MASTER'S THESIS | LUND UNIVERSITY 2017

Multi-threaded execution of Cypher queries

Ragnar Mellbin, Felix Åkerlund

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-19

Multi-threaded execution of Cypher queries

(in Neo4j)

Ragnar Mellbin
ragnar.mellbin.498@student.lu.se

Felix Åkerlund
felix.akerlund.978@student.lu.se

September 8, 2017

Master's thesis work carried out at Neo Technology, Inc..

Supervisors: Per Andersson, per.andersson@cs.lth.se
Johan Svensson, johan@neotechnology.com

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

In this report we investigate parallel execution of queries in graph databases. We analyse different methods of parallelization, how to introduce query parallelization to a graph database, which query operations that are suitable for parallelization and if we can improve the execution time of a single query. We do this by designing and implementing a parallel runtime for the Cypher query language in the graph database Neo4j, but many of the design ideas and operators investigated are applicable to any graph database.

We focus on increasing performance for a select few operators, while still being fully integrated with Neo4j. We take much inspiration from a design called morsel-driven parallelism. This means that we strive to split the workload into many small pieces, "morsels", and then hand these morsels to the threads executing the query. This is in contrast to a more classical parallelization approach, where you split the workload into a few big parts of equal size.

We conclude that the operators best suited for parallelization are the operators that can be split into several smaller parts, where each part can be computed independently. We successfully introduce parallel execution of Cypher queries to Neo4j and by doing so we increase the performance of a single query by up to 15 times under certain conditions.

Keywords: Neo4j, Cypher, query, graph database, parallel execution, parallel query, parallel database, openCypher

Acknowledgements

We would like to thank all the developers at Neo Technology, and our supervisors Johan Svensson and Per Andersson, for their help and support during the writing of this thesis.

Special thanks to Andrés Taylor for his willingness to discuss the design of Cypher runtimes and to share his technical expertise, Alex Averbuch for sharing his knowledge about LDBC and benchmarking and for helping us to get our tools working and pointing us in good directions, Chris Vest for answering a multitude of easy and difficult questions about anything related to the Neo4j kernel, Henrik Nyman and Mats Rydberg for their help and support, Mattias Finné for helping us break Neo4j in just the right ways and finally Anton Persson for sharing his first-hand wisdom about writing a thesis at Neo Technology.

Contents

1 Introduction **9**

1.1 Problem statement . . . 9

1.1.1 Purpose . . . 9

1.1.2 Scope . . . 10

1.2 Contribution . . . 11

1.3 Related work . . . 11

1.3.1 Volcano operator model parallelism . . . 12

1.3.2 Morsel-driven parallelism . . . 12

1.3.3 Parallel query execution in PostgreSQL . . . 13

1.3.4 Parallelism in Oracle Database EE . . . 13

2 Approach **15**

2.1 Theory . . . 15

2.1.1 Relational databases . . . 15

2.1.2 Structured Query Language . . . 16

2.1.3 Graph databases . . . 16

2.1.4 The Cypher query language . . . 17

2.1.5 openCypher . . . 17

2.1.6 Transactions . . . 17

2.1.7 LDBC . . . 18

2.1.8 JMH . . . 19

2.2 Neo4j . . . 19

2.2.1 Cypher in Neo4j . . . 19

2.2.2 Neo4j Cypher query execution plan . . . 19

2.2.3 Neo4j query operations . . . 20

2.2.4 Neo4j Cypher runtime . . . 20

2.3 Choosing query operation . . . 21

3 Architecture **23**

3.1 The parallel Cypher runtime . . . 23

3.1.1 Morsel-driven parallelism . . . 23
3.1.2 The ”parallel pipe builder” . . . 24
3.1.3 The ”morsel pile” . . . 24
3.2 Implementation . . . 25
3.2.1 Initial operator . . . 25
3.2.2 Additional operations . . . 26
3.2.3 Implemented pipes . . . 26
3.2.4 All nodes scan . . . 27
3.2.5 Expand all . . . 27
3.2.6 Filter . . . 28
3.2.7 Sort . . . 28
3.2.8 Produce results . . . 29
3.3 Discussion . . . 30
3.3.1 Expand operators . . . 30
3.3.2 Choosing query operation . . . 31
3.3.3 Store access in operators . . . 31
3.3.4 Synchronization in the morsel pile . . . 31
3.3.5 Attempt without transaction splitting . . . 32
3.3.6 Limit operator . . . 32
3.3.7 Moving to a common thread pool . . . 32

4 Evaluation 33
4.1 Measure of performance . . . 33
4.2 Queries . . . 33
4.2.1 LDBC-inspired queries . . . 33
4.2.2 Small queries . . . 35
4.3 Dataset . . . 37
4.3.1 LDBC-SNB dataset . . . 37
4.3.2 Small queries dataset . . . 37
4.4 Hardware and tools . . . 40
4.4.1 Hardware . . . 40
4.4.2 Reference Neo4j version . . . 40
4.4.3 numactl . . . 41
4.4.4 JMH . . . 41

5 Evaluation results 43
5.1 LDBC results . . . 43
5.2 Small Query 1 (scan) results . . . 45
5.3 Small Query 2 (filter) results . . . 46
5.4 Small Query 3 (expand) results . . . 47
5.5 Small Query 4 (sort) results . . . 49
5.6 Small Query 5 (mixed) results . . . 50

6 Conclusions 53
6.1 Query performance . . . 53
6.1.1 LDBC query performance . . . 53
6.1.2 SQ query performance . . . 53

6.2 Designing for expand operators 54
6.3 Limitations 54
6.3.1 Limited operator support 54
6.3.2 Multiple transactions per query 54
6.3.3 Synchronization 55
6.4 Conclusion 55
6.5 Future work 55
6.5.1 Multi-threaded transaction support in Neo4j 56
6.5.2 Expand-centred design 56
6.5.3 Additional query operations 56

Bibliography **57**

Appendix A Evaluation results **63**

Chapter 1
Introduction

The Cypher query language is looking to become the standard for graph database queries. Neo Technology, the company behind the graph database Neo4j, is the original creator of Cypher and it is in their interest to show the capabilities of the query language as well as of the database. Today all Cypher queries in Neo4j are executed sequentially and thus not able to fully utilize a multi-core system, in the case where there is only a single agent running queries.

We start by introducing our problem statement in section 1.1. In section 1.2 we specify the authors' contribution and in section 1.3 we present previous work done is this area.

1.1 Problem statement

One typical use case of a database system consists of multiple small queries, for example a social network site being used by a large number of people simultaneously. The database serves as a backend for a system or service, which is constantly updating and retrieving small amounts of data. In scenarios like this, Neo4j is already able to utilize the full potential of a multi-core machine by letting different threads serve different queries, and thus achieving parallelism.

An analytical use case however, where a user typically runs a few or a single very time-consuming query on a huge dataset, differs. Neo4j currently does not have any capabilities to utilize multiple processing units for a single query, and is thus unable to fully utilize a multi-core machine in this use case.

1.1.1 Purpose

Taking the above into account, we theorize that it is possible to shorten the execution time of a single Cypher query when a Neo4j instance is used in an analytical fashion. We plan to

achieve this by making use of parallel execution of different parts of the query on modern multi-core machines.

We aim to answer the following questions:

1. Which graph database query operations are suited for parallel execution?

2. How can parallel execution of Cypher queries be introduced to Neo4j?

3. Can we increase query performance by introducing parallel execution of Cypher queries in Neo4j?

1.1.2 Scope

We limit the scope in certain areas due to the time constraints of this project. We work with the assumption that the database is accessed in a read-only analytical fashion, which means that we will not have to consider the effects of concurrent writes to the database. This also limits the amount of changes we will have to do in the Neo4j kernel, which currently does not support multi-threaded access in the same transaction.

The changes needed in the Neo4j kernel to allow for multi-threaded access in a transaction, safe from writes from other transactions are large enough to warrant another project in itself. This mainly due to the fact that it has been designed from the ground up to only support one thread per transaction, with all the assumptions and simplifications that this fact entails.

Additionally, we choose to only support execution plans that are non-branching, i.e. plans without joins, Cartesian products, apply operators, etc. Again, this is due to time constraints.

Additional goals

If time allows, the implementation will be improved in one or more of the following manners:

- Make the new implementation function on a non read-only database without introducing corruption or other errors.

- Determine the degree of parallelism used for executing the query based on the number of computing units the system has available.

- Determine the degree of parallelism used for executing based on the size and/or characteristics of the current dataset.

- Make the query planner take the possibility of parallelism into account when planning a query.

- Parallelize all operators required for one or more Linked Data Benchmark Council (LDBC) queries.

1.2 Contribution

Both of the authors of this thesis have very similar skill sets and experience. Ragnar had prior experience of working with Neo4j. Most of the report and almost all of the implementation was done together as a pair. Notable exceptions include:

- Felix implemented the parallel sort operator.
- Ragnar implemented the final rewrite of the parallel expand operator, but the authors had implemented it several times together prior to this.
- Felix designed the dataset and queries used for the evaluation, as well as implemented a generator for the dataset.
- Ragnar implemented the JMH benchmarks and the scripts to run them.
- Felix wrote large parts of chapter 4.
- Ragnar plotted the result graphs used in chapter 5.
- Ragnar wrote most of chapter 6.

1.3 Related work

Parallel databases are not in any way a new phenomenon, the idea started getting popular in the early 1990s [1]. Since then, the focus has shifted slightly, from parallelism between machines to parallelism on the same multi-core machine, but many of the general principles still apply.

In section 1.3.1 we present the Volcano operator model style parallelism, whose parallelism principles still are utilized in many modern Database Management Systems (DBMS) [2]. We also introduce the more recent morsel-driven parallelism model in section 1.3.2. In section 1.3.3 we present the recent work on parallelism done in PostgreSQL and in section 1.3.4 we present the parallel execution in Oracle Database Enterprise Edition.

All of these describe techniques utilized for parallelization in relational databases. While there are many differences between relational databases and graph databases, many of the ideas and techniques can be applied to graph databases, at least to some extent.

We were inspired by the work of Robert Haas that we described in section 1.3.3. He chose to work with the store scan, similar to the all nodes scan in Neo4j. Initially his implementation was similar to the common Volcano-inspired approach described in section 1.3.1 with an exchange operator (which he named `Gather`) interfacing the parallel scan to other operators. Later in development the implementation was changed to let the split parallel pipelines execute several steps before gathering, akin to the architecture described in section 1.3.2. When designing the parallel Cypher runtime we borrowed several concepts from the morsel-driven parallelism outlined in 1.3.2. Not only did the idea itself sound like it had merit, it also seemed to have worked very well for the developers of PostgreSQL as mentioned in 1.3.3. Yet another thing to take into consideration is that Oracle Database EE, which uses a more traditional work-splitting method, has a tendency to suffer from skewed workloads when left unconfigured (see section 1.3.4).

1.3.1 Volcano operator model parallelism

Volcano is the name of a query processing system developed at the University of Colorado [3]. In the Volcano model, queries are expressed as algebra expressions, where the operators are algorithms for query processing. All operators are iterators, and feature anonymous inputs, meaning that they do not care which kind of other operator is feeding them.

In order to introduce parallelism to the Volcano design, the *exchange operator* was invented. This operator can be inserted anywhere in a query tree without otherwise modifying its structure or making other operators aware of the change.

For example, imagine two operators, A and B. We insert an exchange operator X between A and B. Normally A would call B for input, but now A will call X, which in turn calls B. When the exchange operator X is called, it creates multiple instances of B. The B operators work as normal while their input is distributed between them by a support function.

Volcano also features vertical parallelism, which is when a producer and consumer run in parallel. The exchange operator handles the transfer of data packets from the producer to the consumer so that neither needs to be aware of the parallelism.

1.3.2 Morsel-driven parallelism

In [2], Leis et al. present an approach called *morsel-driven parallelism*, where the operators are kept largely unaware about parallelism and shared state is avoided, which differs from the common Volcano-inspired model. With morsel-driven parallelism, work is divided into several small parts (morsels) and dynamically distributed between threads. The threads take the morsel as far through the pipeline of operators as possible, stopping only where a point is reached where synchronization with other threads is needed. For example, a morsel of nodes, might be scanned from the store, then filtered or otherwise processed in the same pipeline. But, for example, when the results are to be sorted or aggregated the pipeline has to end since the operation is dependent on the rest of the intermediate result set.

Certain operations can increase or decrease the size of the result set. In order to make sure that this does not lead to skewed morsel sizes, the data is repartitioned into equally-sized morsels between pipelines of operators, instead of retaining the same morsel boundaries.

The degree of parallelism can change dynamically at any time (any morsel pipeline boundary) by using a different number of threads to work with the morsels. For example, this means that the system can at one point allocate all of its resources to a single query and then reallocate the resources if another query arrives before the first is finished.

This approach is new and not widely adopted in database query execution. It can be significantly harder to implement, and if a perfect split of the workload is possible it leads to increased overhead without any tangible benefits.

1.3.3 Parallel query execution in PostgreSQL

Through a series of blog posts, the PostgreSQL developer Robert Haas has outlined his work with parallel query execution in PostgreSQL. Under the course of several years he worked with altering the infrastructure of PostgreSQL to allow for local worker threads to be spawned and to allow for dynamic shared memory for the intermediate results [4].

The first operation to get parallelized was the sequential scan, which returns all entries in the database. Two different approaches were considered for this operation, ”fixed chunk” where the store is split into equal partitions numbering the same as the number of workers. The other is ”block by block”, where the store is split into numerous smaller blocks and the worker threads scan them one by one. The second approach was chosen due to better load balancing when the different worker threads worked at different speeds [5].

The implementation of the node scan introduced two new query operations, `Parallel Seq Scan` and `Gather`. The `Parallel Seq Scan` is executed in parallel on the worker threads, and `Gather` gathers the result before passing it on to the next operator. Performance testing showed good scaling for the first few worker threads, but dropped off quickly for higher numbers [6]. Later work added several other parallel operations making use of the `Gather` operation [7].

1.3.4 Parallelism in Oracle Database EE

Oracle Database Enterprise Edition plans parallel operators in a way similar to PostgreSQL 1.3.3. Parallel operators are planned as part of the query plan, and at the end of a parallel subtree is a coordinator operation. Operators that contains buffering of results are marked as such in the query plan.

The database does not utilize ideas similar to the morsel-driven parallelism described in 1.3.2. Instead it determines a degree of parallelism, and utilizes that set number of workers. This number has a tendency to be too large for a given system and query if left at its default value. This approach also means that the system is susceptible to the problems caused by skewed workloads between the parallel threads, where the system has to wait for the slowest worker before continuing with the sequential part of the query [8, 9].

Chapter 2
Approach

Our goal is to introduce parallel computing to parts of the the Neo4j query execution for the purpose of decreasing query execution time for single queries. This work consists of analysing methods of parallelization, identifying which query operation(s) to parallelize and implementing multi-threaded versions of said operations. This should bring performance benefits in terms of decreased query execution time for certain workloads, and will hopefully encourage further work in this area. The results will be evaluated by comparing the query performance of the multi-threaded implementation to the original one.

Before and during implementation of the different parts of our work we used a whiteboard to sketch, define and coordinate our visions of the design. We worked as a pair when implementing and used test-driven approach for our work.

We present the theory and terminology used in this project in section 2.1 and in section 2.2 we present Neo4j. In section 2.3 we describe our process for choosing which query operator to work with initially.

2.1 Theory

This section describes relevant terms and technologies in order to understand our work and how it differs from previous work. These include relational databases (2.1.1), Structured Query Language (2.1.2) graph databases (2.1.3), Cypher (2.1.4), openCypher (2.1.5), transactions (2.1.6), LDBC (2.1.7) and JMH (2.1.8).

2.1.1 Relational databases

Relational databases, also called *relational database management systems* (RDBMS), are based on the *relational model* of data introduced in 1970 by Edgar F. Codd [10]. The relational model organizes data into tables, where rows represent entities and columns represent attributes. Each table contains one or more columns that make up the *primary*

key. A primary key is an set of attributes that is used to identify an entity, and must therefore be unique. A table can also contain *foreign keys*, attributes that make up the primary keys of another table.[11]

Imagine a database containing a table of movies and another table of actors. A movie can feature many different actors, and an actor can star in many different movies. Data entries that have many-to-many relationships like this require extra effort in the relational model. Representing the relationships between movies and the actors starring in them with only these two tables would be incredibly cumbersome. Instead, a third table is introduced, known as a *junction table*, which stores the actor-film combinations. These three tables can then be stitched together by *join operations*, if for example we want to look up horror movies featuring blond actors [11, 12].

2.1.2 Structured Query Language

Structured Query Language (SQL) is a language used to modify or extract information from relational databases. SQL commands are structured like sentences similar to English to communicate with the database [11]. An example of a typical SQL query can be found in example Query 2.1.

```sql
SELECT name FROM Actors WHERE birthyear = 1977
```

Query 2.1: An SQL query that returns a list of names of actors born in 1977.

2.1.3 Graph databases

Graph databases differ from relational databases by representing relationships between entities as explicitly stored first-class citizens, meaning that individual relationships can be referenced by key-value pairs just like objects or other items in the database. This allows for greatly improved performance of queries that in RDBMS would rely heavily on join operations, as they can be computed by simply following the relevant relationship pointers. Each entity is represented by a *node* or *vertex* in the graph, while relationships take the form of *edges*. It is possible for both vertices and edges to feature attributes. Querying the database to explore relationships between entities is done by following the paths between them, and does not involve any kind of matching or join operations. The many-to-many relationships that require extra work in RDBMS are natively supported by graph databases [13].

In order to provide some of the beneficial features of relational database schemas, some graph databases support labels and schema indices. A label is like a tag that can be pinned to an entity or relationship. This makes it possible to index entities under a shared label, with the requirement that they all have at least one attribute in common. Labels can thus be seen as a parallel to tables in relations database, with properties instead of columns [14].

2.1.4 The Cypher query language

Neo Technology has created the Cypher query language, which is similar to SQL but built for graph databases. Cypher has been designed to be intuitive and easy to read. Its syntax features ASCII art to make nodes and relationships look like their graphical representations [13].

For example, if we take a look at Query 2.2, entities are encapsulated in parenthesis to look like nodes. Square brackets are used for relationships, surrounded by hyphens to make them look like edges, sometimes with a greater-than or less-than sign which indicates direction. Attributes can be specified inside curly braces.

```
MATCH (john {name: 'Johan'})-[:friend]->()-[:friend]->(fof)
RETURN fof.name
```

Query 2.2: A simple Cypher query, returning friends of friends for the node with ”Johan” set as name property.

2.1.5 openCypher

openCypher is a project initiated by Neo Technology that aims to deliver a full and open specification of the Cypher query language. It is supported by several large actors in the database sector, such as Oracle and Databricks [15].

The project aims to deliver four types of artefacts:

Cypher language specification A technical expression of the Cypher language to support generation of parsers. The Cypher language specification is licensed under the Creative Commons license.

Cypher reference documentation User documentation for the Cypher query language, including tutorials and examples.

Reference implementation A fully functional implementation of key parts of the stack needed to support Cypher, licensed under the Apache 2.0 licence.

Technology certification kit Used for self-certification purposes, the technology certification kit consists of a number of tests to use in order to test the support for a given version of Cypher.

The source for the openCypher project is available on GitHub at `https://github.com/opencypher`.

2.1.6 Transactions

A database *transaction* is a term used to denote sequences of instructions that perform read and/or write operations on the database. In order to protect the integrity of the

database, transactions must have the following properties: *atomicity*, *consistency*, *isolation* and *durability* (ACID) [12].

Atomicity ensures that a transaction is either performed to completion, or not at all. Consistency makes sure that transactions always leave the database in a valid state, meaning that no rules or constraints are violated. Isolation requires that transactions produce the same results when executed simultaneously as they would have done sequentially. Durability means that all changes made by a completed transaction must persist, even in the event of a system crash or loss of power [12].

2.1.7 LDBC

The Linked Data Benchmark Council started as an EU project in 2012 with the goal of developing industry-strength benchmarks for graph DBMSes and other data management systems. It is a joint open-source effort between academic researchers and actors from the industry to create a industry-neutral entity developing and maintaining benchmarks as well as auditing results [16].

LDBC is now an independent non-profit organization sponsored by IBM and Oracle Labs and it is sustained by its members. LDBC maintains two benchmarks with different sizes and workloads, the Semantic Publishing Benchmark (SPB) and the Social Network Benchmark (SNB)[17]. These are available under the GNU general public license v3 at the LDBC GitHub repository at `https://github.com/ldbc`. Besides the benchmarks themselves LDBC maintains several tools for generating the needed data and executing the workload against a database [18].

LDBC Social Network Benchmark

LDBC's Social Network Benchmark (LDBC-SNB) is a collection of three different workloads designed to run on datasets produced by the LDBC-SNB Data Generator (DATAGEN). The purpose of DATAGEN is to provide realistic datasets with properties that mimic those of social networks such as Facebook [19]. Synthetic data sets are potentially more useful than the sources they are modelled after, as real data sets often comes with privacy concerns and portability issues that make them more difficult to obtain and handle. DATAGEN allows users to scale the dataset to their particular needs by specifying a target size for the uncompressed CSV (Comma Separated Values) data. A feature to note is the deterministic nature of the data generated; a given set of input parameters will always result in the same dataset regardless of the machine used [20].

The datasets generated by DATAGEN have been found to share a large number of properties with real life social network graphs in an analysis published in 2014 [21].

LDBC-SNB Business Intelligence Workload

The Business Intelligence workload is made up of complex queries meant to analyse user behaviour for marketing purposes. As a result, the benchmarks focus highly on the performance of query execution. The workload scales with the size of the database [22].

2.1.8 JMH

The Java Microbenchmark Harness (JMH) is a tool for building, running and analysing benchmarks targeting the JVM. It is developed and maintained by openJDK/Oracle [23].

The tool aims to help developers produce and analyse benchmarks while avoiding several common benchmarking pitfalls such as loop optimizations, dead code elimination, constant folding and inlining [24, 25, 26].

2.2 Neo4j

Neo4j is an open source graph database developed by Neo Technology, Inc. that features native graph storage and processing. The database is mostly written in Java, partly Scala, and the entire source code is hosted at `https://github.com/neo4j/neo4j`. Neo4j is currently the most popular graph database management system according to the database ranking website DB-Engines [27]. The popularity score is derived from the number of online mentions and frequency of related Google searches.

A major version of Neo4j is released roughly every six months, minor versions are released whenever needed (due to bugfixes, etc.) and commits to the GitHub repository are continuously made as a part of the ongoing development.

2.2.1 Cypher in Neo4j

Neo4j supports the Cypher query language, and performance comparisons have been made between different ways of accessing Neo4j. In a study published in 2013, Holzschuher and Peinl conclude that ”[...] Cypher is a promising candidate for a standard graph query language, but still leaves room for improvements.” [28]. When they revisited the subject in 2016 they found that while there had been improvements to Cypher, it still performed significantly worse than e.g. native access in some scenarios. They were not able to completely confirm their hypothesis that Cypher is the best graph query language, but they point out that many indicators point in that direction [29].

2.2.2 Neo4j Cypher query execution plan

When a database server is tasked with a query, there are often a number of different ways of performing the requested operations. The server will attempt to find the fastest way to execute the query on the database, which is represented by what is called an execution plan or a query plan. This plan contains the database operations to be performed and the dependencies between them.

Finding the optimal query plan for a given query is an NP-complete problem [30]. The Neo4j cost query planner avoids this problem by utilizing a technique called Iterative Dynamic Programming (IDP). IDP can be seen as a combination of dynamic programming and greedy algorithms which produces optimal plans for simpler queries and good plans for more complex ones [31].

2.2.3 Neo4j query operations

The Neo4j query operations can be divided into six different categories:

Starting point operators are used to find the starting points of the query. This is achieved by either a scan of nodes in the database, an index seek for nodes or a combination thereof.

Expand operators explores the graph by expanding graph patterns, following relationships to and/or from the nodes. Examples include expanding all relations from a node, or finding the path between two given nodes.

Combining operators are used to piece together other operators, optionally with filters.

Row operators transforms an incoming set of rows to another set of rows. Distinct (filtering of duplicates), filtering, count, limit and sort are all examples of row operations.

Update operators updates the graph. `CREATE`, `DELETE`, `MERGE` and `CREATE CONSTRAINT ON` are examples of actions that use these operators.

Shortest path operators computes the shortest or all shortest paths between given nodes.

In order to view the query plan for a query in Neo4j the `EXPLAIN` and `PROFILE` query prefixes can be used [32].

2.2.4 Neo4j Cypher runtime

The default Neo4j Cypher runtime, named the Interpreted runtime, is implemented in Scala. It is a straightforward translation from a Cypher plan to something that access the database and produce results. After a logical plan is produced, it is used to construct an execution plan. The execution plan is a tree of pipes matching the logical plan, see Figure 2.1 for an example. In a tree without eager operators, the sections of the tree only produce a row when asked for one by their parent. This is to avoid unnecessary work, but some operators (e.g. aggregation, eager) need to process all rows available to them before returning a row to their parent.

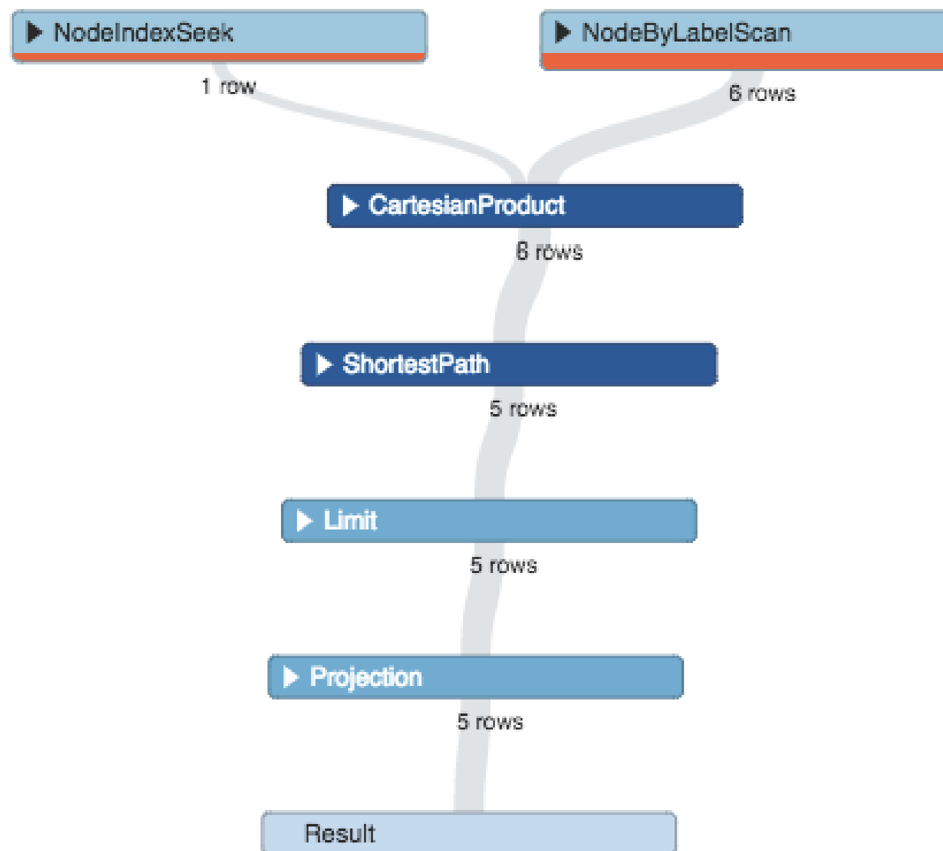

Figure 2.1: An example of a Neo4j execution plan, with an index seek and a label scan as leaves, and the produce results operator as root.

These pipes does not contain any state. This is so that the tree of pipes can be reused instead of rebuilt if the database is tasked with the same query again. These cached trees are discarded if the structure and distribution of the underlying data changes beyond a certain threshold, as the optimal plan for the query might have changed.

2.3 Choosing query operation

In order to answer the first point of our purpose in section 1.1 and by doing so choosing what query operation to start working with we held a discussion with several Cypher developers from Neo Technology. We focused on finding an operation that:

- Was viable to implement given the time available for this project.
- Was deemed to be a costly operation, at least for some queries.

- Had an internally independent workload, i.e. an implementation would require minimal inter-thread communication.

Initially we planned to benchmark a number of queries in order to see which query operations were costly. However, we were unable to find a reliable way to get this information.

Chapter 3
Architecture

In this chapter we describe the architecture of our solution. In section 3.1 we present the design decisions for our implementation. In section 3.2 we describe the interesting parts of the implementation details and lastly in section 3.3 we highlight some of the trade-offs made when building our solution.

3.1 The parallel Cypher runtime

We implemented multi-threaded execution of queries by implementing a new Cypher runtime in Neo4j, henceforth called the parallel runtime. This section gives an overview of the design of the parallel Cypher runtime, and the implementation is outlined in section 3.2.

3.1.1 Morsel-driven parallelism

As mentioned in section 1.3 we used many concepts from the morsel driven parallelism when implementing our parallel cypher runtime. There are several merits to this approach. It is easier to account for operators such as expand or filter, which can drastically change the workload size. In a tradition work-split scenario this can potentially leading to skewed amounts of work for the different threads, where some might finish much sooner than others. Using the morsel design ideas also allows us to process the morsels on-demand, and thus not producing all rows of the query if the user only asks for the first few, but instead only keeping a few morsels ready for reading. Related to this is the fact that, in many cases, we can consume results at the same time as we produce them using this scheme. The exception to this is when operators which needs all rows before it can start providing results are involved. Sorting is an example of this, as it needs to too see all rows it is to sort before being able to supply any rows to its child operators.

Yet another argument for a morsel-driven approach is that in some cases it can be hard to estimate the size of the query result. This leads to problems for a simple work splitting approach as we would have to split a workload of unknown size into equal parts. With the morsel-driven approach we avoid this problem by simply building fixed size morsels on demand as more results are requested.

3.1.2 The ”parallel pipe builder”

Whereas the Interpreted Cypher runtime (section 2.2.4) produces a tree structure of pipes, the parallel runtime produces a tree structure of something we named Parallel pipe builders. These generally follow the same principles as the pipes in the Interpreted runtime. They are produced as the logical plan is parsed and they are stateless and reused in the same way.

The usage differs, however. A single method call on a pipe from the Interpreted runtime returns an iterator of rows ready to use. For a parallel pipe builder you first call a method to build a ”morsel pile” consisting of morsels of Cypher pipes (further described in section 3.1.3), then you ask this pile for its single ProduceResults-pipe (this is always the case for the topmost morsel pile, other morsel piles can contain multiple pipes) which is then executed in order to retrieve the result iterator.

The parallel pipe builders also contain the functionality to create additional pipes to be added to an existing morsel pile. This is used when the number of source rows is increased while running the query, e.g. when the query contains expand operators.

3.1.3 The ”morsel pile”

The morsel pile is the central collection of morsels for a given query. These morsels in turn consists of a number of cypher pipes, which applies their respective operator as a part of the row-producing pipeline (see Figure 3.1). This means that the morsel pile is the central collection of work items for the query. In the best case scenario a query only has one morsel pile. In this pile each morsel represents a small chunk of the rows to be processed. When a morsel is processed by a thread it will pass thorough all the pipes of the query, and the rows will be altered accordingly.

There can also be several morsel piles for a single query. This is the case when the morsel can not pass through all of the pipes independently of each other. If the query plan contains operators that cannot be computed independently, or increases the number of rows in the working set, breaks are introduced in the pipeline. These breaks split the pipeline into segments, and each of these segments have their own morsel pile. These morsel piles are connected with monitors, which differ depending on the operation. These monitors are responsible for synchronizing the data between the morsel pile producing results and the one that wants to consume them. For example, sorting is an operation that breaks the pipeline, since no sorted results can be produced until all rows have been sorted. Thus the monitor for sorting operations blocks threads asking for sorted morsels until all results have been sorted. The morsel pile following a sort monitor makes sure that the morsel results are supplied to the following monitor in the correct order.

The morsel pile has a method for fetching a new morsel for execution if there is one available. For morsel piles that are in the middle of the pipeline the morsel pile that pro-

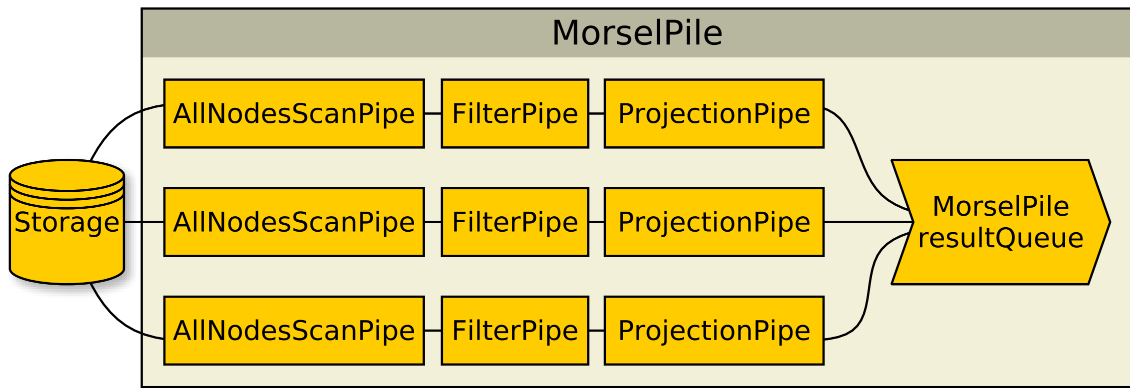

Figure 3.1: An example of a typical morsel pile, featuring three pipelines running in parallel.

vides results to the connecting monitor needs to produce some results before this pile can return a morsel. In cases like this the morsel pile will delegate the call for another morsel to that pile. The behaviour of this method varies between the morsel pile implementations in order to satisfy the requirements of that specific part of the pipeline. For example, a morsel pile following a sort operation will not return any of its own morsels before returning all morsels that belong to the morsel pile with the sorting operation. The morsels of the pile following the sorting pile would be unable to get any sorted results from the monitor before all entries has been sorted.

The morsel pile also provides functionality for adding new morsels to itself and piles that depend on it. This is needed for operations like expand that increase the amount of intermediate results, and thus also increases the amount of morsels.

3.2 Implementation

We implemented the Parallel Cypher runtime in Java, as opposed to Scala which was used for the Interpreted runtime. The primary motivation for this choice was our familiarity with Java and Java parallelism combined with comparatively lacking experience in Scala.

One of the challenges with the parallel Cypher runtime is also one of its biggest features; it is completely integrated with Neo4j and the Cypher planner. This has allowed us to test it by running end-to-end queries in Neo4j. It also means that we have been able to make a fair comparison of the performance compared to the default Interpreted runtime by simply running queries with the two runtimes.

3.2.1 Initial operator

Looking back on 1.1, we worked with the Cypher team to reason about which operator to work with. Given the pipeline nature of the Cypher runtime in Neo4j, we concluded that we wanted to begin with a starting point operator. The reasoning behind this is that since we will want to produce and consume multiple rows in parallel, it would make sense to start with the most simple operation producing a large amount of independent rows. This

would allow us to start processing the data in parallel immediately. Another benefit of the all nodes scan is that we know the maximum amount of results from the start, and thus also the amount of morsels that the operator can produce.

Despite the fact that *all nodes scan* is not necessarily common in well-optimized queries on well-structured datasets we still choose to work with it. It is a very costly operation, as it involves traversing the entire dataset, and its use is unavoidable for certain queries.

Another point speaking in favour of our choice is that the split workloads for all nodes scan are completely independent. Simply scanning nodes from the database is not in any way dependent on what is found in other parts of the database by other threads.

We were somewhat inspired by the work of Robert Haas that we described in section 1.3.3. He chose to work with the store scan, similar to the all nodes scan in Neo4j, for what we suppose might be similar reasons to ours. Initially his implementation was similar to the common Volcano-inspired approach described in section 1.3.1 with an exchange operator (which he named `Gather`) interfacing the parallel scan to other operators. Later in development the implementation was changed to let the split parallel pipelines execute several steps before gathering, akin to the architecture described in section 1.3.2.

3.2.2 Additional operations

We continued our work with the filter operator, a simple, independent and common operator that is used to filter out nodes matching given criteria on the working set.

After implementing the first query operations, we changed our approach for selecting query operations slightly. In addition to the above we also, took into account what additional operations we would need to implement in order to run queries. We focused on LDBC-queries, as these are the industry standard for benchmarking. We studied the plans that the different queries produced and took these into account, in addition to the points above, when we choose additional operators to implement.

We also found expand very interesting to look into for a couple of reasons. Expand takes all of the nodes in a set and then adds their neighbours to the set. It is an operation unique for graph databases, its nature of greatly increasing the number of results is special as well. To our knowledge, no attempt to parallelize an expand operation in a graph database has been made public before. The typical use case of starting with a single node and then expanding one or multiple times to end up with a large result makes many of the typical parallelization techniques inapplicable, since it is impossible to split the initial data set consisting of a single entry between multiple threads. Thus you have to utilize other techniques in order to split the dataset as it grows.

3.2.3 Implemented pipes

The following pipes have been implemented in our parallel runtime:

- All nodes scan
- Node by label scan
- Index seek
- Index seek unique
- Expand all

- Filter
- Projection
- Sort
- Produce results

3.2.4 All nodes scan

The all nodes scan is a leaf operator, meaning that it is a starting point in the query plan. As such, it communicates with the storage layer of the Neo4j kernel. This operator was a prerequisite for implementing other operators, as we needed something that could feed intermediate results in parallel to the next step. A pre-determined number of nodes, i.e. a morsel, is loaded from the storage medium and passed on to the parent operator, upon request. The standard morsel size was chosen to be a multiple of the number of records per page in the store to optimize page cache usage. Because this operation is a leaf node, it is also responsible for determining the total number of morsels that need to be created. Similar operators are *index seek* (which finds matching nodes from an index) and *node by label scan* (which finds all nodes with a certain label).

3.2.5 Expand all

The expand all operator follows the relationships connected to a collection of nodes and adds whatever it finds to the result set. Because this operation has the potential to greatly affect the number of rows in a morsel by adding new nodes, the result sets from all parallel expand pipes are passed on to a common monitor and redistributed into new morsels of the correct size.

Since the growth cannot be predicted prior to execution, we have to add more morsels as the query is executed. Children (operators closer to the leaves) to the expand operator are unaffected, but new parent operators have to be built for every new morsel added. The way we do this is by splitting the tree at the point of the expand operator, so that we end up with two separate morsel piles. The expand operator that bridges the piles, is broken into two halves; a parent and a child. The child performs the expansion and passes the results to a monitor, while the parent is responsible for relaying the results to the next operator, see Figure 3.2.

Since the Neo4j kernel does not support multiple threads utilizing the same transaction, we had to do some tricks in order to parallelize expand. The concept of a transaction in Neo4j keeps track of changes that occurred in the current transaction and augments the values retrieved from the store to reflect these changes. Since we have limited ourselves to only work with read-only queries we can relax the use of the transaction objects in Neo4j.

We worked around the issue by letting each worker-thread executing the query have their own transaction object. If an expand morsel results in more result rows than the morsel size boundary, the morsel is returned and a new morsel added to the morsel pile. However, another thread cannot later continue expanding this morsel, it must be done by the same thread.

A simple solution to this problem would be to simply keep processing the expanding morsel until it does not have any more results, and store the extra morsels in a buffer for when other threads ask for more morsels. However, this buffer can become quite large - a

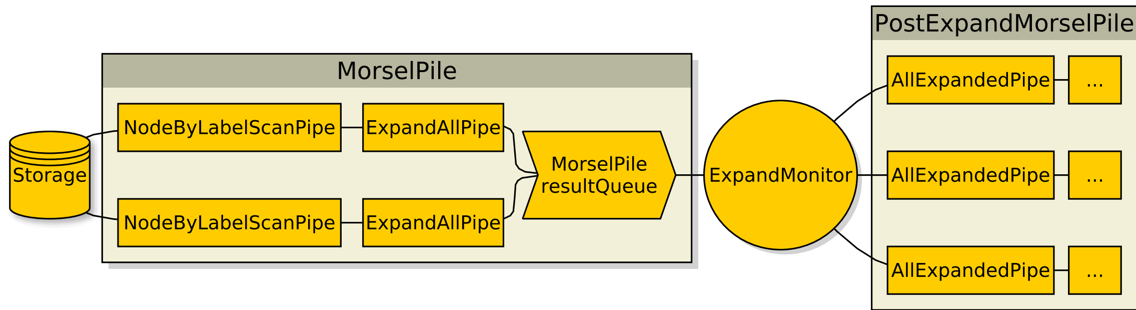

Figure 3.2: Expand operators break the pipeline due to the changed number of rows, and thus have pipes on both sides of the break. *ExpandAllPipe* performs the expansion, while *AllExpandedPipe* relays the results to the new, potentially larger morsel pile.

single morsel can expand into huge numbers. This means that that millions of rows will be stored in relatively long-lived lists. In practice this leads to extreme garbage collection times, in some scenarios we had runs which consisted of 75% garbage collection time.

Instead, we adopted an approach that makes use of the `ThreadLocal` Java class. It allows to store a class field that is local to each thread. When we have expanded a morsel worth of rows we take those and continue down the pipeline with them. We store the half-expanded morsel in the `ThreadLocal` variable, and the thread will continue to use it the next time it is asked for more rows. In this way we avoid storing large amounts of intermediate results in memory.

3.2.6 Filter

Filter is an operation that removes rows that fail to fulfil certain criteria, for example if the user requests nodes that feature a specific value on some attribute. Whenever the parent pipe asks for more results, filter will request a row from its child until it finds one that fulfils the given criteria and discard the rows that do not.

Since filtering affects morsel size, it would perhaps be of interest to redistribute the remaining rows among a new and potentially smaller pile of normalized morsels. We decided against this, the reason being the increased overhead it would entail. This is something that we feel should be explored further in future work.

3.2.7 Sort

We implemented sort using merge sort with a monitor to handle the pairing of morsels as well as the final merge. Just like *expand*, sort consists of two pipes, a parent and a child. The child is tasked with sorting and merging before ultimately submitting its result to the monitor. The parent's job is simply to fetch a sorted morsel from the monitor and pass it forward. In order to preserve the sorted order of rows while they are split into different morsels, each parent pipe is assigned an ID when fetching a morsel from the monitor. This

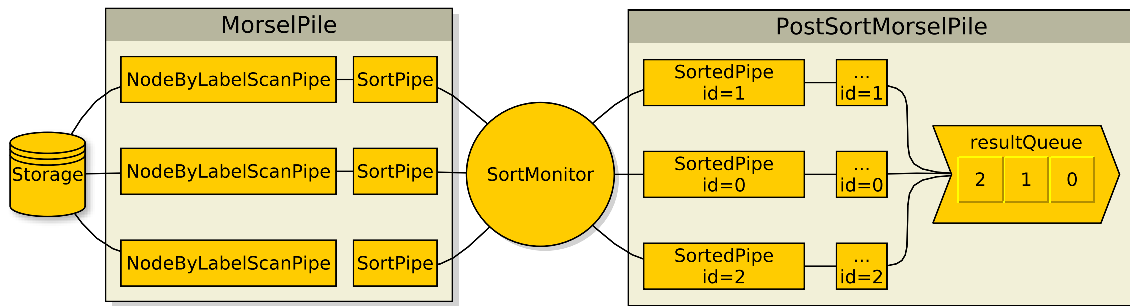

Figure 3.3: The sort monitor will flag the morsel pile as sorted to let it know that the results need to be queued in order.

ID is later used to determine the proper order of the result during the final stages of the query execution, as shown in Figure 3.3.

An executing child sort pipe will start out by sorting its rows before offering the sorted results to the monitor. If the monitor does not already hold a result of the same scale, it will buffer the offered rows and let the morsel shut down. However, if the monitor already has a matching set of rows buffered, it will refuse the pipe's offer and instead supply it with the buffered rows. The pipes task is then to merge these rows with its own, and then repeat the process by offering the merged set to the monitor. When all morsels have finished executing, the monitor merges any leftovers that could not be paired before it starts supplying the child pipes.

Example: Morsels A, B, C and D hold 3 rows each. For the sake of convenience, say that the morsels will finish their sorting in the order they were mentioned. Morsel A finishes first, and puts 3 sorted rows in the monitor's buffer. When morsel B finishes, there is already a sorted set of the same scale in the buffer, so it will combine this with its own, before storing the resulting 6-row set in the monitor. Morsels C and D will behave in the same way at first, but after they have been merged together by D, it is also up to D to combine their 6 rows with the other 6 stored in the buffer, finally resulting 12 sorted rows.

3.2.8 Produce results

Produce results is the last operation of a query, and it works a bit differently than the other operations. The goal of the produce results is, generally speaking, to filter out only the columns the query requested from the intermediate result.

In the parallel runtime it is also responsible for aggregating the results from the executing morsels into a single iterator. It achieves this by utilizing the same morsel piles split by monitors as the other operations detailed here, see Figure 3.4. The morsel pile after the monitor, however, only contains a single morsel, regardless of the amount of morsels before the monitor.

It is also at this stage that the executor threads are created. These are responsible for creating the threads which will fetch and execute morsels from all the morsel piles. The number of threads created to compute the query is equal to the number of processors available to the JVM, in order to assure that we can use all of the available resources if possible. Each of these threads will try to fetch a morsel from the morsel pile, execute it,

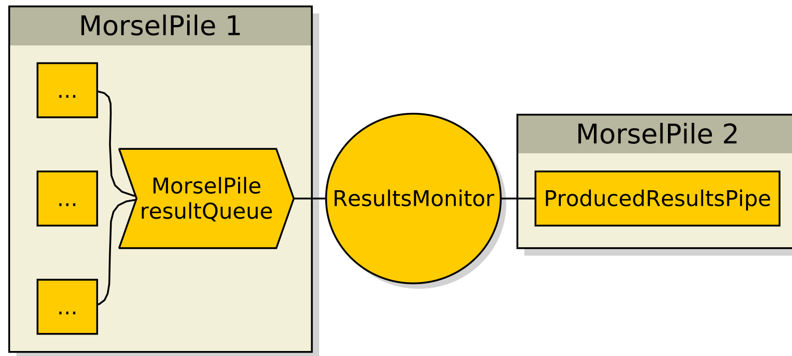

Figure 3.4: Produce results filters the columns and consolidates the query results to a single thread.

and repeat as long as there are pipes available.

3.3 Discussion

When we first drafted the architecture of the parallel runtime we took inspiration both form the morsel driven parallelism outlined in section 1.3.2 and the Interpreted runtime. We wanted our trees of `ParallelPipeBuilders` to be stateless and reusable in the same way that the tree of pipes generated by the Interpreted runtime is. This means that, for a given query plan we do not need to rebuild the `ParallelPipeBuilders` each time we are tasked with the query. All state is contained in the structures built by the `ParallelPipeBuilders`. During our continued work, the key ideas behind the design remained the same, but major implementation changes were necessary to adapt to the requirements of the expand operator.

3.3.1 Expand operators

Expand operators has been, by far, the most challenging aspect of this thesis. On multiple occasions it has been the driving force behind shortcuts, "hacks" and even several redesigns of the parallel runtime. It would be a fair estimate that expand is responsible for at least a third of the time spent on this thesis work.

While our `ThreadLocal` solution to the thread-bound transaction problem works, it is far from ideal. There is some overhead of having to check and use the `ThreadLocal` variables, but perhaps more importantly it adds a lot of complexity to the code. Not only do we have to keep track of which morsels belongs to which thread, we also need to make sure that we finish reserved morsels first and that we do not stop prematurely when we only have reserved morsels left. This system introduces both complexity and additional synchronization that could have been avoided otherwise.

3.3.2 Choosing query operation

Initially we were planning on using another method of deciding what query operator to work with. We were planning on first deciding on a query and dataset to work with and use those as grounds for our decision on what query operator to use. By profiling our query on the dataset we wanted do find out which query operation that was the most time-consuming, and then work on improving that operation.

There were multiple flaws with this approach. Firstly, even though you can see the number of database hits caused by each operation when you profile a query, these are not directly connected to the actual time taken by that operator. It turns out that there is (partly because of the pull-based piped Cypher runtime) no way of getting the actual time consumed by the different operators. Additionally, even if we would have been able to discern the most costly operators for our query, we would have no way of knowing if these would be suited for parallel execution. There could possibly be other operators better suited to parallel execution that weren't a part of the execution plan for our chosen query.

Some of these concerns were raised by us, and some of them by developers at Neo Technology when we discussed our approach with them. These discussions lead to our new approach where we sat down with several developers from Neo Technology to discuss the pros and cons of choosing different operators, as well as different methods of parallelization.

3.3.3 Store access in operators

There are several ways to access the store to fetch data in Neo4j. These all have different performance implications. The only operator where we have tried to optimize store access is the expand operator, where we managed to increase the performance of our implementation by an order of magnitude by not needlessly fetching data from the store multiple times. There might be gains to be had for other operators as well.

3.3.4 Synchronization in the morsel pile

The synchronization of different methods in the morsel pile is not inherently a part of its design, this part of the code organically grew as the design, needs and responsibilities of the morsel pile changed. With the way it currently works, both getting a new pipe to execute and handling results from already executed pipes synchronize on the same monitor. This is because of the fact that these depend on some state that is shared. This leads to heavy synchronization when using a large amount of threads, especially so when running expand operations, as there are extra guards in order to properly distribute and reuse the morsels in that case.

With a redesign of the morsel pile and kernel support for multi-threaded transactions the synchronization could be significantly reduced, and probably split into two sections: one responsible for results, an another responsible for handing out tasks.

3.3.5 Attempt without transaction splitting

In our first iteration of implementing the parallel runtime we tried to use the current kernel-API with minimal modifications. This meant that we worked with a single statement inside a single transaction to access the stores.

In order to do this we had to remove some checks in the Neo4j kernel, such as the one that checks that only the owning thread can access a transaction. Removing these lead to sporadic failures related to thread safety, but this was not the biggest problem of this approach. The biggest drawback was that we did not see any significant performance improvements over the old Interpreted runtime. This might have been due to the fact that transactions by design are constructed to only serve one thread with operations, and as such only have a single entry point to the underlying store.

These problems made us choose the approach where we let each executing thread open its own transaction.

3.3.6 Limit operator

In order to support the more advanced LDBC queries 2 and 8 in full, we originally attempted to implement the limit operator which limits the maximum result size to a given number of rows. This proved to be a greater task than anticipated due to certain aspects of expanding and filtering operators. As such, the support for this operation had to be abandoned due to time constraints.

Because expand and filter pipes have the potential to increase or decrease the intermediate row count, it is practically impossible to predict the number of morsels required to complete a query prior to execution. As a result, we saw no other option than to blindly feed the thread pool with more work until the result quota had been met. This became an issue in cases where the limit was reached before all queued morsels had finished executing because they would continue running in the background needlessly.

A possible solution to the problem would have been to implement a means of terminating leftover morsels and discarding their results upon query completion.

3.3.7 Moving to a common thread pool

In the original implementation of the parallel runtime each morsel pile created their own thread pool, instead of the current design with a thread pool common for the entire runtime. This worked fine when testing and benchmarking locally on our own machines. But when we started running benchmarks on our benchmarking server we quickly ran into problems.

The server executed queries faster than the worker threads managed to stop themselves after having finished their work. This lead to our system racing to approximately 11,000 active threads before crashing due to low memory. To avoid this behaviour we made the results monitor responsible for accessing the thread pool, and made it shared across the entire Neo4j instance. This is also why we had to implement the pipe-serving mechanism for the morsel pile and the special cases of morsel piles.

Chapter 4
Evaluation

In this chapter we describe in detail how we evaluated our work. We describe our measure of performance (4.1), the queries (4.2) and dataset (4.3) used, as well as the hardware, tools and configuration settings used (4.4).

4.1 Measure of performance

Our target use case is of an analytical fashion, where a single user typically runs very large queries in sequence. With this in mind, the most interesting metric to use as our measure of performance is the execution time of a single query. We will compare this time between our parallel runtime and the standard Interpreted runtime for our reference Neo4j version.

4.2 Queries

We used several queries to evaluate the performance of the parallel runtime. Two of these are inspired by the LDBC benchmark and the rest were mainly created for the purpose of evaluating the individual operators of the parallel runtime. The execution time for the queries were compared between the parallel and Interpreted runtime.

4.2.1 LDBC-inspired queries

Since LDBC (Section 2.1.7) provides an open and standardized measure of graph database performance we wanted to include at least one LDBC query in our evaluation. We also looked at the draft for the LDBC business intelligence benchmark, but there were no implementations of these queries in Cypher for us to use. However, the business intelligence benchmark as such might be more relevant to the analytical mindset and approach of this project, so it would be very interesting to explore this in the future.

We searched the LDBC SNB queries for a query that both lead to a relatively small plan and where we had already implemented most of the operations and expressions. We wanted a small plan in order to be able to implement the query operations within the time frame of this project.

We decided to use LDBC social network benchmark query 2 (Query 4.1), as it had the characteristics we were looking for. Since the operators we have implemented allows us to support LDBC query 8 we included that query as well (Query 4.2).

```
CYPHER 3.0
MATCH (:Person {id:{1}})-[:KNOWS]-(friend:Person)
 <-[:POST_HAS_CREATOR|COMMENT_HAS_CREATOR]-(message:Message)
WHERE message.creationDate <= {2}
RETURN friend.id AS personId,
 friend.firstName AS personFirstName,
 friend.lastName AS personLastName,
 message.id AS messageId,
 CASE exists(message.content) WHEN true THEN message.content
 ELSE message.imageFile END AS messageContent,
 message.creationDate AS messageDate
ORDER BY messageDate DESC, messageId ASC
```

Query 4.1: Query inspired by the LDBC social network query 2

```
CYPHER 3.0
MATCH (start:Person {id:{1}})
  <-[:POST_HAS_CREATOR|COMMENT_HAS_CREATOR]-(:Message)
  <-[:REPLY_OF_POST|REPLY_OF_COMMENT]-(comment:Comment)
  -[:COMMENT_HAS_CREATOR]->(person:Person)
RETURN person.id AS personId,
  person.firstName AS personFirstName,
  person.lastName AS personLastName, comment.id AS commentId,
  comment.creationDate AS commentCreationDate,
  comment.content AS commentContent
ORDER BY commentCreationDate DESC, commentId ASC
```

Query 4.2: Query inspired by the LDBC social network query 8

The only difference between our "LDBC-inspired queries" and the actual LDBC queries is that the actual queries have a limit operator limiting the number of result rows. This change should not significantly affect performance, since both the queries we use sort the results, which in turn requires all result rows to be computed.

The parameters for the queries are randomly chosen from a list of applicable parameters each time the query is run. Some of these parameters produce no or very few results, which leads the query execution time to be very fast. It is thus of importance to run the query a large number of times so that we use most, if not all, of the parameters.

4.2.2 Small queries

In addition to the LDBC query we also created five small queries (SQ) that each are centred around a specific query operator.

We wrote these queries as a means of gauging performance and multi-threaded scaling of individual operators, in contrast to the LDBC-inspired queries that run several operators in unison, and thus run the risk of being bottlenecked by operators that do not scale well.

We also created a query that consisted of most supported operators, but still remained simple and did not rely on actions on properties as much as the LDBC-inspired queries does.

Small Query 1 — Scan

SQ1 (Query 4.3) is a simple query, consisting only of a scan of all nodes in the database. This query is interesting as an example of how the parallel runtime performs with tasks that are not computationally heavy, but rather I/O bound. Considering that other queries we have chosen to benchmark also rely on `AllNodesScan` as a starting point, it also serves as a reference for whenever other operations scales better or worse than this operation.

The pipes used in SQ1 are `AllNodesScan` and `ProduceResults`.

```
CYPHER 3.0
MATCH (n)
RETURN n
```

Query 4.3: SQ1 — Scan

Small Query 2 — Filter

SQ2 (Query 4.4) is designed to test the scaling of the filter operator. Filter should in theory be an operator that scales well in a multi-core environment due to the independent nature of the operator. Unlike other operations which require communication and collaboration between morsels, filtering can be applied to morsels in complete isolation and should therefore suffer minimally from overhead introduced by parallelization.

Pipes used in SQ2 are `AllNodesScan`, `Filter` and `ProduceResults`.

```
CYPHER 3.0
MATCH (n)
WHERE (n.id % 7) / (n.id % 5 + 1) > n.id % 3
RETURN n
```

Query 4.4: SQ2 — Filter

Small Query 3 — Expand

SQ3 (Query 4.5) uses a single node as its starting point followed by four consecutive expands. Expand is another operation which also has the potential to scale with core count, though it comes with some added overhead due to morsel resizing. The results of the small query 3 benchmarks are especially interesting to us as expand is an operation exclusive to graph databases.

Pipes used in SQ3 are `IndexSeekUnique`, `ExpandAll`, `Filter` and `ProduceResults`.

```
CYPHER 3.0
MATCH (:LBL_ALL {id: 777})--()--()--()--(n)
RETURN n
```

Query 4.5: SQ3 — Expand

Small Query 4 — Sort

SQ4 (Query 4.6) makes heavy use of the `Sort` pipe by sorting the entire dataset by the two properties idMod7 (descending) and id (ascending). In contrast to filtering and expanding, sorting is a task that *does* require pipe intercommunication. It is therefore interesting to see how much can be gained in this case and how much the added overhead limits scaling.

Because sorting is such a taxing operation, we only sort the first 10 million nodes in the dataset, which are available under the label `LBL_10M`. This brings execution time down to more reasonable levels on par with other small queries.

Pipes used in SQ4 are `AllNodesScan`, `Projection`, `Sort` and `ProduceResults`.

```
CYPHER 3.0
MATCH (n:LBL_10M)
RETURN n
ORDER BY n.idMod7 DESC, n.id ASC
```

Query 4.6: SQ4 — Sort

Small Query 5 — Mixed workload

SQ5 (Query 4.7) features a larger selection of pipes working in tandem to showcase the performance of the parallel pipeline in a slightly more complex scenario. In this scenario all the different kinds of morsel piles and monitors are used in a single query and can be viewed as a more basic alternative to the LDBC queries.

Pipes used in SQ5 are `AllNodesScan`, `Filter`, `ExpandAll`, `Projection`, `Sort` and `ProduceResults`.

```
CYPHER 3.0
MATCH (m)--(n)
WHERE (m.id % 1113) = 7
RETURN n
ORDER BY m.idMod7 DESC
```

Query 4.7: SQ5 — Mixed workload

4.3 Dataset

In this section we describe the datasets used for the evaluation. We used different datasets for the LDBC-queries and the queries we designed ourselves.

4.3.1 LDBC-SNB dataset

LDBC-SNB (Section 2.1.7) datasets can be generated in different sizes, called scale factors. For example, the scale factor SF001 implies that the size of the generated dataset in comma separated values (CSV) format has a size of 1GB. For this evaluation we use the scale factor SF300, i.e. the CSV data was 300GB. This translates into roughly 870 000 000 nodes and 5 025 000 000 relationships.

The schema for the LDBC SNB dataset in Neo4j can be seen in Figure 4.1[33].

4.3.2 Small queries dataset

Our SQ dataset (Figure 4.2) was designed to meet the technical requirements of the small queries without unnecessary complexity. It therefore features the bare minimum number of properties and labels required.

The set consists of 100 million nodes, all of them under the label 'LBL_ALL'. Each node has two properties, 'id' and 'idMod7'. The id property is an integer value unique to each node and ranges from 0 to 99,999,999 whereas idMod7 is equal to the id value modulus 7. The id property has been indexed under LBL_ALL to enable the use of unique index seeking in our queries. The first 10 million nodes in the set are also available under the label 'LBL_10M'. The set features a total number of 5 billion relationships, all of the type 'REL_ONE'.

Each node in the dataset features 50 outgoing relationships leading to the next 50 nodes in (id) sequence, with a wrap around from the end of the set to the beginning. To further clarify, a node with an id property value of n has outgoing relationships to the nodes with id values $(n + k) \bmod 10000000, 1 \leq k \leq 50, k \in \mathbb{N}$. See Figure 4.3 for a graphical representation of the relationships.

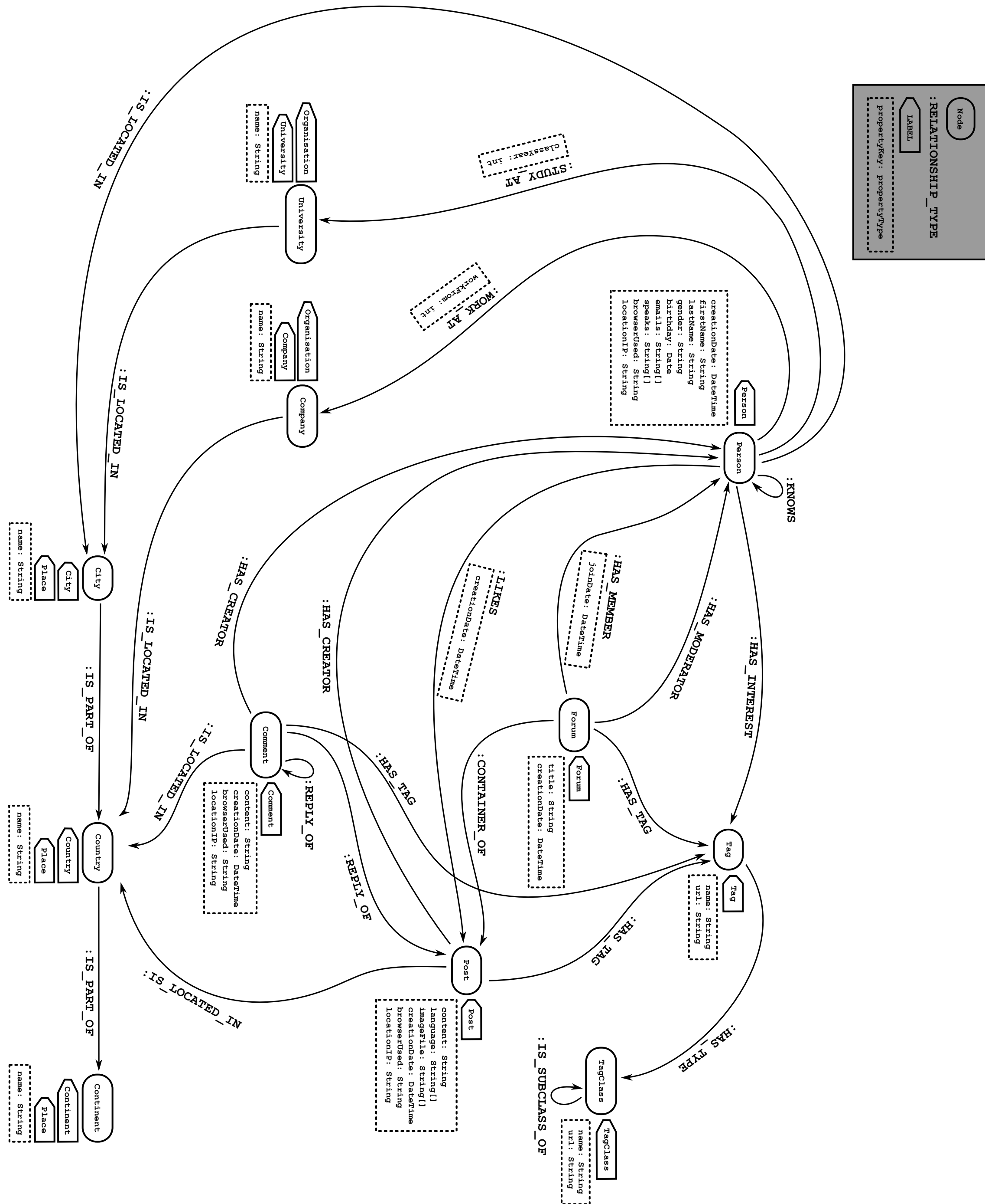

Figure 4.1: The schema for the LDBC SNB dataset in Neo4j

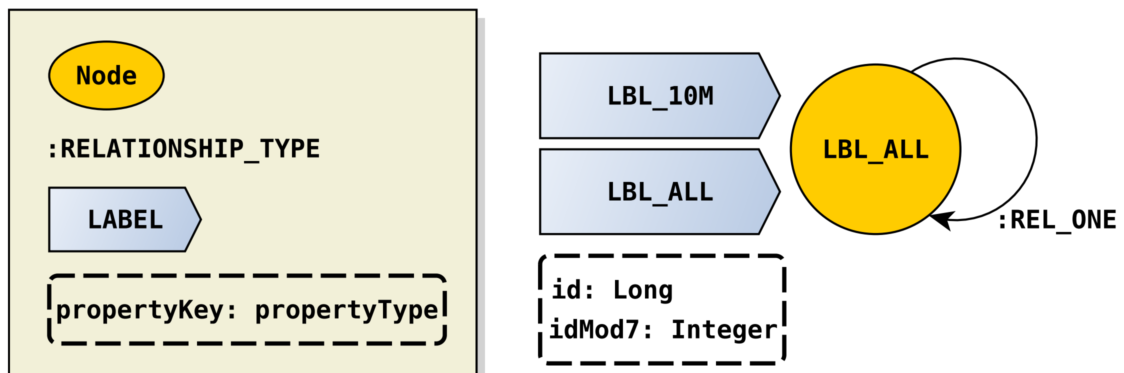

Figure 4.2: The schema for our small query dataset.

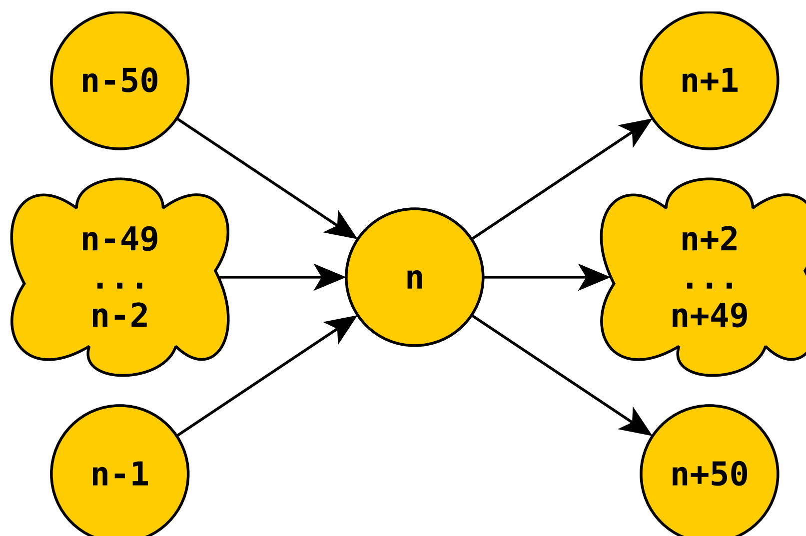

Figure 4.3: Visualisation of entity relationships in the small query dataset. Each node has outgoing relationships to leading to the next 50 nodes the in series. The relationships loop around at the end of the series, meaning that every node, including the first, also has incoming relationships from the previous 50 nodes.

4.4 Hardware and tools

This section contains a detailed description of the setup used for the evaluation. We specify the hardware, tools, versions and parameters used.

4.4.1 Hardware

The benchmarks were run on a dedicated server with the specification outlined in Table 4.1. We had exclusive access to the server, so no other jobs were running in the background.

Part	#	Specification	Note
CPU	2	Intel Xeon E5-2699v4, 22-Core, 2.2/3.6GHz, 55MB cache	In total 44 cores, 88 threads.
RAM	16	Samsung 32GB DDR4 ECC REG 2400MHz x4 DR	Runs at 2133MHz, 512GB total.
Disk controller	3	Supermicro AOC-S3008L-L8e	12Gb/s Eight-Port SAS Internal Host Bus Adapter
Disks	24	Samsung SM863 1920GB SATA SSD 520/485MB/s	Raid 0
Motherboard	1	Supermicro X10DRI-T-B	Intel C612 Express chipset
OS		Ubuntu 16.04.1 LTS	Kernel 4.4.0-38
Java version		Oracle Java 1.8.0_101	build 1.8.0_101-b13

Table 4.1: The machine used for the evaluation

4.4.2 Reference Neo4j version

Choosing a reference version is a trade-off between recent software and stable software. The software being more thoroughly tested and less changes being made are both arguments to choose an older version, while a newer version might have additional features. There is also a choice to be made whether a specific commit, minor version or major version should be used. We decided to keep up to date with the current newest minor version, keeping up to date with the latest commit would require too much overhead work, and would make our evaluation harder to reproduce.

We thus used Neo4j 3.0.6 for our evaluation, which was the latest Neo4j version at the time of testing. We used the default configuration settings with two exceptions; we configured the page cache size to be 200GB and the heap size to be 64GB.

4.4.3 numactl

We used the tool numactl in order to investigate the scaling of the parallel runtime. It allows the execution of a program with limited access to the hardware of the machine. By utilizing this we ran our benchmarks with several different core configurations.

We used three different series of core configurations for our small queries benchmarks:

1. Cores on one socket, no Hyper-Threading(HT). Starting with a single core, then 2, 4, then incrementing by 4 until the max of 22 is reached.
2. Cores on two sockets, no HT. Starting with two cores, then 4, 8, then incrementing by 8 until the max of 44 is reached.
3. Cores on one socket, HT. The same as configuration 1, but including the hyper-threaded core for each included core.

For reasons explained in section 5.1 we only used 1, 2, 3 and 4 cores, without HT, from the same socket for the LDBC benchmarks.

When running benchmarks on one socket only we disable access to the RAM attached to the other socket. When running benchmarks on both sockets we enable all RAM, but we keep the RAM configurations for the JVM heap and the page cache the same.

4.4.4 JMH

We used JMH (see section 2.1.8) to write, run and analyse our benchmarks. The setup for the benchmark includes starting the dataset and loading all the possible parameters into memory. The time for the setup is not counted towards the benchmark results. When running the LDBC benchmarks all of the possible parameters are stored in an array, and when executing a query a random number generator chooses a parameter from the array and passes it to the database along with the query.

Because of the complex and non-deterministic nature of JVMs we run multiple ”forks”, i.e. JVM invocations, for each benchmark. This is to minimize the effect of run-to-run variance. JMH automatically aggregates the result of the different forks for each individual benchmark.

For the LDBC benchmark we used 5 forks, 15 warmup iterations, 10 iterations and 30 seconds for each iteration and warmup iteration. The high number of warmup iterations is to make sure that we use most, if not all, of the possible parameters at least once during warmup. These settings were chosen as they provided consistent results with low variance.

We used slightly different parameters for the SQ benchmarks, as their characteristics differs. Where the LDBC benchmarks are comparatively very fast and contains a random element, the SQ benchmarks are slower and more stable. Since the SQ queries always access the same nodes, we do not need to run as many warmup iterations to make sure that we load all necessary data into the page cache. And since the queries does not contain any random element, and thus differ less in execution time, we do not need as many measurement iterations. The iteration time, however, needs to be longer since the queries take much longer to execute.

With the above in mind, we used 5 forks, 3 warmup iterations, 5 iterations and 100 seconds for each iteration for the SQ benchmarks. This was the highest we could set these

parameters to in order to have our benchmark complete in reasonable time. I took roughly three weeks for the benchmark to run with these parameters.

Chapter 5

Evaluation results

In this chapter we present and discuss the results of our evaluation. As the LDBC benchmarks only showed low levels of CPU utilization, we only briefly touch upon those in section 5.1.

For the full results for the SQ benchmarks, including error, please see appendix A. In general, the execution time for the two-socket runs were higher than for the single-socket runs with the same core count. In some cases the two-socket benchmark never reached the performance of the single-socket or Hyper-Threading runs, despite having twice the number of cores. We attribute this to the fact that Java is completely unaware of the consequences brought on by dual-socket configurations and thus does not distribute work in a way that minimizes the need for inter-socket communication.

Unfortunately the JVM is completely Non-uniform memory access(NUMA)-unaware. This means, among other things. that the `runtime.availableProcessors()` call does not reflect the actual number of processors available to the JVM. Even if the number of processors is restricted by e.g. numactl, the JVM still reports the total number of processors for the machine it runs on.

In this case, it means that the parallel runtime always spawns 88 threads per query, which may potentially degrade performance slightly.

5.1 LDBC results

The results and for the LDBC query 2-inspired query can be seen in Figure 5.1 and the results for the query 8-inspired query in 5.2. These figures clearly show lack of scaling when we increase the number of cores used. There is even a hint of negative scaling as the amount of cores used increases.

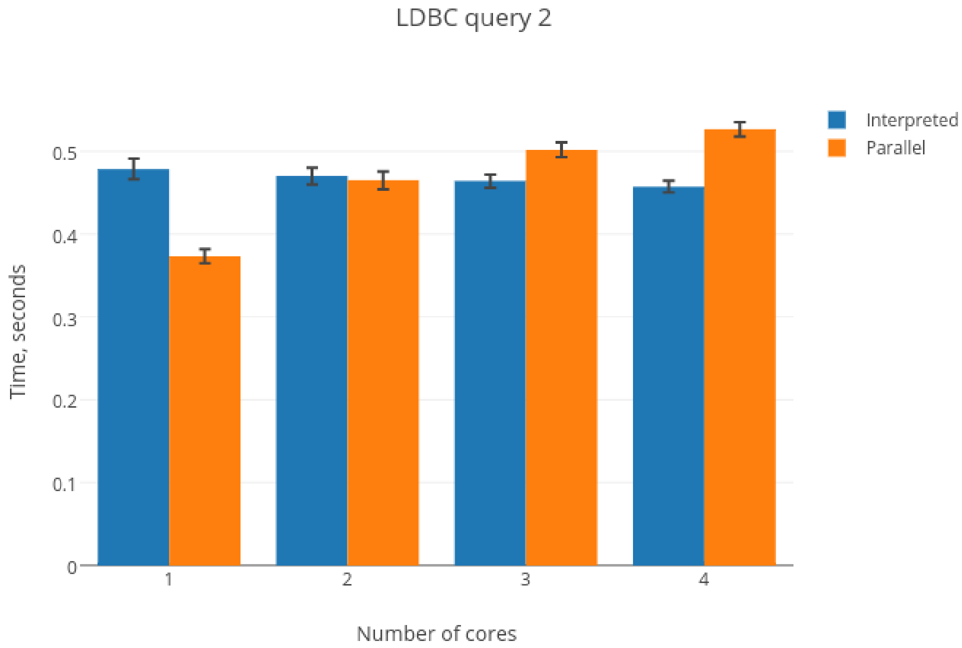

Figure 5.1: Execution time for the LDBC query 2-inspired query, with error bars

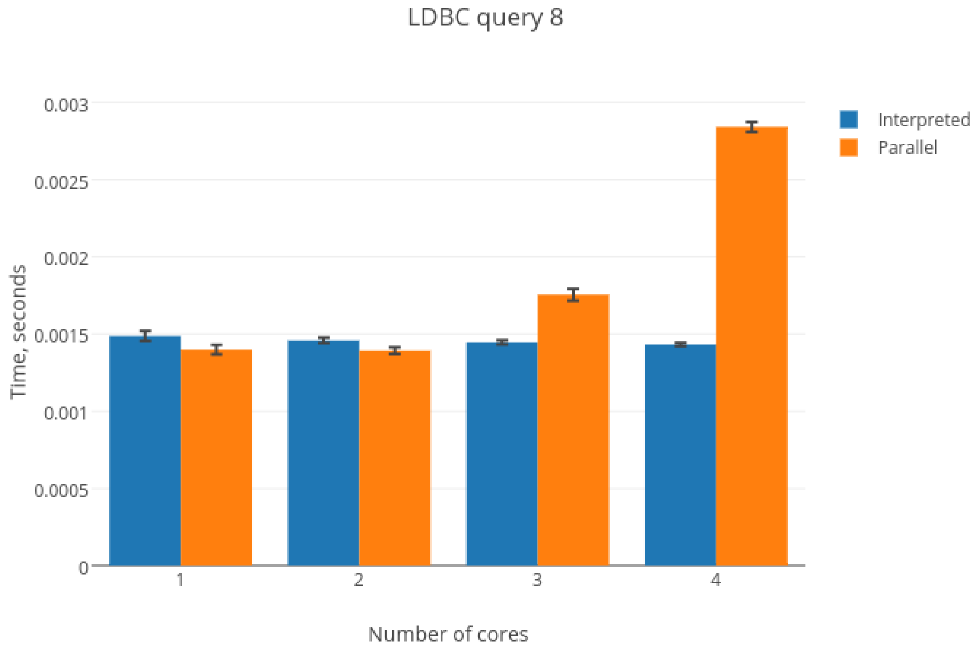

Figure 5.2: Execution time for the LDBC query 8-inspired query, with error bars

In order to understand these results, some knowledge on the characteristics of the benchmarked queries is needed. These queries result in a relatively small number of rows, even in the intermediate steps of the execution, only enough for one or a few morsels. This by itself limits the parallelism possible with our approach.

By looking at the resource utilization when running the queries it is very clear that CPU is not the bottleneck of these queries, even when running with the Interpreted runtime. The usage is around 25% of a single core. This means that the benefits from parallelization is negligible, but we still see the disadvantages of the additional synchronization and overhead. We are not entirely sure of what causes the bottleneck, but we suspect it to be something related to properties, as these queries heavily rely on properties, which in turn leads to many disk accesses to the property store.

5.2 Small Query 1 (scan) results

The execution time for SQ1 can be seen in figure 5.3. The first thing that stands out from this graph is that the single-core results for the parallel runtime is much slower than for the Interpreted runtime. This is the only query that behaves this way, for all other queries the parallel runtime is faster than the interpreted even when the core count is 1. This is an indication that we are doing something unnecessarily costly in our nodescan-pipe, and this is something that we would have investigated had time allowed it.

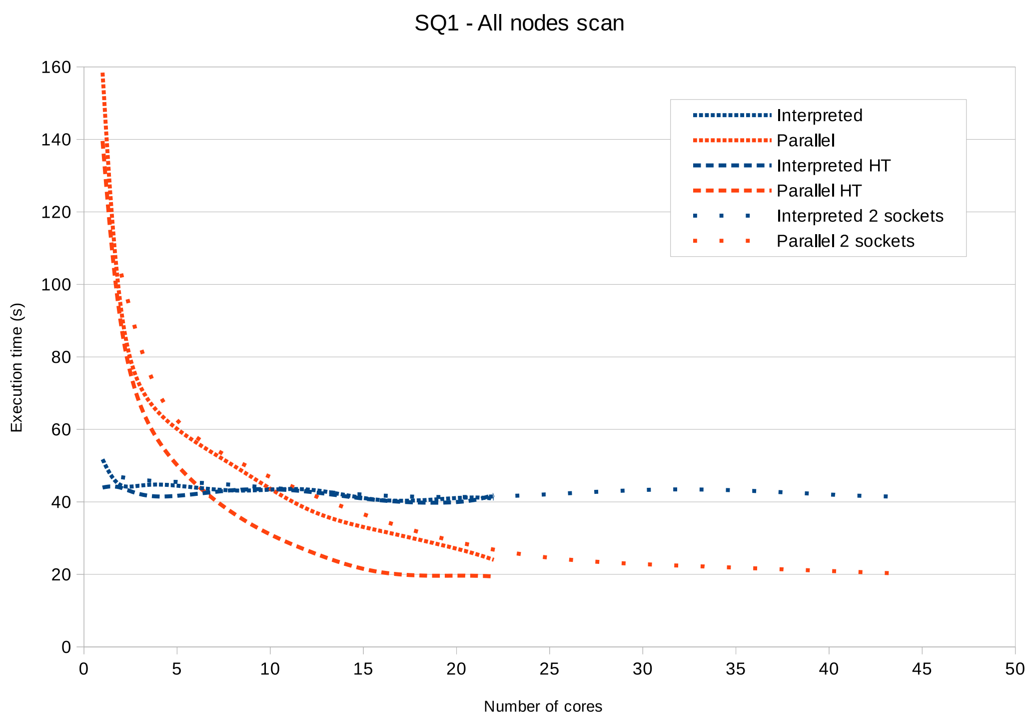

Figure 5.3: Execution time for SQ1, linear scale.

With a core count of around 8 the parallel runtime starts to outperform the Interpreted runtime, and the performance continues to improve as the number of cores increases. The

best speed up for SQ1 is 2.16 times faster at 22 cores with hyper-threading. Thus the single CPU with Hyper-Threading outperforms the two cores without hyper-threading at the same number of hardware threads. The two CPUs does however outperform the single CPU without hyper-threading.

The results were plotted on a logarithmic scale in figure 5.4 in order to show the scaling of the runtime. We see that the scaling does not decrease as we increase the core count, with the exception of the single processor with HT for very high core counts.

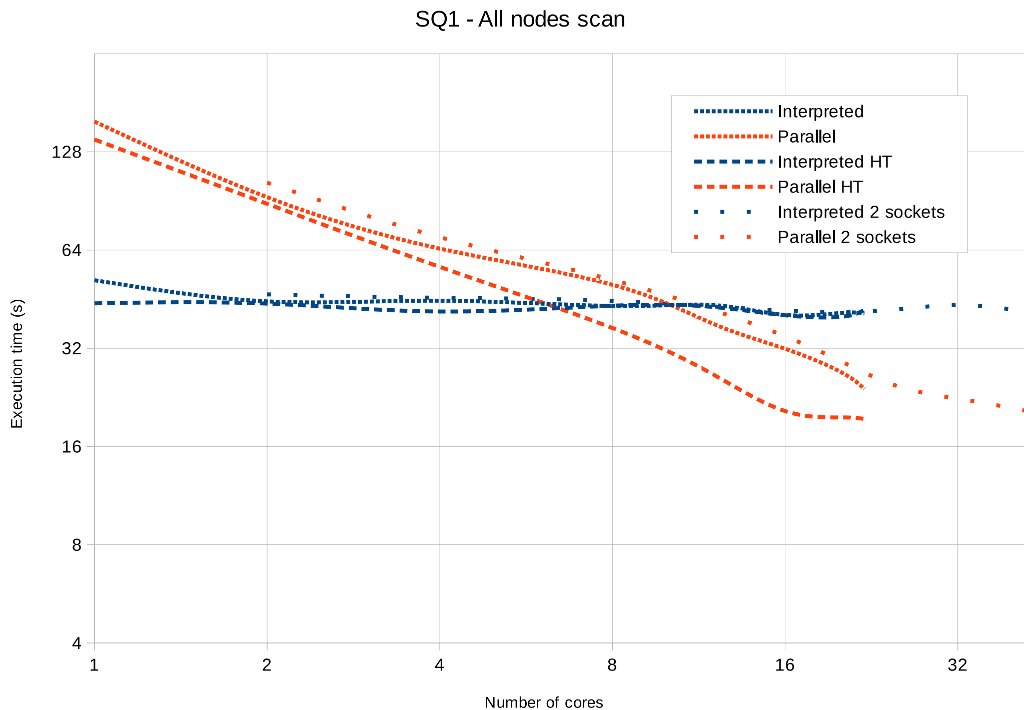

Figure 5.4: Execution time for SQ1, logarithmic scale to show scaling.

5.3 Small Query 2 (filter) results

The execution times for SQ2 can be seen in 5.5. For SQ2 the parallel runtime is generally faster than the interpreted even for the single-core scenario. We attribute this to differences between Java and Scala as well as the reduced overhead the parallel runtime has due to its limited functionality.

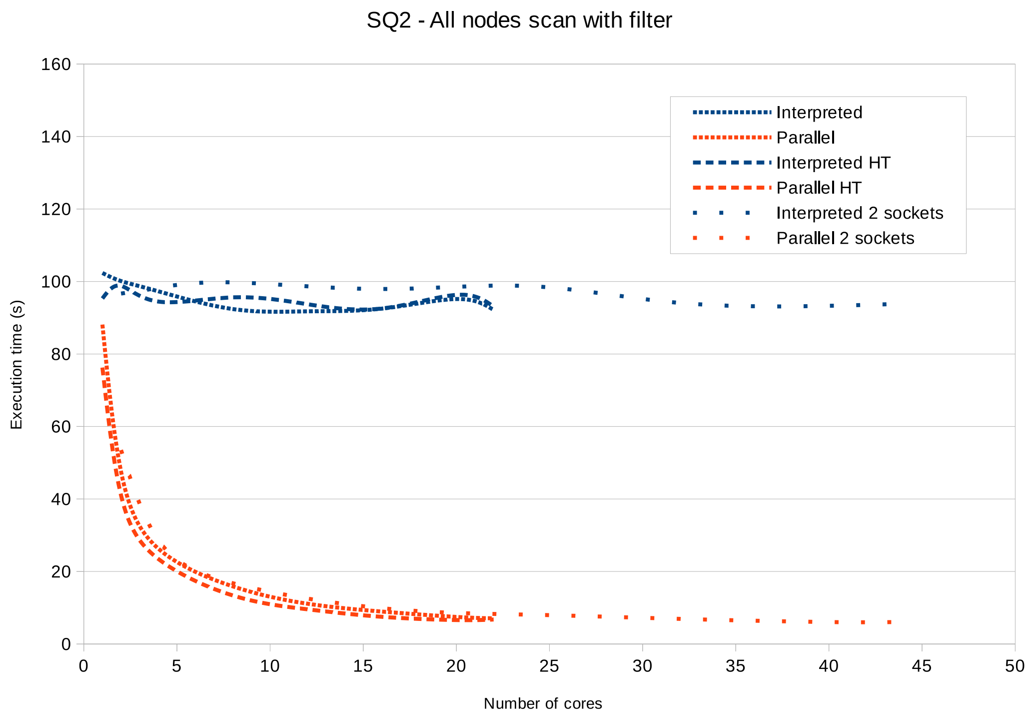

Figure 5.5: Execution time for SQ2, linear scale.

At its best the parallel runtime is 15.4 times faster than the interpreted. This is for the two-core setup with 40 and 44 cores. That the two-core setup outperforms the hyper-threading setup for SQ2 is probably related to the very independent workload SQ2 has, with a scan followed by a filter. This would mean that the effects of the slow inter-socket communication is limited, since such communication is not necessary.

Looking at the logarithmic graph in figure 5.6 we see a very steep and straight line for the parallel runtime, indicating very good scaling even with higher core counts.

5.4 Small Query 3 (expand) results

Looking at the results for SQ3 in figure 5.7 we see something very interesting. Up to 8 cores the execution time decreases, but further increasing the number of cores actually increases the execution time. For very high number of cores the execution time decreases again, but not down to the level of 8 cores.

This behaviour is probably due to the heavy synchronization of the expand pipes, which is required due to the lacking support of multi-threaded transactions (see section 3.2.5 and 3.3.1). At its best the parallel runtime is 5.2 times faster compared to the interpreted, for 8 cores across two sockets.

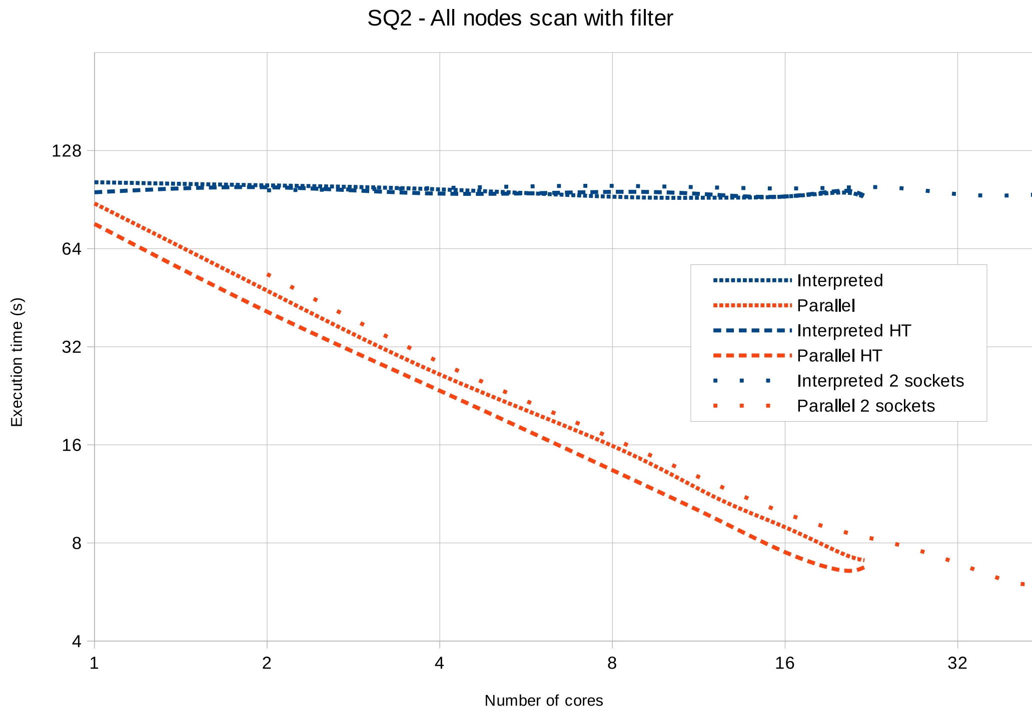

Figure 5.6: Execution time for SQ2, logarithmic scale to show scaling.

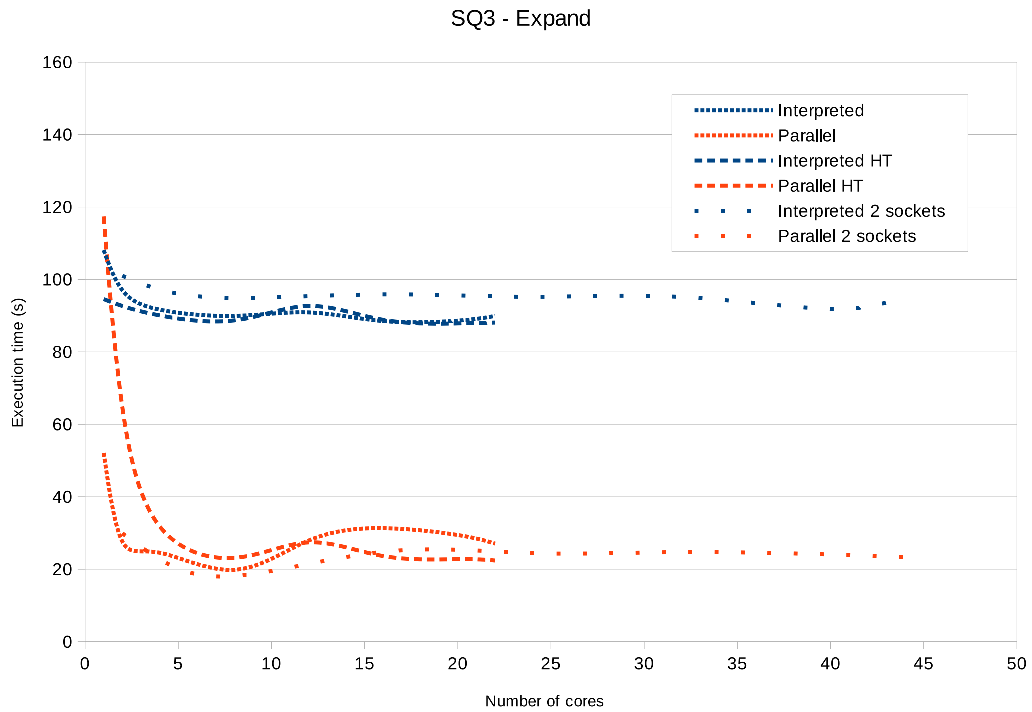

Figure 5.7: Execution time for SQ3, linear scale.

As for the scaling shown in figure 5.8 we see that we start out with decent scaling for a small number of cores, but also that it drops off quickly. Towards the end of the graph we even experience some negative scaling. One interesting thing to note is that SQ3 is the only query that generally performs worse with hyper-threading than without. This supports our theory that synchronization is a major problem for SQ3, as Hyper-Threading adds a new thread contending for the monitors without adding the compute power of a dedicated core.

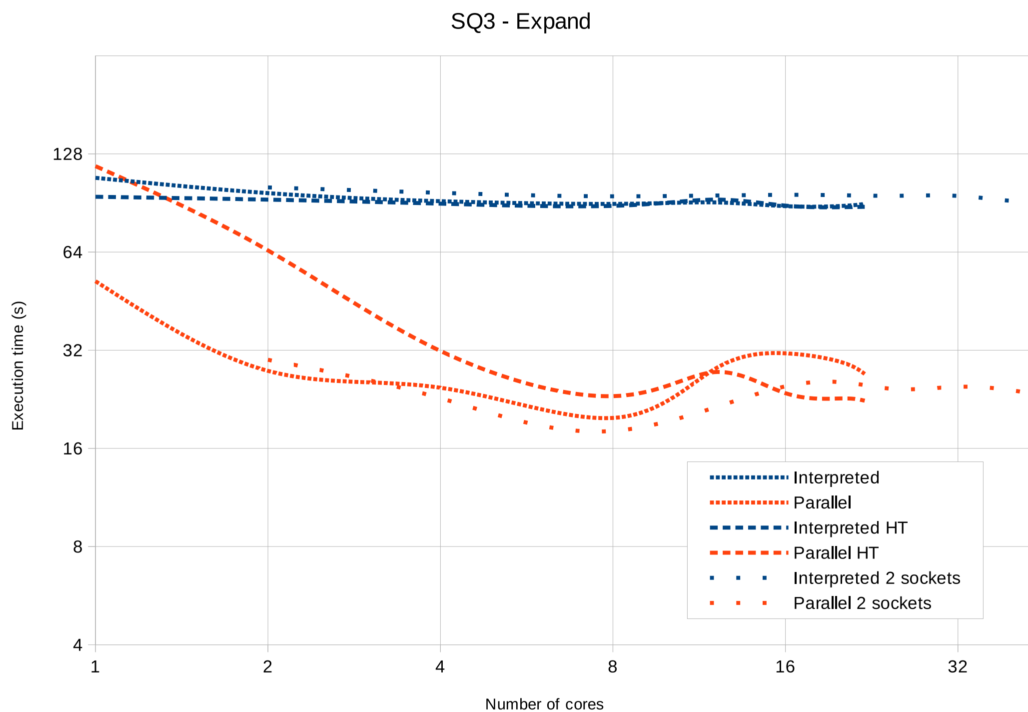

Figure 5.8: Execution time for SQ3, logarithmic scale to show scaling.

5.5 Small Query 4 (sort) results

The results for the SQ4 benchmark can be seen in figure 5.9. The first thing to note is that Interpreted runtime also scales with core count. This is because sorting is done with a multi-threaded library function in the Interpreted runtime. Another thing to note is that the results for the parallel runtime stabilizes at around 8 seconds. We believe that the node by label scan is partially responsible for this. Even though the node by label scan is supported by our runtime it is not parallelized. It is only implemented as a multiplexer, the actual store access is still single-threaded. This means that the execution time for the 10-million node by label scan is constant regardless of the core count, which would explain the diminishing returns of increasing the core count.

For around 8 - 12 cores the parallel runtime is 4 times faster than the interpreted. By looking at the scaling graph in figure 5.10 we see that even if the Interpreted runtime scales,

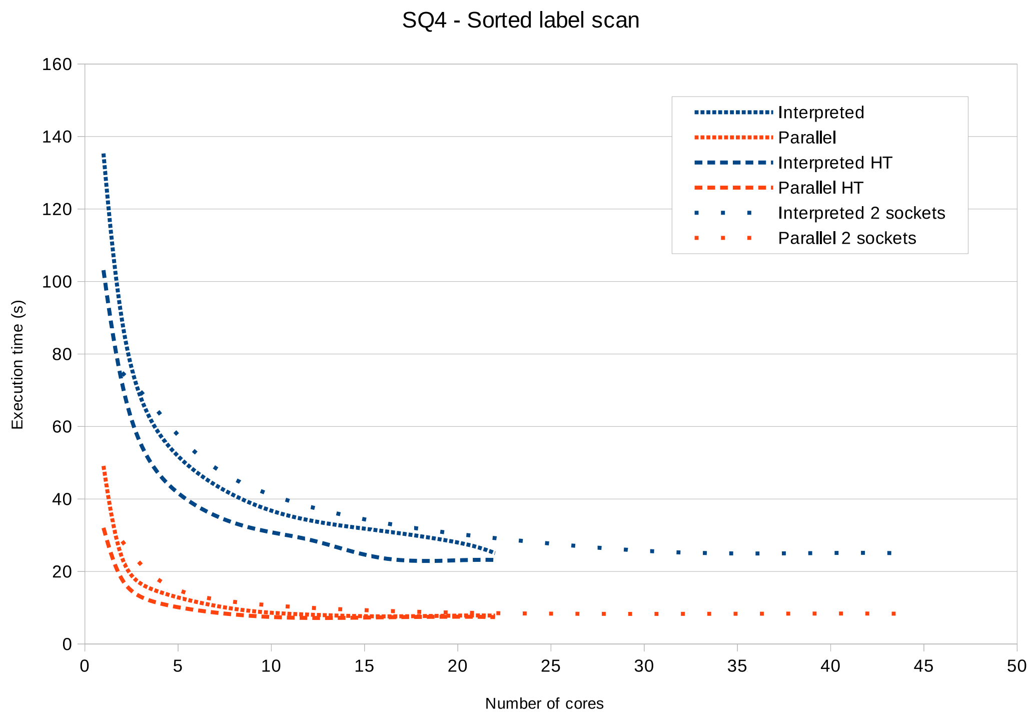

Figure 5.9: Execution time for SQ4, linear scale.

the parallel runtime shows slightly better scaling with a steeper slope. We also clearly see how the scaling drops off when the execution time approaches 8 seconds.

5.6 Small Query 5 (mixed) results

While the results for SQ5 might not be as interesting for evaluating single operators it shows how the parallel runtime behaves in a slightly more complex scenario. The result can be seen in figure 5.11. We see that the parallel runtime is significantly faster, even if the execution time for the interpreted also decreases somewhat due to the sorting.

Looking at the scaling in figure 5.12 we see some very good scaling for 1 to 8 cores. After that we see some negative scaling, which is most probably the expand operator showing the same characteristics as we saw in the results for SQ3. At its best the parallel runtime is 6.4 times faster at 16 cores. That we keep having good scaling even with at larger query tree shows that our way of using a common fixed-sized thread pool for the entire query is a scalable solution.

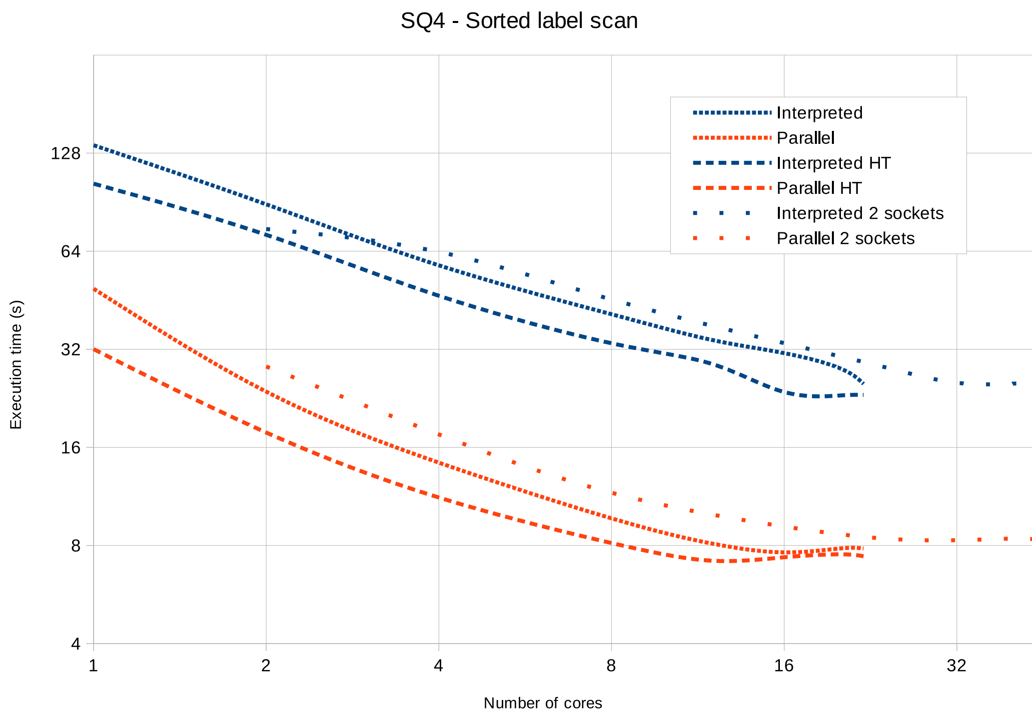

Figure 5.10: Execution time for SQ4, logarithmic scale to show scaling.

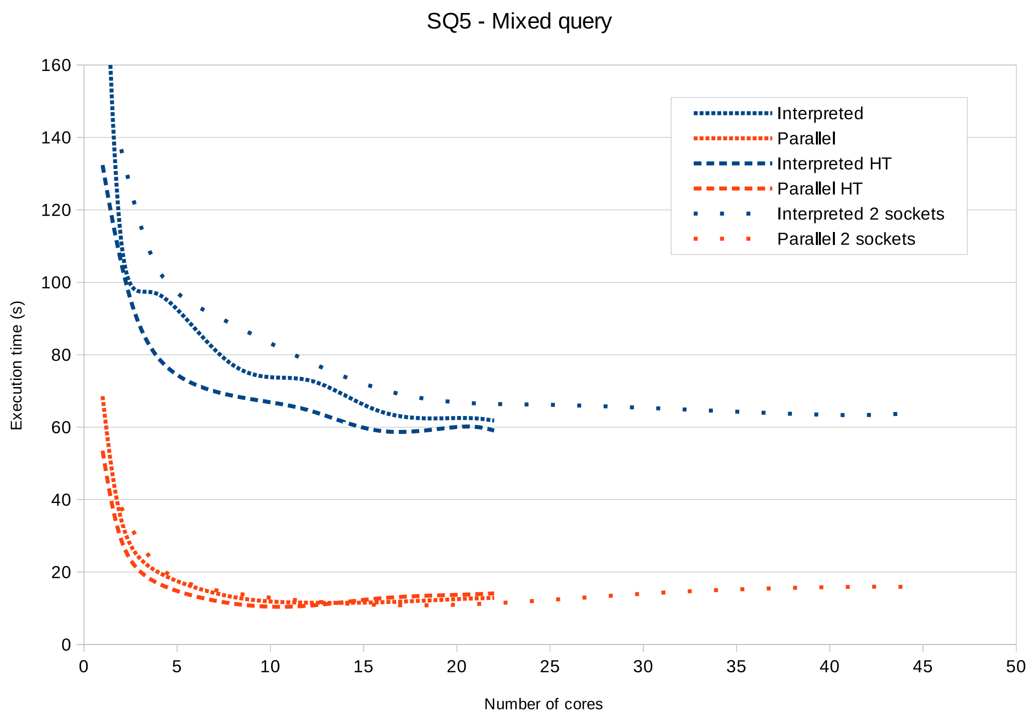

Figure 5.11: Execution time for SQ5, linear scale.

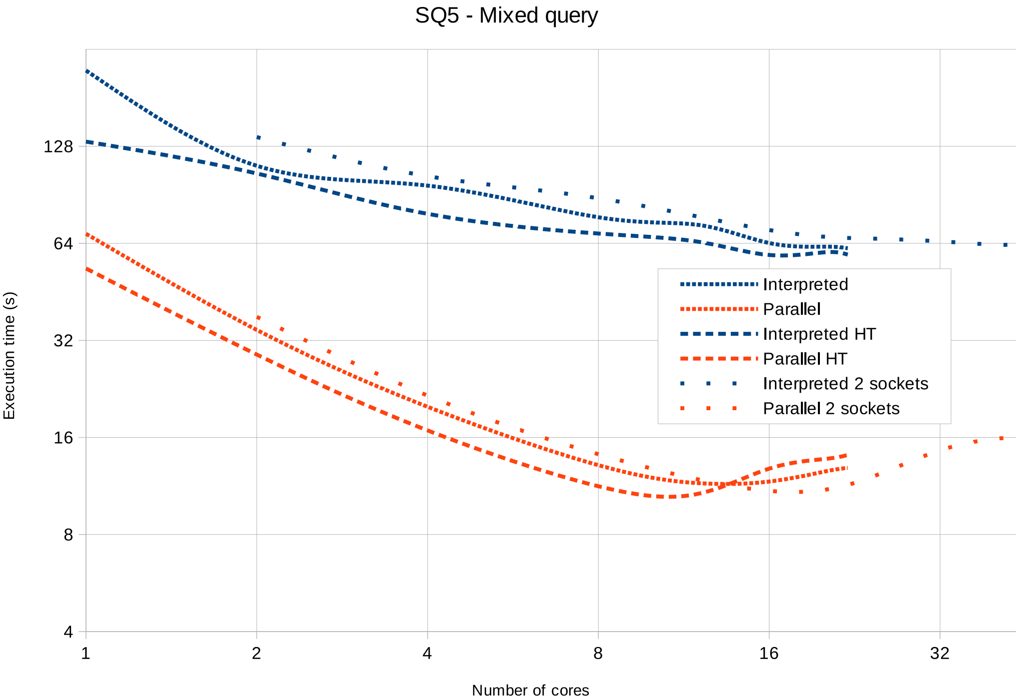

Figure 5.12: Execution time for SQ5, logarithmic scale to show scaling.

Chapter 6
Conclusions

6.1 Query performance

In general we are satisfied with the query performance of the parallel runtime, with some exceptions, as it showed significant scaling capabilites.

6.1.1 LDBC query performance

The LDBC query performance does not in any way speak in favour of the parallel runtime. However, after discussing the results with developers at Neo Technology, the queries and the low resource utilization leads us to believe that the poor performance is due to some other bottleneck in the neo4j kernel. After discussing this with neo4j engineers the property handling in Neo4j was brought forward as a possibility. We conclude that this is a problem in Neo4j, and not necessarily a characteristic of the parallel runtime.

6.1.2 SQ query performance

In general the parallel runtime showed good scaling and performance for the SQ queries, with a few caveats. For SQ1 the baseline performance was poor, so even though the scaling was good we only ended up with a 2x speed-up over the Interpreted runtime.

The SQ3 results are also not without concern. Even though we show good scaling for lower core-counts it quickly drops off and even turns into negative scaling as the core-count increases. We believe that this result had been different if Neo4j had supported multi-threaded transactions, as this would have given us the opportunity to remove almost all of the extra overhead and synchronization associated with the expand operator.

With the achieved performance gains taken into account, we conclude that the design used for the parallel runtime is viable for multi-threaded execution of graph database queries.

6.2 Designing for expand operators

Implementing the expand operator correctly has by far been the most difficult part of this work, especially taking into account that we have had to work around the lack of support for multithreaded execution in the Neo4j kernel.

We did not have the expand operator in mind when we made the original design for the parallel runtime. When we later added support for the expand operator, we had to introduce some shortcuts and references in the code that we think would be unnecessary if the requirements of the expand operator was taken into account in an earlier stage in the design process of the parallel runtime.

This mainly revolves around the increasing amount of intermediate results as the query is being executed. More specifically, adding additional morsels requires the creator to retain knowledge of the rest of the query following the expand operator, in order to create the correct morsels. Originally, we discarded any such information after the query tree was built.

We conclude that expand-like operations should be given a high priority when designing a query runtime for a graph database. The expand operations are very different from the operators that can be found in conventional relational databases. Being able to efficiently start from a few or a single entity, and then effectively expanding a multitude of relations to that entity is one of the key benefits of utilizing a graph database, as compared to multiple-join queries.

6.3 Limitations

There are several limitations of the parallel runtime, some of which are intentional to limit the scope of this thesis.

6.3.1 Limited operator support

Our parallel runtime only supports a select few query operations. The real-world usability of the parallel runtime is thus almost non-existent. It is generally only able to run queries specifically designed for it. It is more fitting to see our work as a proof-of-concept rather than as a alternate cypher runtime for Neo4j.

6.3.2 Multiple transactions per query

Because of the lack of support for multi-threaded transactions in the Neo4j kernel we have had to implement our parallel runtime by creating a new transaction for each thread that is working with the query. Not only does this create additional overhead that might affect performance, it also makes us loose all the other benefits that comes with the transaction context such as transaction isolation.

Therefore we can not guarantee the correctness of a query executed with the parallel runtime, if there are queries that can introduce changes to the database running in parallel. This is one of the biggest limitations of our work.

Furthermore, the lack of this support greatly added to the complexity of implementing the expand operator. The way the Neo4j kernel works, if one thread fetches a result iterator from the kernel it cannot later be utilized by another thread. Because of this we had to utilize synchronization and thread-local buffers in order to implement the expand operator. Had the Neo4j kernel supported these kind of operations, the implementation of expand could have been a lot simpler and more efficient.

6.3.3 Synchronization

The synchronization in the parallel runtime is not something that has been designed, but it has instead grown organically as needed. We believe that if one were to redesign e.g. the morsel pile from scratch it would probably be possible to utilize several smaller monitors for the different functionalities, instead of a single monitor for the entire class, as is the case now. There might also be cases where we hold our exclusive access for longer that is strictly necessary, and cases where volatile class members could be used instead of taking exclusive access.

6.4 Conclusion

Looking back at the questions in section 1.1 we draw the following conclusions:

1. Parallel execution of Cypher queries can be introduced to Neo4j by creating a new cypher runtime that splits the workload across multiple threads. However, the lack of support for multi-threaded transactions in Neo4j currently limits the viability of this approach. We have worked around this problem, but we had to sacrifice both performance and transaction isolation in the process.

2. The operations that are fully independent are best suited for parallel execution. One such example is the filter operation, where we have achieved great speed ups. Other operations, such as expand, does also show some potential for speed up, but not on the level of the filter operation.

3. This work has shown that there are performance improvements to be had when running a single query in parallel, despite not having support for multi-threaded transactions. The parallel runtime should be seen as a proof of concept, and a future implementation of parallel runtime should be reimplemented from scratch, drawing knowledge from this work and its conclusions.

6.5 Future work

There are several topics that should be improved upon or investigated further, the most prominent are listed here.

6.5.1 Multi-threaded transaction support in Neo4j

Support for multiple threads accessing the same transaction in the Neo4j kernel would lead to a lot of improvements. First and foremost a parallel runtime could be utilized in a normal database context. This means that we would no longer be restricted to only executing in a read-only environment, and would also enjoy the ACID-ness that comes with proper transaction support.

Another main benefit would be the possible simplification and increased efficiency of the parallel runtime itself. Without having to worry about the thread ownership, the expand operator could be greatly simplified and also have a higher degree of parallelism in certain scenarios.

6.5.2 Expand-centred design

If the parallel runtime were to be redesigned with the requirements of the expand operators taken in consideration from the very beginning, we feel the the result would not only be more maintainable than the current parallel runtime, but perhaps also more efficient.

6.5.3 Additional query operations

Finally, the parallel runtime only supports a very small subset of the query operators present in Neo4j. This set needs to be greatly expanded before the parallel runtime is viable for any usage in production.

Bibliography

[1] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Commun. ACM*, vol. 35, pp. 85–98, June 1992.

[2] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 743–754, ACM, 2014.

[3] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," *SIGMOD Rec.*, vol. 19, pp. 102–111, May 1990.

[4] "Robert haas: Parallelism progress." `http://rhaas.blogspot.se/2013/10/parallelism-progress.html`. Accessed: 2016-05-20.

[5] R. Haas and K. Amit, "Parallel sequential scan." Talk at the PostgreSQL Conference 2015. `https://www.pgcon.org/2015/schedule/events/785.en.html` Accessed: 2016-05-20.

[6] "Robert haas: Parallel sequential scan is committed!." `http://rhaas.blogspot.se/2015/11/parallel-sequential-scan-is-committed.html`. Accessed: 2016-05-20.

[7] "Robert haas: Parallel query is getting better and better." `http://rhaas.blogspot.se/2016/03/parallel-query-is-getting-better-and.html`. Accessed: 2016-05-20.

[8] R. Geist, "Understanding parallel execution - part 1." `http://www.oracle.com/technetwork/articles/database-performance/geist-parallel-execution-1-1872400.html`. 2012, Accessed: 2016-09-09.

[9] R. Geist, "Understanding parallel execution - part 2." http://www.oracle.com/technetwork/articles/database-performance/geist-parallel-execution-2-1872405.html. 2012, Accessed: 2016-09-09.

[10] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

[11] J. A. Hoffer, R. Venkataraman, and H. Topi, *Modern database management*. Upper Saddle River, N.J: Prentice Hall, 2011.

[12] S. Sumathi, *Fundamentals of relational database management systems*. Berlin London: Springer, 2010.

[13] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.

[14] "The neo4j developer manual v3.0." http://neo4j.com/docs/developer-manual/3.0/#graphdb-concepts. Accessed: 2016-05-25.

[15] "openCypher · openCypher." http://www.opencypher.org/. Accessed: 2016-05-10.

[16] P. Boncz, "Ldbc: Benchmarks for graph and rdf data management," in *Proceedings of the 17th International Database Engineering & Applications Symposium*, IDEAS '13, (New York, NY, USA), pp. 1–2, ACM, 2013.

[17] "Ldbcouncil |." http://ldbcouncil.org/. 2016-05-11.

[18] "Linked data benchmark council." https://github.com/ldbc. Accessed: 2016-05-11.

[19] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The ldbc social network benchmark: Interactive workload," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 619–630, ACM, 2015.

[20] "Ldbc-snb datagen|." http://ldbcouncil.org/blog/datagen-data-generation-social-network-benchmark. 2016-09-20.

[21] A. Prat-Pérez and D. Dominguez-Sal, "How community-like is the structure of synthetically generated graphs?," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, (New York, NY, USA), pp. 7:1–7:9, ACM, 2014.

[22] "Ldbc-snb |." http://ldbcouncil.org/benchmarks/snb. 2016-09-20.

[23] "Openjdk: jmh." http://openjdk.java.net/projects/code-tools/jmh/. Accessed: 2016-09-08.

[24] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic microbenchmark generation to prevent dead code elimination and constant folding," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 132–143, ACM, 2016.

[25] "Jmh vs caliper: reference thread - google groups." `https://groups.google.com/forum/#!topic/mechanical-sympathy/m4opvy4xq3U/discussion`. Accessed: 2016-09-08.

[26] A. Shipilev, "The art of java benchmarking." `https://vimeo.com/78900556`. Oredev 2013-11-07, Accessed: 2016-09-08.

[27] "DB-Engines Ranking - popularity ranking of database management systems." `http://db-engines.com/en/ranking`. Accessed: 2016-05-20.

[28] F. Holzschuher and R. Peinl, "Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, (New York, NY, USA), pp. 195–204, ACM, 2013.

[29] F. Holzschuher and R. Peinl, "Querying a graph database – language selection and performance considerations," *Journal of Computer and System Sciences*, vol. 82, no. 1, Part A, pp. 45 – 68, 2016. Special Issue on Query Answering on Graph-Structured Data.

[30] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing n-relational joins," *ACM Trans. Database Syst.*, vol. 9, pp. 482–502, Sept. 1984.

[31] D. Kossmann and K. Stocker, "Iterative dynamic programming: A new class of query optimization algorithms," *ACM Trans. Database Syst.*, vol. 25, pp. 43–82, Mar. 2000.

[32] "The neo4j developer manual v3.0." `http://neo4j.com/docs/developer-manual/3.0/#execution-plans`. Accessed: 2016-05-17.

[33] "GitHub - neo4j_schema.svg." `https://github.com/neo-technology/ldbc_snb_workload_interactive_neo4j/blob/3.1/schema/neo4j_schema.svg`. Accessed: 2016-11-28.

Appendices

Appendix A

Evaluation results

This chapter contains the measured results of our small query benchmark runs. The tables show the number of cores, result (execution time in seconds), error (standard error), group (Interpreted/Parallel runtime with Hyper-Threading or dual-socket configuration), and the speed-up factor for parallel runtime.

sq1_results

#Cores	Result	Error	Group	Parallel Speedup
1	51.7375787008000	2.603072044869490	Interpreted	0.3265885688
2	44.5558117717333	0.589485256933831	Interpreted	0.4794275028
4	44.7769514569143	1.410105148869580	Interpreted	0.6924275017
8	43.1483151974400	1.295690935567100	Interpreted	0.8613277698
12	43.4086975897600	0.691282111179007	Interpreted	1.1420834495
16	40.4863302587733	0.455367200179784	Interpreted	1.2693060903
20	41.0625717043200	0.770204803243277	Interpreted	1.516374856
22	40.9176165580800	0.291931764988163	Interpreted	1.7042194142
1	158.4182168024620	39.006866734261200	Parallel	
2	92.9354521993846	14.986360479805300	Parallel	
4	64.6666276944314	5.721256910310390	Parallel	
8	50.0951167580328	2.990890293269190	Parallel	
12	38.0083413423158	1.540153092675650	Parallel	
16	31.8964279528247	1.333508948574170	Parallel	
20	27.0794332569600	0.824578717971269	Parallel	
22	24.0095941973333	0.642806471198322	Parallel	
1	43.9446737169067	1.009599290874580	Interpreted HT	0.3148791132
2	43.8757752832000	1.504030295826280	Interpreted HT	0.4947164101
4	41.4858049399467	0.877519912687479	Interpreted HT	0.7299041045
8	43.2207927705600	0.430267311267751	Interpreted HT	1.1705257692
12	42.8333509290667	1.442302635181110	Interpreted HT	1.6120464955
16	40.3914830643200	0.337231694668282	Interpreted HT	1.9660496867
20	39.9512489164800	0.444424429315572	Interpreted HT	2.0340284492
22	41.9654096213333	0.399533198490517	Interpreted HT	2.1590945169
1	139.5604594952260	20.955705897231800	Parallel HT	
2	88.6887404410435	5.748211420936180	Parallel HT	
4	56.8373361427692	2.682045659733550	Parallel HT	
8	36.9242556708571	2.533200861676050	Parallel HT	
12	26.5707912567129	0.949772202464507	Parallel HT	
16	20.5444874240000	0.451844150979395	Parallel HT	
20	19.6414405764354	0.346043182747406	Parallel HT	
22	19.4365782934069	0.411808563376046	Parallel HT	
2	46.8084326400000	1.174256046326060	Interpreted 2 sockets	0.4545701407
4	45.6992977849863	1.281868669168840	Interpreted 2 sockets	0.6522351232
8	44.6989773482667	0.640496375170460	Interpreted 2 sockets	0.8627806563
16	41.7488716868267	0.435290693353756	Interpreted 2 sockets	1.1982792937
24	41.8634041480533	0.771264833784737	Interpreted 2 sockets	1.6604555149
32	43.4471733384533	0.911727902243424	Interpreted 2 sockets	1.9377366999
40	42.0360976247467	0.717950427032157	Interpreted 2 sockets	2.0072635604
44	41.4339074184533	0.802609594258328	Interpreted 2 sockets	2.051285721
2	102.9729594026670	15.181112477843100	Parallel 2 sockets	
4	70.0656805478400	6.620005025528990	Parallel 2 sockets	
8	51.8080430080000	3.574073101909460	Parallel 2 sockets	
16	34.8406852266667	1.362360899311980	Parallel 2 sockets	
24	25.2119998238532	0.436854530348620	Parallel 2 sockets	
32	22.4216083333120	0.337223186017267	Parallel 2 sockets	
40	20.9419920998400	0.258832269363484	Parallel 2 sockets	
44	20.1989937309538	0.219929116194282	Parallel 2 sockets	
Max. Speedup				2.1590945169

sq2_results

#Cores	Result	Error	Group	Parallel Speedup
1	102.3590224171710	4.591025274189470	Interpreted	1.1617020265
2	100.1227975818380	6.188647616655800	Interpreted	2.1051502639
4	97.3045791968780	4.682779110732640	Interpreted	3.6984508022
8	92.3972346212174	3.969348889943980	Interpreted	5.815846786
12	91.7721171740445	4.315484767450020	Interpreted	8.2570155924
16	92.5188500371064	4.088241439619370	Interpreted	10.3476929607
20	95.1695203607273	5.263161793410090	Interpreted	12.7579418397
22	92.1533209055319	3.776764891882810	Interpreted	13.0201128561
1	88.1112540774400	1.538275916880840	Parallel	
2	47.5608792863562	0.764485167915958	Parallel	
4	26.3095507821714	0.570581594291866	Parallel	
8	15.8871507488555	0.147872284876995	Parallel	
12	11.1144415493447	0.148440324172658	Parallel	
16	8.9410122999172	0.097607151961612	Parallel	
20	7.4596295826286	0.033164151069612	Parallel	
22	7.0777666771570	0.034206762130009	Parallel	
1	95.3145767543830	4.393560020962160	Interpreted HT	1.2500421337
2	98.8083381694359	3.888928406463770	Interpreted HT	2.4097002374
4	94.4601027450435	3.852466762171400	Interpreted HT	4.0267969714
8	95.6179295883636	5.354227078904780	Interpreted HT	7.1455004803
12	93.7439437671489	3.134223894276870	Interpreted HT	9.798636483
16	92.5583754705455	5.200053563970920	Interpreted HT	12.3454807852
20	96.3045752832000	6.504485159027500	Interpreted HT	14.5850012122
22	93.0397290496000	4.721185995149550	Interpreted HT	13.7927345294
1	76.2490912768000	3.164303963160340	Parallel HT	
2	41.0044106888533	0.810330434275450	Parallel HT	
4	23.4578756802645	0.293581163395137	Parallel HT	
8	13.3815580661709	0.125042111513680	Parallel HT	
12	9.5670396518400	0.113150143611575	Parallel HT	
16	7.4973487935073	0.055591628272878	Parallel HT	
20	6.6029871291603	0.044268515571261	Parallel HT	
22	6.7455607770471	0.038777713561647	Parallel HT	
2	96.6815034026667	4.295961528356290	Interpreted 2 sockets	1.8074858856
4	98.2439354157949	4.490604789634930	Interpreted 2 sockets	3.4260442732
8	99.7527919532973	3.259219803759640	Interpreted 2 sockets	5.9544929458
16	97.9278108769524	5.246808603937030	Interpreted 2 sockets	9.9249356153
24	98.7220493486829	5.203489854199670	Interpreted 2 sockets	12.2279406456
32	94.0957785367273	5.593329998952560	Interpreted 2 sockets	13.5818986684
40	93.2959098434783	4.304404559621160	Interpreted 2 sockets	15.4299172491
44	93.8948877165714	4.874542249737100	Interpreted 2 sockets	15.3988606609
2	53.4894928782222	1.629241293850830	Parallel 2 sockets	
4	28.6756175872000	0.399845157542328	Parallel 2 sockets	
8	16.7525250028679	0.146604494696998	Parallel 2 sockets	
16	9.8668459597576	0.089129403929835	Parallel 2 sockets	
24	8.0734812353208	0.090270285647818	Parallel 2 sockets	
32	6.9280283142827	0.081675003271167	Parallel 2 sockets	
40	6.0464296947953	0.052698504709900	Parallel 2 sockets	
44	6.0975217442660	0.040192106926521	Parallel 2 sockets	
Max. Speedup				15.4299172491

sq3_results

#Cores	Result	Error	Group	Parallel Speedup
1	108.0972262895480	7.188817553813780	Interpreted	2.0745814539
2	97.0459645502439	3.029331009355150	Interpreted	3.5086122155
4	91.6653395148800	1.362727629873600	Interpreted	3.7327969742
8	89.9419838873600	1.564548037606690	Interpreted	4.5343003648
12	90.9029828198400	1.335283899188850	Interpreted	3.2558716365
16	88.5085385523200	0.613110342476589	Interpreted	2.8260411963
20	88.6266501529600	0.458650762635206	Interpreted	3.0106575783
22	89.9392995328000	0.379257605603497	Interpreted	3.3235169908
1	52.1055589717333	2.525465774187690	Parallel	
2	27.6593589113535	0.758868011351117	Parallel	
4	24.5567439503360	5.924161073843540	Parallel	
8	19.8359121920000	3.593478453331130	Parallel	
12	27.9197072147692	3.762771032602650	Parallel	
16	31.3189130680430	2.404526488683390	Parallel	
20	29.4376387379200	0.653855400960734	Parallel	
22	27.0614832969505	0.624159000724954	Parallel	
1	94.5724955033600	1.072189205754890	Interpreted HT	0.8055222623
2	92.6746568294400	0.969490633980406	Interpreted HT	1.4308444403
4	90.0815703244800	0.930718096057494	Interpreted HT	2.8297197306
8	88.6883903078400	1.971253541617330	Interpreted HT	3.8332071764
12	92.6818151082667	3.841101931539370	Interpreted HT	3.3820381653
16	88.8494515814400	1.060665750539070	Interpreted HT	3.7604746089
20	87.8696621670400	0.644527938050019	Interpreted HT	3.8599139202
22	88.0924635955200	0.589198646237445	Interpreted HT	3.9322046084
1	117.4051915452630	31.900919289310800	Parallel HT	
2	64.7692049687273	16.626946943232800	Parallel HT	
4	31.8340962713600	6.273139500147690	Parallel HT	
8	23.1368632650323	3.475238258832950	Parallel HT	
12	27.4041304615385	2.032468178002940	Parallel HT	
16	23.6271909326452	0.436906967175828	Parallel HT	
20	22.7646688460800	0.282875850793187	Parallel HT	
22	22.4028178513920	0.427689956239847	Parallel HT	
2	101.0252771452120	4.808312579168200	Interpreted 2 sockets	3.3770545982
4	97.2380595814400	0.764459318786519	Interpreted 2 sockets	4.2638688525
8	94.8949840989091	2.075637462480910	Interpreted 2 sockets	5.2433363063
16	95.8725448515919	1.394235367935170	Interpreted 2 sockets	3.8656174576
24	95.2033188249600	1.068025897746650	Interpreted 2 sockets	3.8925918523
32	95.1657378611200	1.130042626866550	Interpreted 2 sockets	3.8498311333
40	91.8935096524800	2.347481122293310	Interpreted 2 sockets	3.8184774957
44	95.3986740976327	1.398013818451530	Interpreted 2 sockets	4.0862318344
2	29.9152039761702	1.738321569601740	Parallel 2 sockets	
4	22.8051243943307	4.257740560249030	Parallel 2 sockets	
8	18.0982066674872	2.797937408141960	Parallel 2 sockets	
16	24.8013534455172	3.286929070436780	Parallel 2 sockets	
24	24.4575651486897	1.634579916851640	Parallel 2 sockets	
32	24.7194577027523	0.881283305536306	Parallel 2 sockets	
40	24.0654841506341	0.459762114092037	Parallel 2 sockets	
44	23.3463684792320	0.367983975008637	Parallel 2 sockets	
Max. Speedup				5.2433363063

sq4_results

#Cores	Result	Error	Group	Parallel Speedup
1	135.3256573584910	86.295091467969200	Interpreted	2.7554467824
2	89.1290521825055	35.839428842894700	Interpreted	3.7570760216
4	57.9225158274510	19.579272776062800	Interpreted	4.0323760273
8	41.0000663829533	11.688422551963600	Interpreted	4.2301203992
12	34.1570959159216	8.116314060630900	Interpreted	4.2148900201
16	31.1482996053333	6.283242943928350	Interpreted	4.0866038531
20	27.9925306957576	5.091271047505050	Interpreted	3.5732566761
22	25.0598626357895	3.176417108290810	Interpreted	3.2001492542
1	49.1120562449067	14.005373408625700	Parallel	
2	23.7229834240000	0.512304877318731	Parallel	
4	14.3643636989247	0.292343761105292	Parallel	
8	9.6924112114036	0.174416624348990	Parallel	
12	8.1039115500308	0.143037215002617	Parallel	
16	7.6220501730233	0.216845712658008	Parallel	
20	7.8338986625542	0.500956381884809	Parallel	
22	7.8308418280727	0.475223588009562	Parallel	
1	103.1250728277330	57.156795140893800	Interpreted HT	3.2191330344
2	71.8577379530550	29.011028927705900	Interpreted HT	4.0430080787
4	46.7155878524272	15.017949642207400	Interpreted HT	4.1607168763
8	33.3991605107451	7.657284114232470	Interpreted HT	4.1030885169
12	28.8173764143505	5.423201914931850	Interpreted HT	4.0116288423
16	23.6572167372800	2.564291618033900	Interpreted HT	3.2138472038
20	23.0869256110080	2.387412740072440	Interpreted HT	3.0715554448
22	23.1885284311040	2.373396125480960	Interpreted HT	3.1278971586
1	32.0350453760000	0.835690871537820	Parallel HT	
2	17.7733352379733	0.330928620441737	Parallel HT	
4	11.2277737806009	0.211470871770053	Parallel HT	
8	8.1400048703210	0.136618674950110	Parallel HT	
12	7.1834602719106	0.122909845201603	Parallel HT	
16	7.3610272166328	0.310726172499549	Parallel HT	
20	7.5163629716406	0.399910539509750	Parallel HT	
22	7.4134561512727	0.363547380047310	Parallel HT	
2	74.8431605760000	39.641625995984400	Interpreted 2 sockets	2.6412703676
4	63.6827039325591	22.956876716647500	Interpreted 2 sockets	3.6325154836
8	45.5162446158368	12.324708020958800	Interpreted 2 sockets	3.9189050555
16	33.4228720487921	6.043752346712970	Interpreted 2 sockets	3.6537080764
24	28.1766631833600	3.838903898182540	Interpreted 2 sockets	3.3552029614
32	25.2475747979636	2.478316527598710	Interpreted 2 sockets	3.0407961075
40	25.1071821422056	2.439596356154660	Interpreted 2 sockets	2.9890622508
44	25.0173985395315	2.345827709522370	Interpreted 2 sockets	2.9985596455
2	28.3360467353600	0.828770580705732	Parallel 2 sockets	
4	17.5312959351467	0.422029728053291	Parallel 2 sockets	
8	11.6145310924800	0.278709643545974	Parallel 2 sockets	
16	9.1476580366525	0.189412341260289	Parallel 2 sockets	
24	8.3979012618039	0.186441014063915	Parallel 2 sockets	
32	8.3029489334511	0.210119972719906	Parallel 2 sockets	
40	8.3996852644103	0.274301275218233	Parallel 2 sockets	
44	8.3431385388026	0.267703877414175	Parallel 2 sockets	
Max. Speedup				4.2301203992

sq5_results

#Cores	Result	Error	Group	Parallel Speedup
1	220.0454911317330	116.689220147353000	Interpreted	3.2103600428
2	111.3885126283640	28.487079138115000	Interpreted	3.2291004595
4	96.6703185920000	24.560453302192500	Interpreted	4.8532549695
8	77.1233367505455	12.635241279770700	Interpreted	5.8761085394
12	73.0240310125714	9.708726119369250	Interpreted	6.3238318574
16	64.1614426931200	6.524607389031380	Interpreted	5.4931341569
20	62.5293551206400	5.141143730130940	Interpreted	4.9884174394
22	61.7978685030400	3.290373255475440	Interpreted	4.7994667204
1	68.5423093350400	2.718764618258830	Parallel	
2	34.4952143872000	0.686712686418548	Parallel	
4	19.9186564892444	0.248213435918878	Parallel	
8	13.1248999628800	0.104960541986731	Parallel	
12	11.5474340019649	0.171283748415780	Parallel	
16	11.6802977792000	0.274429383375299	Parallel	
20	12.5349082910622	0.216442588675934	Parallel	
22	12.8759864592077	0.240597778320894	Parallel	
1	132.3655233536000	49.438708357859100	Interpreted HT	2.4737263068
2	105.4264646727440	31.996769363951600	Interpreted HT	3.6447462309
4	79.0298385687273	14.670789004104000	Interpreted HT	4.6965438831
8	68.6644474675200	6.901187519018210	Interpreted HT	6.0814254427
12	64.7385789235200	5.314340379517660	Interpreted HT	6.0479777652
16	58.9121873510400	2.719487347771130	Interpreted HT	4.6003201465
20	60.0490115072000	2.193854780133700	Interpreted HT	4.3779168255
22	59.0584846745600	2.765764135222520	Interpreted HT	4.184101867
1	53.5085562976604	1.872428488874650	Parallel HT	
2	28.9255981056000	0.438437608871721	Parallel HT	
4	16.8272330752000	0.135150569742513	Parallel HT	
8	11.2908475347478	0.094194681599835	Parallel HT	
12	10.7041694657814	0.212508711589955	Parallel HT	
16	12.8061059827573	0.241077444997548	Parallel HT	
20	13.7163436176080	0.186925034580616	Parallel HT	
22	14.1149729503179	0.143728041335781	Parallel HT	
2	136.7244414494120	48.073928865092000	Interpreted 2 sockets	3.6106634197
4	103.4288054653020	23.872431882476700	Interpreted 2 sockets	4.7932713672
8	88.3386750928372	17.741522277070200	Interpreted 2 sockets	6.2212646891
16	70.3958561587200	6.123363574464190	Interpreted 2 sockets	6.4435310077
24	66.2646344908800	3.580206676808860	Interpreted 2 sockets	5.5287706004
32	64.9305102745600	3.258008161065660	Interpreted 2 sockets	4.4485172222
40	63.4044547072000	3.085763780780900	Interpreted 2 sockets	4.0014278713
44	63.7735534592000	2.727882943185820	Interpreted 2 sockets	4.0054553337
2	37.8668476006400	0.915847187279720	Parallel 2 sockets	
4	21.5779156951040	0.325577286080034	Parallel 2 sockets	
8	14.1994722146878	0.132868717041492	Parallel 2 sockets	
16	10.9250434389333	0.095321165915031	Parallel 2 sockets	
24	11.9854194141867	0.165820545739397	Parallel 2 sockets	
32	14.5959894119551	0.127770184909761	Parallel 2 sockets	
40	15.8454573582629	0.127126165906616	Parallel 2 sockets	
44	15.9216738538057	0.144130964307417	Parallel 2 sockets	
Max. Speedup				6.4435310077

EXAMENSARBETE Multi-threaded execution of Cypher queries
STUDENT Ragnar Mellbin, Felix Åkerlund
HANDLEDARE Per Andersson (LTH), Johan Svensson (Network Engine for Objects in Lund AB)
EXAMINATOR Flavius Gruian (LTH)

Parallelliserad sökning i grafdatabas

POPULÄRVETENSKAPLIG SAMMANFATTNING **Ragnar Mellbin, Felix Åkerlund**

Grafdatabaser blir allt vanligare, samtidigt som antalet processorer i moderna datorer ökar mer och mer. Vi tittar i detta arbete på hur parallelliserad sökning kan leda till prestandavinster i den populära grafdatabashanteraren Neo4j.

Datorer är idag tusentals gånger kraftfullare än de var för tjugo år sedan, mycket tack vare de framsteg som gjorts i tillverkningen av centralprocessorn, den komponent som står för utförandet av alla logiska beräkningar i maskinen. På senare tid har dock fysikaliska begränsningar satt hinder för hur fort man kan köra processorn, därför har man istället börjat bygga datorer som innehåller flera processorer. Detta gör det möjligt för datorn att arbeta på flera uppgifter samtidigt. För att dra full nytta av denna sortens design, krävs dock att mjukvaruutvecklare skriver sina program så att de kan delas upp i mindre beståndsdelar som kan utföras parallellt.

En typ av databas som vuxit i popularitet på senare tid är den så kallade grafdatabasen. Grafdatabaser gör sig av med den traditionella tabellstrukturen och använder istället noder och bågar för att representera data. Ett exempel på data som lämpar sig för denna struktur är sociala nätverk som Facebook eller Twitter. Varje användare i nätverket kan representeras av en nod, medan vänskapsrelationer eller följare representeras av bågar som binder samman noderna.

Den populära grafdatabasen Neo4j har i dagsläget stöd för att besvara flera sökningar parallellt och klarar på så sätt av att utnyttja en modern processor fullt ut. Varje enskild sökning körs dock bara som en enda uppgift, vilket innebär att processorn kan ha delar som står outnyttjade om antalet aktiva sökningar är för lågt.

För simpla frågor som tar millisekunder att besvara så är detta sällan ett problem, speciellt när det finns tusentals användare som använder databasen samtidigt. Vill man däremot beräkna något tyngre, så som att analysera väldigt stora mängder data, är det vanligt att man sitter som ensam användare och kör något som tar flera timmar eller till och med dagar att få svar på. Det är i detta fallet intressant att se om man på något sätt kan få sökningen att utnyttja all den extra processorkraft som annars går till spillo.

För att ta reda på om det går att parallellisera en enskild sökning i en grafdatabas och hur stor påverkan detta då har på svarstider, skapade vi vår egen modifierade version av Neo4j. Vi började med att ta reda på vilka delar av mjukvaran som bäst lämpade sig för parallellisering, med hänsyn till hur ofta de förekom i sökningar samt hur pass stora krav de ställde på processorn. Efter att ha valt ut ett antal av dessa så gick vi vidare med att ta fram metoder för att dela upp dem i mindre uppgifter som kunde köras i olika delar av processorn samtidigt, för att slutligen införa dessa ändringar i Neo4j.

Resultatet är en version av Neo4j som under rätt förhållanden ger upp till 15 gånger snabbare svar på enskilda sökningar.