comput. Graphics*, 25(1):215 – 224, 2019.

Appendices

Appendix A

Cypher Queries

A.1 Time Series Analysis

```
MATCH
    (fn:Function)
WHERE
    fn.profileKey IN $profile_keys AND
    NOT EXISTS (()-[:RECURS]->(fn))
MATCH
    (run:TestRun)--(profile:Profile)
WHERE
    profile.key = fn.profileKey
WITH
    fn.codePath AS codePath,
    run.date AS date,
    profile.key AS key,
    sum(fn.totalSamples) AS samples
ORDER BY
    date DESC
WITH
    codePath,
    collect(key) AS keys,
    collect(samples) AS samples
WITH
    codePath,
    CASE keys[0] WHEN $profile_keys[-1] THEN samples[0] ELSE 0 END AS
   candidate,
    CASE keys[0] WHEN $profile_keys[-1] THEN samples[1..] ELSE samples END
```

```
    AS historic
WITH
    *,
    CASE size(historic) WHEN 0 THEN 0 ELSE apoc.coll.sum(historic) /
    $window_size END AS mean
WITH
    *,
    CASE WHEN size(historic) > 0
        THEN sqrt(
            (apoc.coll.sum([x IN historic | (x - mean)^2]) +
            ($window_size - size(historic)) * mean^2) / ($window_size - 1)
        )
        ELSE 0
    END AS std
RETURN
    codePath AS code_path,
    CASE WHEN std > 0
        THEN (candidate - mean) / std
        ELSE 0
    END AS score
ORDER BY
    score DESC
```

A.2 Correlation

```
MATCH
    (fn:Function)
WHERE
    fn.profileKey IN $profile_keys
MATCH
    (profile:Profile)--(benchmark:Benchmark)
WHERE
    profile.key = fn.profileKey AND
    profile.metricMean IS NOT NULL
WITH
    profile.metricMean AS metricMean,
    fn.codePath AS codePath,
    benchmark,
    sum(fn.baseSamples) AS samples
ORDER BY
    samples DESC
WITH
    codePath,
    benchmark,
    collect(samples) AS x,
    collect(metricMean) AS y
WHERE
    size(x) >= $leastOccurrences
```

```
WITH
    *,
    apoc.coll.avg(x) AS xhat,
    apoc.coll.avg(y) AS yhat
WITH
    *,
    apoc.coll.sum([
        i in range(0,size(x) - 1) | (x[i] - xhat) * (y[i] - yhat)
    ]) AS nominator,
    sqrt(apoc.coll.sum([
        xi in x | (xi - xhat) ^ 2
    ]) * apoc.coll.sum([ yi in y | (yi - yhat) ^ 2])) AS denominator
WITH
    *,
    CASE WHEN denominator > 1e-8 THEN nominator / denominator ELSE 0 END
   AS pcc
RETURN
    codePath AS code_path,
    collect(benchmark) AS benchmarks,
    collect(pcc) AS pccs,
    avg(pcc) AS avg_pcc,
    collect(x) AS x,
    collect(y) AS y
ORDER BY
    avg_pcc DESC
```

Appendix B

Program Output

Result

▼ Help

This document is a summary of the code paths that changed the most during the reported regression. These code paths deviated enough to become probable sources to the regression and are referred to as candidates.

Each candidate code path has the following information:

- **Expected:** The historical average number of samples for the candidate code path. This value is based on a number of previous test run observations, specified by `window_size`.
- **Actual:** The observed number of samples at the time of the reported regression.
- **Diff:** The difference between Actual and Expected.
- **Score:** The difference between Actual and Expected, normalized by the standard deviation of historical observations. The score will promote code paths which are usually stable, but have deviated significantly during the reported regression.
- **Status:** The status column may have the following values:
  - `+` The code path is observed for the first time.
  - `-` The code path has been observed historically, but not during the reported regression.

Plot: Each candidate code path also includes a plot of the historic values and the current actual value with the regression date marked in red. These plots also display the moving average based on `window_size` (default 10), previous observations and a lower and upper bound, 2 standard deviations from the moving average. For intuition, a candidate with a score of 2.0 would appear excactly on the upper band. These bands are commonly referred to as Bollinger bands.

Each candidate code path may also include a few call stack contexts which may be of interest. These call stack contexts can be viewed by expanding the Child/Parent collapsed sections. The call stack contexts have been explored in an iterative manner where a single code path has been added at each iteration and the new call stack discarded or further pursued. Each row displays the values produced from evaluating the call stack so far.

For **Child Traces**, the candidate code path is displayed at the bottom of the table. Each row above represents the stack trace between the call stack on this row and the candidate code path at the bottom. For example, if the three bottom rows are `read`, `parse`, `run` (see example table below) and we are looking at the candidate code path `run`, then the row `read` represents the entire stack trace `run;parse;read`, i.e. `run` calls `parse` which in turn calls `read`. The "candidate" diff/score on the row `read` is then the diff/score value for `run` given that it calls `parse` and that `parse` calls `read`.

Trace	Diff	Candidate Diff	Candidate Score	Status
readChar	1.0	1.0	1.0	
read	1.0	1.0	1.0	
parse	1.0	1.0	1.0	
run	1.0	1.0	1.0	

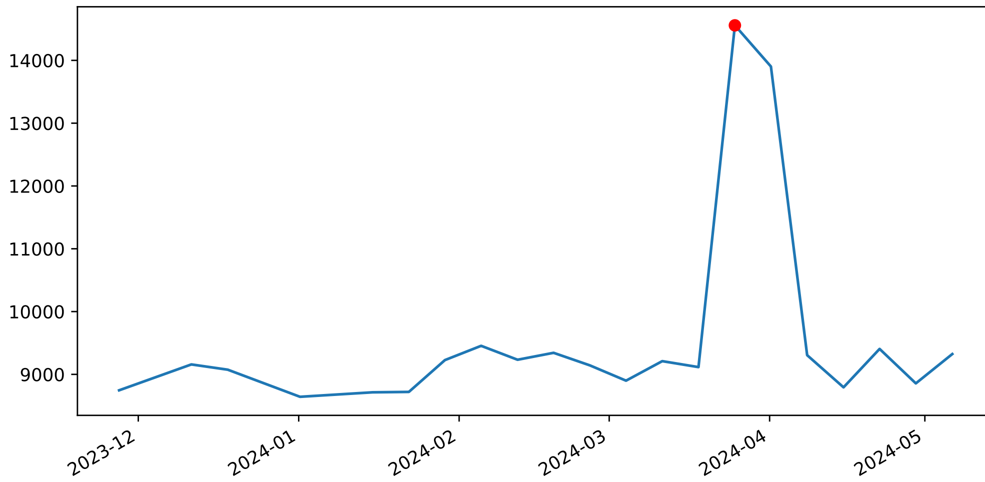
For **Parent Traces**, the candidate code path is displayed at the top of the table. Each row below represents the stack trace between the call stack on this row and the candidate code path at the top. For example, if the three top rows are `read`, `parse`, `run` (see example table below) and we are looking at the candidate code path `read`, then the row `run` represents the entire stack trace `run;parse;read`, i.e. `read` is called by `parse` which in turn is called by `run`. The "candidate" diff/score on the row `run` is then the diff/score value for `read` given that it is called by `parse` and that `parse` is called by `run`.

Trace	Diff	Candidate Diff	Candidate Score	Status
read	1.0	1.0	1.0	
parse	1.0	1.0	1.0	
run	1.0	1.0	1.0	
main	1.0	1.0	1.0	

Parameters

- Benchmark group: `accesscontrol`
- Benchmark name: `Q29_(deployment,EMBEDDED)_(execution_mode,EXECUTE)_(host,m5d.large)_(jdk,17)_(planner,DEFAULT)_(runtime,INTERPRETED)_(store_forma`
- Git Branch: `dev`

Overview



Candidate 1

Code Path	Expected	Actual	Diff	Score	Status
scala/collection/immutable/ArraySeq$ofRef.unsafeArray	0.1	7	6.9	21.8197	

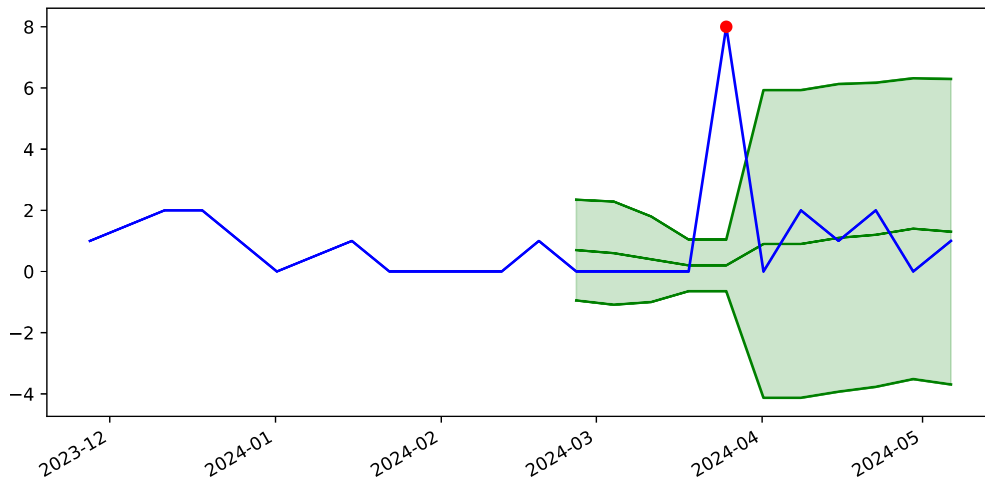
▶ Plot

▶ Parent Traces

Candidate 2

Code Path	Expected	Actual	Diff	Score	Status
scala/collection/mutable/ArrayBufferView.	0.2	8	7.8	18.4993	

▼ Plot



▼ Parent Traces

Trace	Diff	Candidate Diff	Candidate Score	Status
scala/collection/mutable/ArrayBufferView.	7.8	7.8	18.4993	
scala/collection/mutable/ArrayBuffer.view	7.8	7.8	18.4993	
scala/collection/mutable/ArrayBuffer.view	7.6	7.8	18.4993	
scala/collection/IndexedSeqOps.reverseIterator	9.3	7.8	18.4993	
scala/collection/IndexedSeqOps.reverseIterator$	9.3	7.8	18.4993	
scala/collection/mutable/ArrayBuffer.reverseIterator	9.3	7.8	18.4993	
<proprietary>	99.8	7.8	18.4993	
<proprietary>	143	7.8	18.4993	
<proprietary>	321.8	7.8	18.4993	
<proprietary>	321.8	7.8	18.4993	
<proprietary>	321.8	7.8	18.4993	

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-05-30

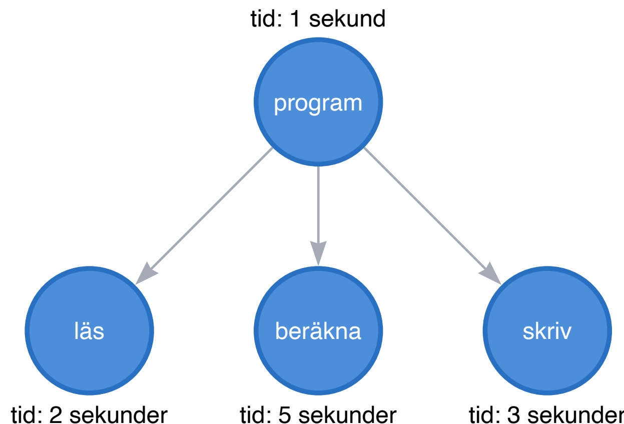
EXAMENSARBETE Modeling Profiling Data in a Graph Database for Performance Analysis
STUDENT Richard Lundberg, Marcus Rettig
HANDLEDARE Jonas Skeppstedt (LTH), Simon Priisalu (Neo4j), Jaroslaw Palka (Neo4j)
EXAMINATOR Per Andersson (LTH)

Modellering av profileringsdata i en grafdatabas för prestandaanalys

POPULÄRVETENSKAPLIG SAMMANFATTNING **Richard Lundberg, Marcus Rettig**

Att identifiera långsam kod i mjukvara är en viktig, men tidskrävande uppgift. Vi undersöker hur profileringsdata kan modelleras i en grafdatabas för att underlätta processen att identifiera flaskhalsar och regressioner.

Programkod brukar delas in i mindre bitar, så kallade *funktioner*. Tiden det tar att köra hela programmet är summan av hur mycket tid som spenderas i varje funktion. För att optimera programmet vill man därför hitta de mest tidskrävande funktionerna och försöka förbättra dessa.



Genom att köra ett program med ett så kallat *profileringsverktyg* kan man göra mätningar kring hur funktioner anropar varandra och hur mycket tid som spenderas inuti dessa. Resultatet brukar kallas för *profileringsdata* och kan visualiseras i en trädliknande struktur, till exempel som i bilden. På företaget Neo4j görs hundratals sådana mätningar varje dag vilket resulterar i tusentals träd, där varje träd består av tusentals funktioner.

Vi har undersökt hur man kan analysera stora mängder profileringsdata genom att lagra den i en grafdatabas. De mest populära databaserna just nu representerar data i tabellformat, men en grafdatabas lagrar istället data som noder och relationer. Till exempel kan noder representera personer i ett socialt nätverk och relationerna kan representera vänskap mellan två personer. En grafdatabas passar även utmärkt för de trädliknande strukturer som beskrevs ovan eftersom sådana strukturer är problematiska att representera i tabellformat.

I en grafdatabas kan användaren ställa frågor, till exempel: "I vilka testkörningar tog denna funktionen mer än två sekunder?", eller "Hur lång tid tog denna funktionen idag jämfört med förra veckan?". Svaret på den första frågan kan identifiera funktioner som ofta presterar dåligt och därför påverkar prestandan för hela systemet, så kallade flaskhalsar (eng. bottlenecks). Svaret på den andra frågan kan identifiera funktioner som går långsammare idag än de gjorde för en vecka sedan, så kallade regressioner (eng. regressions).

Vårt arbete visar att en grafdatabas är ett lämpligt verktyg för att modellera profileringsdata. Vår föreslagna databasstruktur går att använda för flertalet analysmetoder och ökar tillgängligheten till data insamlad över tid.