MASTER’S THESIS | LUND UNIVERSITY 2016

The Shortcut Index

Anton Persson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-03

The Shortcut Index

(Path Indexing in Graph Databases)

Anton Persson
antonjpersson@gmail.com

January 19, 2016

Master's thesis work carried out at Neo Technology.

Supervisors: Krzysztof Kuchcinski, krzysztof.kuchcinski@cs.lth.se
Johan Svensson, johan@neotechnology.com

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

Abstract

With a novel path index design, called the Shortcut Index, we partially solve the problem of executing traversal queries on dense neighborhoods in a graph database. We implement our design on top of the graph database Neo4j but it could be used for any graph database that uses the labeled property graph model.

By using a B+ tree, the Shortcut Index can achieve what we call *neighborhood locality* and *range locality* of paths. This means that data that belongs to the same part of the graph is located in the same space on disk. We empirically evaluate how this affects performance in terms of response time. In our benchmarks we use two different datasets, one that simulates a real world use case and a "lab environment" that makes it possible to vary *neighborhood density* and *percent of neighborhood interest* to more accurately examine how it affects performance. Our experiments show that response time of the index scales very well with *neighborhood density* and *percent of neighborhood interest* when compared to Neo4j without the index.

We put focus on making the Shortcut Index useful in an OLTP (online transaction processing) environment, which enforces restrictions on update overhead which in turn restricts how long paths that can be indexed. The presented design can index arbitrary long paths but in our implementation we only index paths of length one.

We conclude that the Shortcut Index improves response time at a reasonable cost and is especially useful when indexing dense neighborhoods that are often queried with limitations on some property value.

Keywords: path index, labeled property graph, graph database, neighborhood locality, range locality, neighborhood density, percent of neighborhood interest, oltp, B+ tree, ldbc, neo4j, graph density

Acknowledgements

This work would not have been possible for me without the help from a lot of people. I can not list all of them here but I would like to give a special thanks to some of them:

- *Krzysztof Kuchcinski*, my supervisor, for continuous feedback and consultation.
- *Per Andersson*, for serving as my examinator.
- *Flavius Gruian*, for stepping in as examinator during my presentation.
- *Alex Averbuch*, for the idea and many fruitful discussions.
- *Johan Svensson* and *Magnus Vejlstrup*, for supporting the idea.
- *Max Sumrall*, for first exploring the topic of path indexing.
- *Davide Grohmann*, for making me focus on the most important things first.
- *Petra Selmer*, for proof reading with the eyes of a hawk.
- *Mattias Persson, Chris Vest* and *Andrés Taylor*, for sharing their in depth knowledge.
- *Neo4j engineering staff*, for general support and interest.
- *Björn Elmers*, for companionship through the last 5.5 years.
- *My parents*, for everything.

Contents

1 Introduction **9**
1.1 Purpose . . . 9
1.1.1 Typical use case . . . 9
1.2 Terminology and declarations . . . 10
1.2.1 Property graph model . . . 10
1.2.2 Notation . . . 12
1.2.3 Paths and segments . . . 12
1.2.4 Patterns . . . 12
1.2.5 Schema . . . 13
1.2.6 Dense nodes and neighborhoods . . . 13
1.3 Graph database . . . 15
1.3.1 Relational database . . . 15
1.3.2 Neo4j . . . 16
1.3.3 Result rows and database queries . . . 16
1.3.4 Database indexes . . . 17
1.4 Problem statement . . . 17
1.4.1 Density is hard . . . 17
1.4.2 About LDBC . . . 18
1.4.3 Our statement . . . 19
1.5 Previous work . . . 19
1.5.1 K-path indexing . . . 19
1.5.2 Database cracking and adaptive merging . . . 20
1.5.3 B+ trees . . . 20
1.5.4 Bitmap index . . . 20
1.5.5 R-Tree . . . 21
1.5.6 Composite indexes . . . 21

2 Approach **23**
2.1 The Shortcut Index . . . 23
2.1.1 Defining a Shortcut index . . . 23

2.1.2 Keys and values 24
2.1.3 What is a B+ tree? 24
2.1.4 How do we use the B+ tree 27
2.1.5 Why do we use a B+ tree? 30
2.2 Usefulness 32
2.2.1 Solve the problem 33
2.2.2 Indexing in an OLTP environment 33
2.2.3 Using path index as composite index 34
2.3 Implementation 34
2.3.1 Partial implementation 35
2.3.2 Store layout 35
2.3.3 Neo4j page cache and page fault 38

3 Evaluation approach 39
3.1 How do we evaluate performance? 39
3.1.1 The workload 39
3.2 Machine and tools 40
3.3 Environment setup 40
3.3.1 No benchmarks on limited RAM 40
3.4 Datasets 41
3.4.1 LDBC Dataset 41
3.4.2 LAB Dataset 41
3.4.3 Cost analysis of range queries 43
3.5 Insert time 44
3.5.1 What is insert time overhead? 44
3.6 Memory overhead 45

4 Result and discussion 47
4.1 How did we decide what queries to run? 47
4.2 Result tables 47
4.3 Query analysis 48
4.3.1 The LDBC queries 48
4.3.2 The LAB queries 50
4.3.3 The Holy Grail 53
4.3.4 Result for first query to return 54
4.4 Insert times 54
4.4.1 Page cache hit rate vs page cache coverage 55
4.4.2 Insert time vs page cache coverage 55
4.5 Memory overhead 56

5 Conclusions 59
5.1 Response time improvement 59
5.2 Scaling with percent of neighborhood interest 60
5.3 Insert time 60
5.4 Limitations and unknowns 60
5.4.1 Implementation 60
5.4.2 Linear vs binary search 60

5.4.3 Response time as performance . . . 60
5.4.4 Configuration . . . 61
5.4.5 Datasets and environment . . . 61
5.4.6 Response time for "first" query . . . 61
5.5 Did we solve the problem? . . . 62
5.6 Final conclusion . . . 62
5.7 Future work . . . 62

6 Bibliography 65

Appendix A Target queries 69
A.1 LDBC Queries . . . 69
A.2 LAB Queries . . . 72

Appendix B Result tables 75

Appendix C Dataset statistics 81

Chapter 1
Introduction

We present the purpose of this project in section 1.1. In section 1.2 we define most of the terminology and define some useful notation and concepts. We present briefly what a graph database is and how it differs from a relational database in section 1.3. In that section we also talk about what a query is. We define the problem statement in section 1.4. Finally in section 1.5 we present what previously has been done on the subject of indexing in databases and in particular path indexes.

Before we continue with the actual report we want to point out that our full implementation is available in a public repository on GitHub, see [18].

1.1 Purpose

The purpose of this project is to investigate if and how the density problem in graph databases can be solved by indexing paths. The density problem is further described in section 1.4.1 but we can think of it like this: When we query a graph database for some data that belongs to a neighborhood (group of neighboring nodes) with a lot of nodes and the data is only a fraction of the entire neighborhood, we end up doing a lot of unnecessary work which slows down the query execution. Our hypothesis is that by indexing paths in such a "dense" neighborhood we can get around this problem and we can expect better performance in terms of query execution time or response time.

To make sure our proposal is of relevance to database vendors we also analyze and adapt our solution to the restrictions enforced by an OLTP environment.

1.1.1 Typical use case

We will now introduce a typical use case to give you, the reader, a sense of direction in the following sections. Our hope is that this will make it easier for you to digest the terminology and definitions by having some concrete example to attach them to.

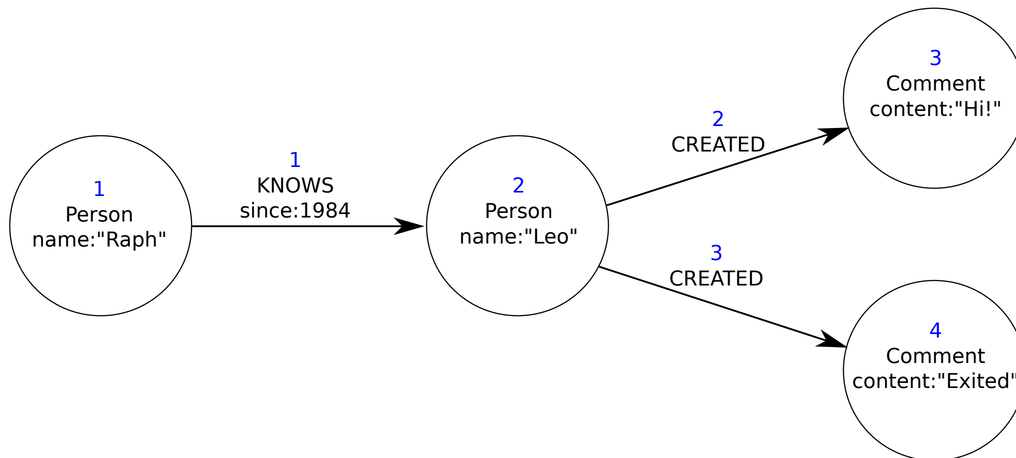


Figure 1.1: A simple example of a social network illustrated as a graph.

The scenario: You develop and maintain a social network website. This social network consist of "Persons" that "know" each other and "Comments" that are "created" by the "Persons". We use quotation marks here because we will later introduce a more rigid syntax. A very small subset of this domain is shown in figure 1.1.

The number of "Comments" created by each "Person" is typically large. To keep track of all this data you use a graph database. You want to "ask" your database for data, e.g. "give me all comments created by Raph's friends". The database is expected to answer these questions, or queries as they are called, within a very short amount of time even if the number of "Persons" and "Comments" is large.

1.2 Terminology and declarations

This section introduces useful terminology around graphs that will be used in this paper. In 1.2.1 we give a quick overview of what a graph is and in particular the features of the property graph model. Some notation is introduced in section 1.2.2. Section 1.2.3 and 1.2.4 explains the concept of a path, segment and pattern. And finally in section 1.2.6 we describe what a dense node / neighborhood is.

1.2.1 Property graph model

The mathematical definition of an undirected graph is an ordered pair $G = (V, E)$ where V is a set of vertices and E is a set of edges. Every edge $e \in E$ is an unordered pair of vertices $v \in V$. Two vertices both belonging to the same edge are said to be incident to each other and end points of the edge. In this description the edges do not have a direction and thus G is said to be an undirected graph. By letting the edges instead be an ordered pair of two vertices, a directed graph is acquired.

This is how a graph as presented in [19]:

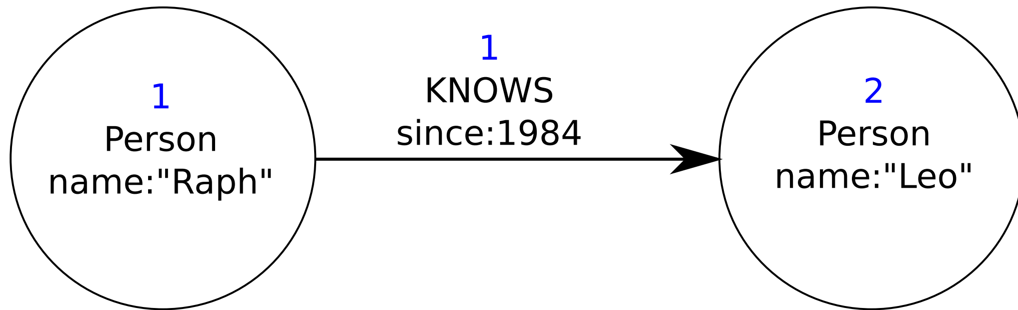


Figure 1.2: A simple graph illustrating a social network

> Formally, a graph is just a collection of vertices and edges - or, in less intimidating language, a set of nodes and the relationships that connect them. Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships.

We will use "node" and "relationship" consistently in this report in favor over "vertex" and "edge", as this is the terminology used by the graph database neo4j which this project is closely tied to. Further readings about neo4j can be found in section 1.3.2.

Depending on what the graph of interest is used for, different graph models are used. Directed and undirected graphs are such examples. However, they all have nodes and relationships as their common core. In *this* project we work with "The Labeled Property Graph Model" as described by [19], and when we talk about graphs from here on it is a "Labeled Property Graph" that we mean. The Labeled Property Graph Model has a set of features:

- A graph is made up of nodes, relationships, labels and properties.
- Both nodes and relationships can have properties in the form of key-value pair. A string is used as key and the value can be a string, a primitive data type or arrays of the same.
- Nodes can have zero or more labels that are used to group nodes together.
- Relationships connect nodes and they always have a direction and a name, or more formally, a relationship type.
- A relationship always has a start and an end node indicating the direction. There are no dangling relationships.

Seen from a node's perspective a relationship can have the directions *INCOMING* (meaning the node is the end node of the relationship) or *OUTGOING* (meaning the node is the start node of the relationship). To uniquely identify a node or relationship they have id numbers. In the small example graph in figure 1.2 the node with id 1 is labeled as *"Person"* and has a property with key *name* and value *"Raph"*. The relationship between the nodes indicates that Raph knows Leo and has done so since 1984. In this way relationships give semantic knowledge about how the nodes relate to each other.

1.2.2 Notation

To make it easy to talk about nodes and relationships, we introduce a formal notation. Nodes will be surrounded by parentheses and relationships will be surrounded by square brackets. To indicate a relationship's relation to nodes a dash (-) will be used and brackets (< or >) will show the direction of the relationship. If no brackets are used, direction is not important. Inside nodes `()` and relationships `[]` we can have three different types of information:

- Some identifier that could be thought of as a variable name, `(Raph)`.
- A label for nodes or relationship type for relationships. Those are always preceded by a colon (:), e.g. `(Raph:Person)` or `[friendship:KNOWS]`.
- A list of properties surrounded by curly brackets, `(Raph:Person {age:15})`.

Any of those three can be left out. The graph in figure 1.2 can with this notation be written as:

```
(raph:Person {name:"Raph"})-[:KNOWS {since:"1984"}]->(leo:Person {name:"Leo"}).
```

This representation is quite detailed and often a simpler description will suffice, for example `(Raph)-[:KNOWS]->(Leo)`. Another way to refer to nodes and relationship is to use the ids. We will do this in a similar way. Instead of identifier, label / relationship type and property, we use only the id. Figure 1.2 can then be written as `(1)-[1]->(2)`.

The notation described here is derived from the syntax used in the Cypher query language. Cypher is a language used and developed together with the graph database neo4j. More about Neo is presented in section 1.3.2 and queries are presented in section 1.3.3.

1.2.3 Paths and segments

Let n_i denote a node and r_j denote a relationship. A path is a sequence of alternating nodes and relationships that begins and ends with at node, $[n_0, r_0, n_1, ..., n_n]$. Relationship r_j need to have nodes n_j and n_{j+1} as endpoints. All relationships and nodes need to be distinct (not appearing more than once), which in graph theory is called a simple path. The length of a path is defined to be equal to the number of relationships in the path.

We define a segment to be a sequence of alternating nodes and relationships, *without* it having to start or end in a node. A sequence could thus look like $[n_0, r_0, n_1, ..., r_n]$. We can think of a segment as "part of a path". A segment from figure 1.2 could be `(1)-[1]->`.

1.2.4 Patterns

A pattern is a sequence of node labels, relationship types with direction and property keys. As a pattern does not point to any specific nodes or relationships, identifiers and property values are never needed. This allows us to use a less cluttered notation when talking about patterns. Labels and relationship types will not be preceded by colons and only the property key will reside inside of the curly brackets. Multiple property keys can be used in the same pattern and even within the same curly brackets. To indicate that any relationship or node can be matched, parts of the pattern can be left without a label or relationship type. Using this slightly modified notation, some patterns are listed in figure 1.3.

1. `(Person)`
2. `(Person) <-[HAS_CREATOR]- (Comment {date})`
3. `(Person {name}) -[]-> ()`
4. `(Person) -[LIKES]->`

Figure 1.3: A list of example patterns

We say that a segment or a path matches a pattern if and only if it has the same labels and relationship types in the same order and also has the property keys of interest on the corresponding node or relationship. Looking at figure 1.4 as an example graph, pattern one would be matched by `(1)`, `(2)` and `(4)`. There are a few unique paths that would match pattern two, path `(2)<-[2]-(3)` being one of them. Pattern three could be matched by e.g. path `(4)-[3]->(1)` or `(2)-[5]->(5)`. Pattern four is matched by `(4)-[7]->`, which is a segment since it does not end in a node.

Consider pattern `(Comment {date})-[HAS_CREATOR]->(Person)`. This is pattern two from figure 1.3, but reversed. The same paths that matched that pattern will also match the reversed one. That does not make the patterns equivalent however, as the order in the pattern will indicate how matching paths should be sorted. This is further explained in section 2.1.4.

1.2.5 Schema

In general, there are no restrictions on what types of relationships, nodes and properties that can exist in a graph and how they relate. When you use a graph to model or describe a domain however, you would normally define some rules on your data domain. We call this collection of domain rules the schema of the domain. The schema shows exactly how nodes of different types can relate to each other, what types of relationships that can be attached and what property types each type of node or relationship can have. In figure 1.5 we can see the schema for the graph in figure 1.4. Note that a schema has a comparable syntax to a pattern. If the domain is very simple, as for the graph in figure 1.2, we can "draw" the schema with a pattern, in this case `(Person {name})-[:KNOWS {since}]->(Person {name})`.

1.2.6 Dense nodes and neighborhoods

A dense node is a node with many relationships. This is an extremely vague definition, but we can at least argue that one node can be more dense than another. Another way to describe a dense node would be that "it is costly to enumerate all of its relationships".

We define the k-hop neighborhood, or just k-neighborhood, of a node to be all nodes and relationships that can be reached with k-hops from that node.

We combine those two definitions to get dense k-neighborhoods. To illustrate, imagine a graph where every node has 1,000 neighbors. The 1-hop neighborhood of a node has

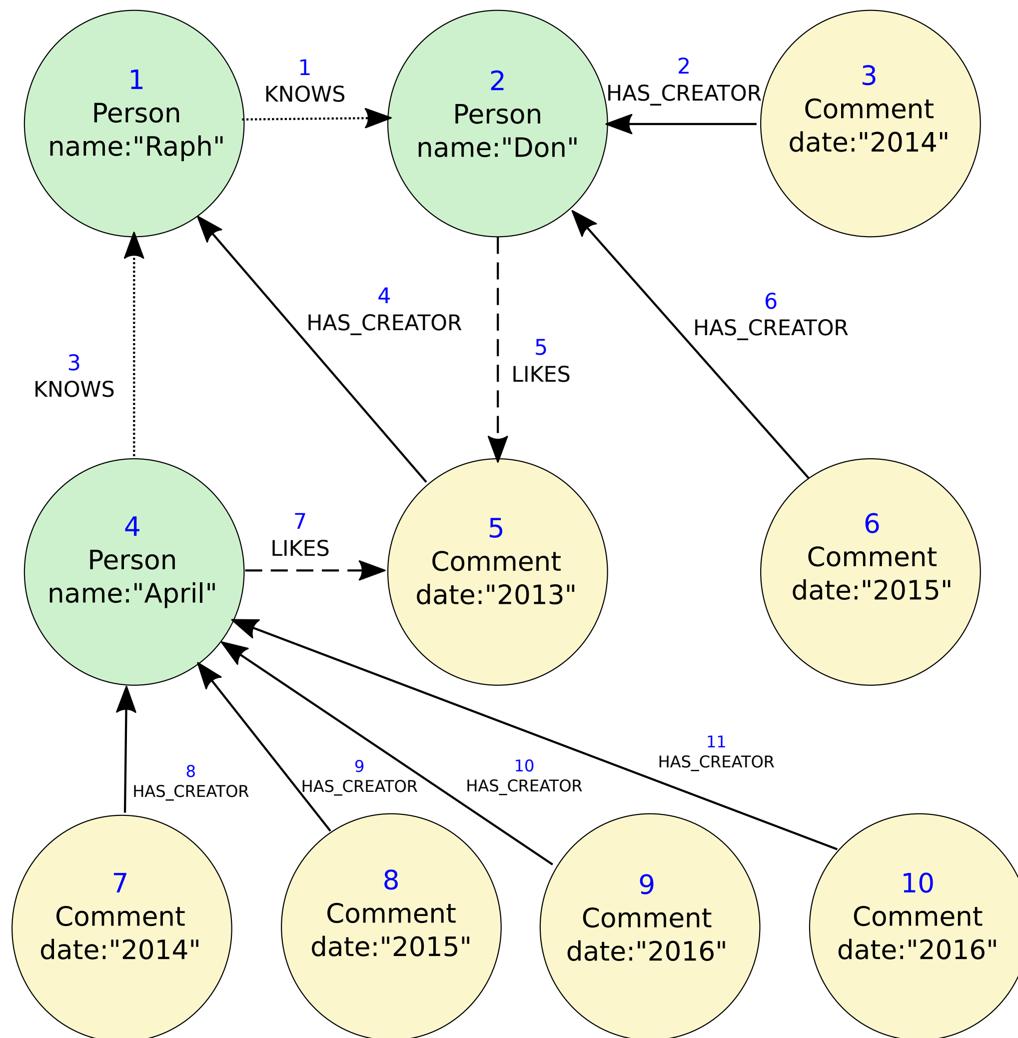


Figure 1.4: A small example graph illustrating a social network

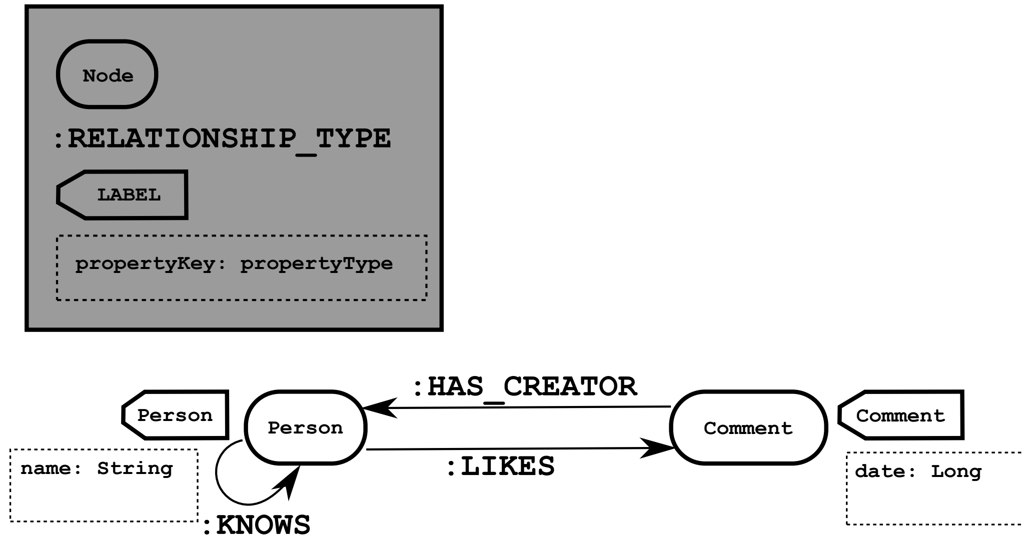


Figure 1.5: An example schema

size 1,000 (small), the 2-hop neighborhood has size 1,000,000 (large), the 3-hop neighborhood has size 1,000,000,000 (huge). This keeps growing exponentially. A dense 1-hop neighborhood is a dense node.

When we talk about density we will typically talk about the average density for all neighborhoods that match a pattern or the exact density of one of those neighborhoods. We say that a neighborhood has density n if there are n nodes in the neighborhood. In the same fashion we say that the average density for neighborhoods covered by some pattern is n if those neighborhoods have n nodes on average.

1.3 Graph database

In this section we will discuss what a database is and differentiate a graph database from a relational database. The relational database is presented in section 1.3.1 and the graph database, using neo4j as an example, in section 1.3.2. Finally in section 1.3.4 we briefly talk about the typical usefulness and trade-off with an index in a database.

1.3.1 Relational database

A database management system (DBMS), henceforth called a database, is a tool used to handle all storage and retrieval of persistent data. All software that uses some sort of persistent data has a database as back end. The first databases evolved from file systems in the late 1960’s. The programmers using these first systems needed to directly interact with the storage structure which complicated development [12].

In 1970 Ted Codd suggested that data should be stored in tables called relations with the storage structure abstracted away from the user behind a high level query language, [9]. This model is called the relational model or relational database management system (RDBMS). For a long time, the relational database has been the default alternative and it

is still by far the most popular choice, [1].

Relational databases are extremely efficient when storing data with fixed schemas of isolated data but when it comes to connected data they struggle, [19]. Connected data as described in [19]:

> Connected data is data whose interpretation and value requires us first to understand the ways in which its constituent elements are related.

Graph databases are an attempt at handling and make reason out of this type of data.

1.3.2 Neo4j

To make modeling and querying of connected data simpler and faster a graph database can be used. In a graph database the data is modeled as a graph which makes it easier to reason about the data in terms of dependencies and how data relate to other data.

It also has the advantage of local / index-free adjacency which means that if we are looking at node A and are also interested in the neighbors of A, w e do not need to search the entire database for those neighbors because we can reach them directly by traversing A's relationships. More about neighborhoods is presented in section 1.2.6.

For a graph database, no schema is needed, the data structure is described by the data itself. This creates a more flexible environment compared to relational databases. It removes the need for users to define the entire domain space up front and instead lets it emerge as the understanding of the domain grows [19].

Graph databases are built to manage data in an online transactional processing (OLTP) environment, more on OLTP in section 2.2.2. This is in contrast to graph compute engines that are used to make graph analysis or online analysis processing (OLAP), [19].

Neo4j [4] [5], is the current market leader in the graph database space [2]. It is an open source project driven by Neo Technology that implements a native labeled property graph model. Native meaning that the storage is designed and optimized to store and manage graphs. Included in neo4j is the development of Cypher, a graph database query language. All queries in this report are examples of Cypher.

1.3.3 Result rows and database queries

Queries are used to retrieve data from databases, whether it is a graph, relational or some other type of database. There are a lot of different query languages, like SQL (Structured Query Language) and Cypher. The result from a query can often be delivered as a table and we can then talk about the number of "result rows", the number of rows in the result table. We can also talk about the intermediate number of result rows in the middle of a query execution. When we have executed a query up to "this" point we have some number of result rows that could potentially be part of the final result. Later in the execution some of those rows can be proven to not match all of the requirements that the query has and are therefore filtered out. An example of this is described in section 1.4.1.

```
MATCH (p:Person)-[:KNOWS]-(friend:Person)
      <-[:COMMENT_HAS_CREATOR]-(comment:Comment)
WHERE id(p) = {1} AND comment.creationDate <= {2}
RETURN friend.id AS friendId,
       comment.id AS commentId,
       comment.creationDate AS creationDate
ORDER BY creationDate DESC
LIMIT 20
```

Query 1.1: The Holy Grail

1.3.4 Database indexes

Indexes are used in databases to speed up lookup time. The trade off for keeping an index is storage space and update overhead. The index needs to be stored somewhere and thus it takes up storage space and when we perform updates to the graph we need to keep the index up to date with the changes which adds an update overhead, [12] page 352.

1.4 Problem statement

When querying a database, performance in terms of response time is extremely important and improving execution time even when datasets get larger is a challenge for database developers. This work discusses a graph property that makes good performance particularly hard to achieve, namely graph density.

The purpose of the work is: Develop a path index to be used to achieve good performance in terms of response time, even in very dense graphs and evaluate the solution.

1.4.1 Density is hard

The idea with graph databases is to only touch data of interest through local / index-free adjacency. That is, the neighbors of a node can always be found without doing scans. However, even if only local data (data that is within the neighborhood of where the result will be found) is considered, it breaks down if the amount of local data is large, as for dense nodes / neighborhoods.

Let us use query 1.1, from here on called "The Holy Grail", as an example to highlight why this is difficult. We call this query "The Holy Grail" because it is one of our main goals to improve response time of this query. (We go into more detail about why this query is a suitable choice in section 1.4.2.) As we see in query 1.1 there are five different clauses. We briefly explain what they mean:

MATCH In this clause we describe what pattern to match paths or segments against. We bind some of the nodes or relationships that we find to variables, *p*, *friend* and *comment* in this example.

WHERE Here we can introduce some limitation or criteria on the matched paths or segments.

RETURN Here we say what is to be returned when the query is finished.

ORDER BY With this clause we define how the result should be ordered.

LIMIT With this clause we say that we are only interested in a limited number of result rows.

The Holy Grail touches `(Comment)-[COMMENT_HAS_CREATOR]->(Person)` which is a dense part of the LDBC dataset, presented in section 1.4.2 and 3.4.1. We examine how it is executed.

On average, every person has $177149 \cdot 2/9987 \approx 35$ friends, using statistics for SF001 in table C.1 in the Appendix. Every friend has created $2015590/9987 \approx 200$ comments on average. Using these metrics execution will look like the numbered list below. We write the number of result rows in each step within parenthesis and the multiplicative cost of expanding relationships within square brackets.

1. Seek some assumed index for `(Person id:{1})`. (1)
2. Expand all `[KNOWS]` [x35]
3. Filter out nodes that are not `(Person)` (35)
4. Expand all `[COMMENT_HAS_CREATOR]` [X200]
5. Filter out nodes that are not `(Comment)` (7000)
6. Project `{comment.creationDate}`
7. Filter out `(Comment)` that does not fulfill the range predicate (<7000)
8. Sort the result
9. For results within limit, project `{friend.id}` and `{comment.id}` (20)

The result set we end up with is only 20 rows, but we still need to examine 7000 `(Comment)`s. This is why density makes execution unnecessary hard and slow.

1.4.2 About LDBC

The Linked Data Benchmark Council (LDBC) is a non-profit organization with members from companies, non-profit organizations and individual members that are all involved in the graph computation sphere. They are "dedicated to establishing benchmarks, benchmark practices and benchmark results for graph data management software" [3] and in this work they have put together the LDBC Social Network Benchmark (SNB). In short, SNB is a collection of queries combined with a generated dataset aimed to benchmark the execution of particularly difficult tasks. More about the SNB can be found in [11].

Query 1.1 is a simplification of Query 2 from the LDBC Social Network Benchmark [6]. This query illustrates the density problem well and is also a good example of a real world use case.

```
MATCH (x)-[:KNOWS]->(y)-[:CREATED]->(z)
RETURN ID(x), ID(y), ID(z)
```

Query 1.2: A cypher query that k-path index handles well

1.4.3 Our statement

We state that by indexing the paths that make up the dense neighborhood we can improve performance in terms of response time. The goal of this work is to create such a path index and evaluate the reached solution. We expand on how we do this in chapter 2.

We aim to answer the following questions:

- How can we create a path index?
- Can we improve response time by indexing paths?
- Is a path index a good tradeoff to improve response time in a graph database?

1.5 Previous work

In this section we present some previous work done in the field of database indexes and specifically path indexing.

1.5.1 K-path indexing

The k-path index developed by Jonathan M. Sumrall in his Master's thesis [22] is an index on all paths of length equal to or smaller than k. A path is described by the pattern of relationship types within the path. For example the path `(Person)-[KNOWS]->(Person)-[CREATED]->(Comment)` has the pattern `<KNOWS,CREATED>`.

The keys stored in the index is a "path identifier" that deterministically describes the path pattern and the graph node ids in the path. It allows for wild card search on, for example "find all paths with this pattern that starts in graph node with id i". The index shows great performance for queries like the one presented in query 1.2.

To store keys, the k-path index uses a variation of the B+ tree. Instead of storing keys mapped to values it only stores keys which by themselves make up the entire indexed path.

Our work is largely based on and inspired by this indexing technique and a lot of similarities can be found between them. Most of all when it comes to syntax, implementation and the choice to use a B+ tree. The major difference between the k-path index and the path index presented in this report is the use of node and/or relationship properties and what type of queries it aims to improve. The k-path index focuses on graph global queries, e.g. query 1.2, while the path index introduced in this report focuses on graph local queries where you have a specified set of or single start node(s).

1.5.2 Database cracking and adaptive merging

Database cracking was first introduced in [15] and it is built around the observation that not all data is of interest. There is skewed "interest rate" between data in a database. The idea is to let the data store adapt to the executed queries. Every query is interpreted as an advice to crack the database store into smaller pieces. Eventually, data that is frequently queried together will be stored in a continuous sequence and indexed. Data that has never been queried will be left out and not rearranged or indexed until it becomes interesting.

Adaptive merging, presented in [13], builds on the same idea as database cracking, that an index should be incrementally optimized for the data that is queried. While database cracking has a low cost for initialization, it takes a long time for the index to converge to a fully optimized state. Adaptive merging tackles this problem by storing the data in a partitioned B-tree and merging the queried ranges in every partition together on query execution time.

> The main difference between adaptive merging and database cracking is the rate of adaptation, i.e., the number of times each record is moved before it is in its final location. [13]

A hybrid approach is presented in [16]. It seeks to combine the low initialization cost of database cracking with the fast adaptation rate of adaptive merging.

These techniques are interesting and could possibly be used for a path index, especially if the interest rate between different paths is skewed.

1.5.3 B+ trees

A thorough introduction to different variations of the B tree is given in [10]. In particular the B+ tree, which is the data structure used in this project, is analyzed. In short a B+ tree is a tree structure with more than one key per tree node that is guaranteed to be balanced. We go into more detail in section 2.1.3, 2.1.4 and 2.1.5. B-trees and B+ trees are also covered by [12].

1.5.4 Bitmap index

Assume that you have records numbered from $1 - n$. A bitmap index over field F is a collection of bit-vectors of length n, one vector for each possible value associated with F. The easiest example would be a boolean field that only takes values of true or false. Then only two bit-vectors are needed. If record i has the value represented by some vector, then that vector will have the bit number i set to 1, otherwise it would be 0. In this way, all values for the targeted field for every record can be stored in a compact way. The basics of how a bitmap index works is covered in [12].

However, it is not certain that any of the records will share values for the targeted field. If in fact the field is unique for every record then n different bit-vectors are needed and storage for the index is n^2.

The bitmap index was first introduced in [24] together with a number of encoding schemes and has since then been widely covered in multiple articles, see [25] for a list. An analysis on optimal time and/or space bitmap indexes is given in [8].

As described in [20] the bitmap is best suited for DSS (Decision Support Systems) and not for OLTP environments. Thus this technique is not used in this project.

1.5.5 R-Tree

The R-tree (region tree) was presented in [14] and is used to index spatial data, that is, data that represent some spatial object in any dimension. It is fully dynamic and supports intermixed inserts, deletes and reads. It always stay completely balanced using the same strategy of merging and splitting nodes as the B-tree. Instead of using splitting values to guide traversal down the tree, like what the B-tree does, the R-tree uses spatial bounding boxes as intermediate guidance points in the internal nodes. When searching the tree for some spatial object with bounding box S you find all index records in the current node of the tree that completely overlaps the S. Hopefully there is only one such index record, but there might be many and in the worst case all of the records in the current node. In the normal case such scenarios can be avoided with a good insertion algorithm that makes it easy to exclude regions that are not interesting to search, but it is not guaranteed. R-trees are also covered by [12].

The idea of using bounding boxes as guidance points is interesting. If we want a path index sorted and queried on more than one property this could correspond to indexing in more than one dimension, ergo spatial indexing. So instead of first sorting on the first property and then on the second property we could sort on the first along one dimension and along the second in another. Of course, as the property values do not likely have a natural spacial equivalent, creating bounding boxes may or may not make any sense at all. For example if we index a path and want to sort on some creation date of a node and weight of a relationship, what does it mean to have a bounding box of (5 sept − 8 sept)X(1 − 5) (date range X weight range)?

This idea is not further investigated in this project.

1.5.6 Composite indexes

Assume that you have a relational database with a person table with columns: lastName, firstName and socialSecurityNumber as the primary key. If we want to search this table for every person with last name x or last name x and first name y we need to do a complete table scan.

A composite index, briefly described in [17], is an index on multiple columns in one table, for example on columns lastName and firstName (in that order). This index will be ordered firstly on last name and secondly on first name. This makes it possible to search for every person with last name x or last name x and first name y much quicker compared to a complete table scan.

Indexing on multiple properties of a path and multiple properties of a table row is quite similar and we will examine this further in section 2.2.3.

Chapter 2
Approach

In this chapter we will present our approach to at least partially solve the problem presented in section 1.4. We begin by presenting a novel path index design in section 2.1. We then analytically evaluate this design in section 2.2. Finally, in section 2.3, we explain how we implement the presented path index design.

2.1 The Shortcut Index

We present the "Shortcut index", a suggestion for how a path index could be designed to solve at least part of the density problem stated in section 1.4. We start by giving a detailed view of the design in section 2.1.2 and describe the core data structure, the B+ tree in 2.1.4. We then continue to motivate the choice of using a B+ tree in section 2.1.5.

2.1.1 Defining a Shortcut index

The objective of the index is to store instances of paths that match a specified pattern in some sorted order. The terminology when initiating an index defined by a pattern is that you "create an index on pattern x" or simply "index pattern x". It means that every path or substructure in the graph that matches the pattern should be indexed. Patterns are defined in section 1.2.4. Paths will be divided into keys and values as explained in section 2.1.2 and stored in the index.

The idea of creating an index on a specified pattern was inspired by the "path signatures" introduced by Max Sumrall in [22].

Note the similarities between indexing a complete path and a composite index as described in section 1.5.6 and 2.2.3.

Matching path	Key	Value	Combined
`(1)<-[4]-(5)`	<1,"2013">	<4,5>	<1,"2013" : 4,5>
`(2)<-[2]-(3)`	<2,"2014">	<2,3>	<2,"2014" : 2,3>
`(2)<-[6]-(6)`	<2,"2015">	<6,6>	<2,"2015" : 6,6>
`(4)<-[8]-(7)`	<4,"2014">	<8,7>	<4,"2014" : 8,7>
`(4)<-[9]-(8)`	<4,"2015">	<9,8>	<4,"2015" : 9,8>
`(4)<-[10]-(9)`	<4,"2016">	<10,9>	<4,"2016" : 10,9>
`(4)<-[11]-(10)`	<4,"2016">	<11,10>	<4,"2016" : 11,10>

Table 2.1: Table illustrating how keys and values are formed from paths

2.1.2 Keys and values

We will here talk about two different types of "value". *Property value* is a value associated with a *property key* in a graph node or a relationship. This is a part of the data in the graph database. The index will have *index keys* and *index values*. These are not to be confused for *property keys* or *property values*. An *index key* will contain *property value(s)*, as we will see below.

Keys and values are tuples. Given a path that matches pattern *P*, we represent the path with an *index key* and an *index value*. Together they will make up the entire path. The key will be the id of the first node in the path and the *property values* associated with the *property keys* included in *P*, <firstNodeId, propValue1, propValue2,...>. This allows *index keys* to be sorted in lexicographic order which will turn out to be an important feature. Lexicographic order means first ordered by comparing the first part, then by the second part and so on.

The *index value* will be ids of the remaining relationships and nodes from the matching path, <relId, nodeId,...>. Note that it does not hold the id of the first node as it is already included in the *index key*.

When writing *index keys* and *index values* in combination we separate them with a colon. Like this: <firstNodeId, propValue... : relId, nodeId,...>, where the first part is the key and the second part is the value (relationship id and node id) mapped by the key.

Given pattern two from figure 1.3 and the graph from figure 1.4 the mapping would look like in table 2.1.

2.1.3 What is a B+ tree?

The B+ tree, that we introduced briefly in section 1.5.3, is the core data structure used by the Shortcut index. In this section we explain what a B+ tree is and how it works, using figure 2.2a as an example.

Note that a node in the context of a binary search tree or a B+ tree is not the same as a node in a graph. If the context does not make it clear what is meant, "graph node" or "tree node" will be used to differentiate between them.

The choice of using a B+ tree as the core data structure was influenced by the design of the k-path index introduced in [22].

Binary search tree

Even if we assume that the reader is familiar with the ordinary binary search tree we want to make a short recap of how they work to make a nice bridge towards the B+ tree. In figure 2.1 we have an example of a binary search tree. The tree has nodes that hold keys. We call the nodes internal nodes and leaf nodes. A node is a leaf node if it does not have any children, otherwise it is an internal node. The top node is called the root and in the example the root holds number 5. Every key lower than 5 is located in the left subtree and every key greater than or equal to 5 is in the right subtree.This is the mechanism that keeps the tree ordered. Note that we do not have unique keys. This is not usually the case in binary search trees but it will make sense when we look at how we use the B+ tree. There are three different operations that we want to use on the tree: find, insert and delete.

Find If we wanted to find number 6, we would start at the root, find out that $6 > 5$ and traverse down to the right subtree. Then continue this traversal until we reach number 6 in the very left leaf of this subtree. This was a successful find, a hit. A find operation on number 22 would end up at number 21 in the very right leaf. As 22 is larger than 21 we would want to traverse down the right subtree, but it does not exist so the find would end in a miss.

Insert Insertion in a binary search tree is fairly straightforward. Execute a find for the key you want to insert. If it results in a hit the new node is inserted where the hit occurred. The left subtree is moved to the new node and we let the hit node together with its right subtree become the right subtree to the new node. We "squeeze" the new node in and push the right subtree down. If it results in a miss, compare the insert key to the key in the leaf where the find terminated and insert it as a new node to the left or the right depending on if it is less than or greater than the key in the node. If we were to insert number 8 we would traverse down to number 7, see that there is no subtree to the right and therefore create a new node for number 8 and insert it as the right child to number 7.

Delete We will not discuss the deletion algorithm here, but there is a rather straightforward way of doing deletions and the algorithm can be found in most books about data structures that cover binary search trees.

There are multiple different variants of the binary search tree, like the self-balancing trees: AVL tree and red-black tree. The only variant we will discuss in this report is the B+ tree.

How does the B+ tree work

In figure 2.2a we see an example of a B+ tree. We will now briefly explain how a B+ tree works but for a detailed description we refer to [10].

We first list the key features of the B+ tree. Then we continue with how the "find", "insert" and "delete" operations work, comparing and contrasting this with how they work in a binary search tree.

Key features that differentiate a B+ tree from a binary search tree are as follows:

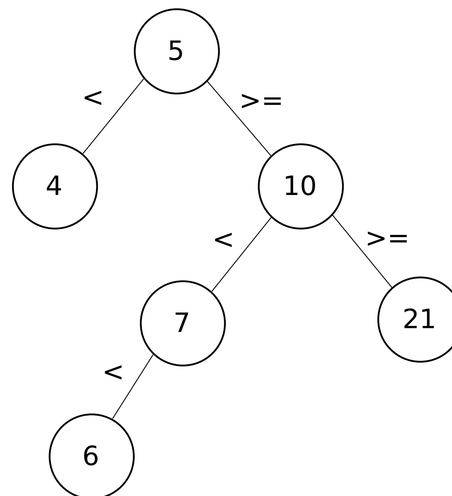


Figure 2.1: A binary search tree with unique keys

Multiple keys per node The order d of a B+ tree decides how many keys and children each node can have. If the order is d, internal nodes contain between d and $2d$ keys and between $d + 1$ and $2d + 1$ children, always one more pointer than the number of keys. Leaf nodes contain between d and $2d$ keys. The root node is an exception, it can contain between 0 (if the B+ tree is empty) and $2d$ keys. The tree in the picture has order 1 which is not the usual case. It is not unusual to see B+ trees of order 50 or more. We only use order 1 in these examples for simplicity. The order affects the height of the tree. A higher order means more keys per node which means a lower height. See "Time complexity analysis" in section 2.1.5 to see how this affects performance.

Sorted The keys are always stored in some sorted order. In this example we use ascending order, that is, a key is always greater than or equal to the key to the left and less than or equal the key to the right, much like in a binary search tree. This holds for leaves as well as internal nodes.

Linked leaves Every leaf holds a pointer to its right and left sibling which makes it possible to do sequential scans along the leaves in any direction without traversing through the internal nodes to find the next leaf. To make this work, all keys in the tree need to be located among the leaves, even the ones that are stored higher up in the tree. Note that this is different compared to the binary search tree where keys were "only" stored where they fitted in the tree. In the B+ tree a key can be stored both in an internal node and in a leaf.

Balanced A B+ tree is always completely balanced, all leaves are on the same depth from root. This is achieved by letting the tree "grow" upwards instead of downwards.

All of those features are key ingredients as to why the B+ tree was chosen for this project and we expand on this in section 2.1.5.

Below is a short explanation of how the different operations are performed in a B+ tree.

Find You perform a find operation on a B+ tree to find the location of given search key, S. Starting at the root node, n, you compare S with the first key in n, let us call it K. If $S <= K$ follow the pointer to the left of K, else let K be next key in n. Repeat until you find a pointer to follow or you reach the last key in n, in that case follow the pointer to the right of K. Repeat until you reach a leaf node. Scan the leaf until you find an exact match or you find a key that is greater than your search key. If the last key in the leaf is still less than your search key, continue scanning in the right sibling. In figure 2.2b we illustrate a successful find on key 25. Note that even if you find an exact match to your search key among the internal nodes you still need to traverse all the way down to the leaf to be certain that the key actually exists in the tree. The only purpose of the keys among the internal nodes is to split the key interval into sub-intervals and thus guide the seek. Note also that a linear search is performed in every node. It would of course be beneficial to use binary search instead and this is suggested as future work.

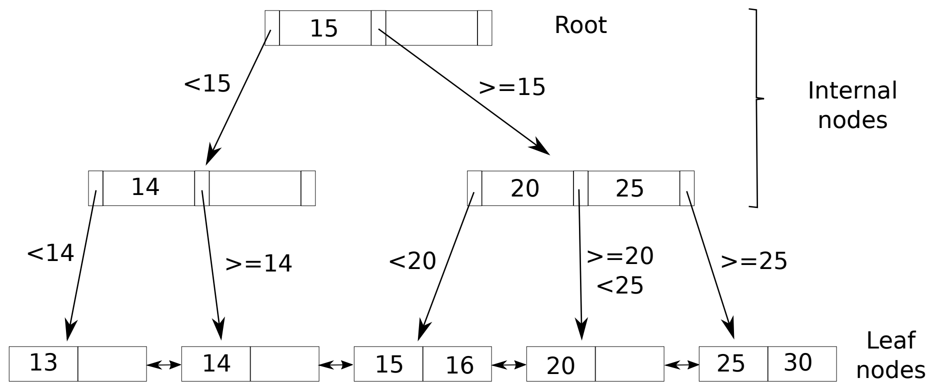
Insert To insert a key you first perform a find on the key to be inserted. Then insert the key on the position where the find terminated. If this leaf is already full an overflow occurs and the leaf has to be split. In figure 2.3 a split occurs after inserting 26. First we initiate a find for 26 which terminates in a miss in the second position in the right-most leaf. This is where the key should be inserted. Because the leaf is full we get an overflow and we need to split the node in two, dividing the keys between them. The left-most key in the new leaf is sent upwards in the tree together with a pointer to the new leaf to be inserted in the parent node. This again results in an overflow and a split. The same procedure of splitting is repeated with a slight difference, the middle key that is sent upwards does not need to be kept in the internal node, which was the case in the leaf node. Remember that all keys need to exist among the leaves. The tree only grows in height when a split occurs in the root, thus the tree grows upwards.

Delete The delete operation is done in a similar way to insert but instead of overflow you can get an underflow when you leave a node with less than d keys. If possible we then collect the keys from a neighboring node and divide the keys evenly, this only works if there are at least $2d$ keys in total among the two nodes. If not, a merge or concatenation occurs. That is, we move all of the keys to one node and remove the other. A more detailed description is given in [10].

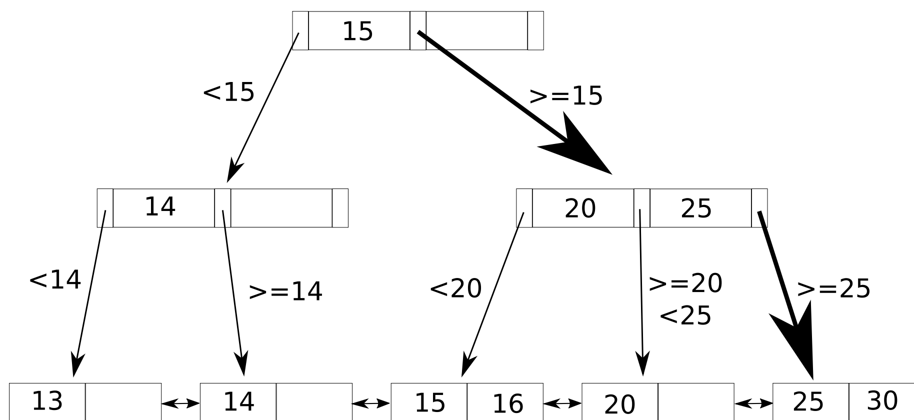
2.1.4 How do we use the B+ tree

We use a B+ tree to store *index keys* and the mapped *index values*. We explain here how we use the B+ tree.

As explained in section 2.1.2 a path matching a pattern, in this case of length one, is divided into a key and a value like this: key: <firstNodeId, propValue>, value: <relId, secondNodeId> and in combination: <firstNodeId, propValue : relId, secondNodeId>. Recall that we use lexicographic ordering for the keys and that values do not affect ordering at all. We use the B+ tree not only to store keys, but also to store the mapping from key to

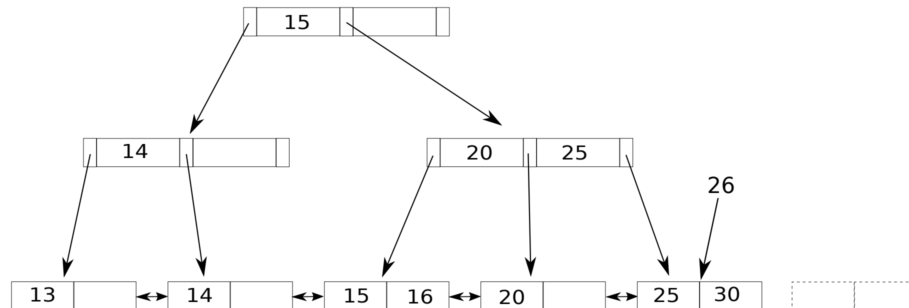


(a) Illustrating the different parts. Only keys are shown.

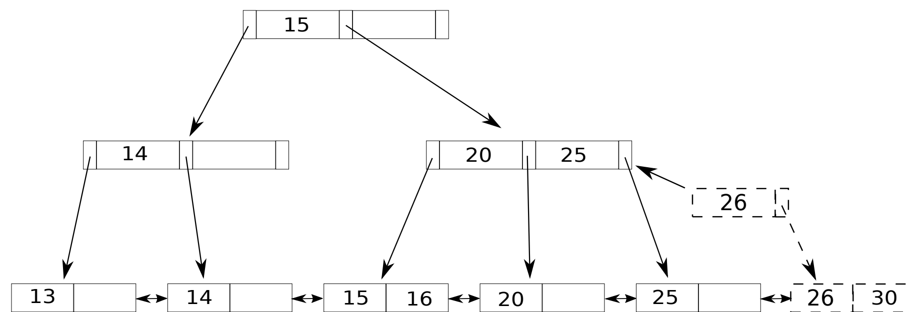


(b) Search for key 25. Chosen path is bold.

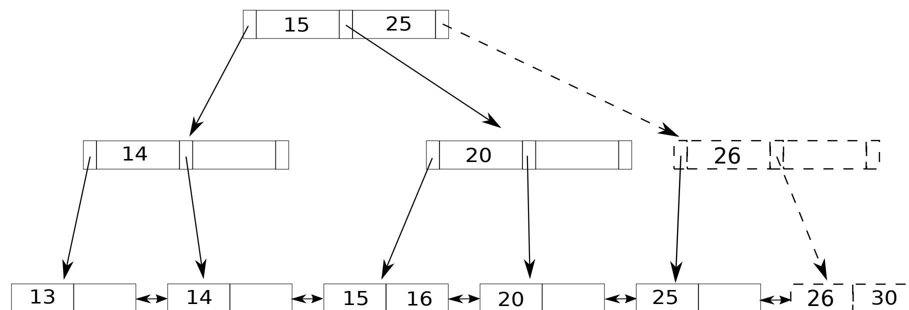
Figure 2.2: A simple B+ tree



(a) Key 26 should be inserted in the right most leaf. There is an overflow so a new leaf is created.



(b) The keys in the previously right most leaf are split and half is moved to the new leaf together with 26. The middle key, 26, is sent to the parent together with a pointer to the new leaf. Note that 26 is still kept in a leaf node.



(c) Inserting 26 and a new pointer results in a split in the internal node. The keys and pointers are divided and the middle key is inserted in the parent without causing a split. Note that the middle key is not kept in the internal node.

Figure 2.3: Step by step insertion of 26.

value. As all keys are stored among the leaves there is no point in storing the values in the internal nodes. Therefore they are only stored in the leaves.

The reason we let keys exist in duplicates in the tree is because they can still map to different values. However the combination of key and value need to be unique, we do not want to keep multiple instances of the same path in the index.

A B+ tree with keys and values from table 2.1 is shown in figure 2.4a. This is the complete Shortcut index for pattern `(Person) <-[HAS_CREATOR]- (Comment {date})` on the graph in figure 1.4. If we were to create an index on the reversed pattern, `(Comment {date}) -[HAS_CREATOR]-> (Person)` instead, the stored paths would be the same but the ordering would be different, see figure 2.4b. For example, the first key, value pair in figure 2.4a <1,2013:4,5> represents the same path as the second key, value pair in figure 2.4b <5,2013:4,1>. The only difference is how the node ids are ordered among the *index key* and the *index value*.

2.1.5 Why do we use a B+ tree?

First a quick recap. We want to index paths. We do this by storing the id of the first node as the first part of the key and the ids of the remaining relationships and nodes as value. We also store some property value(s) in the key which allows for further sorting. The B+ tree has some features that makes it suitable for this index.

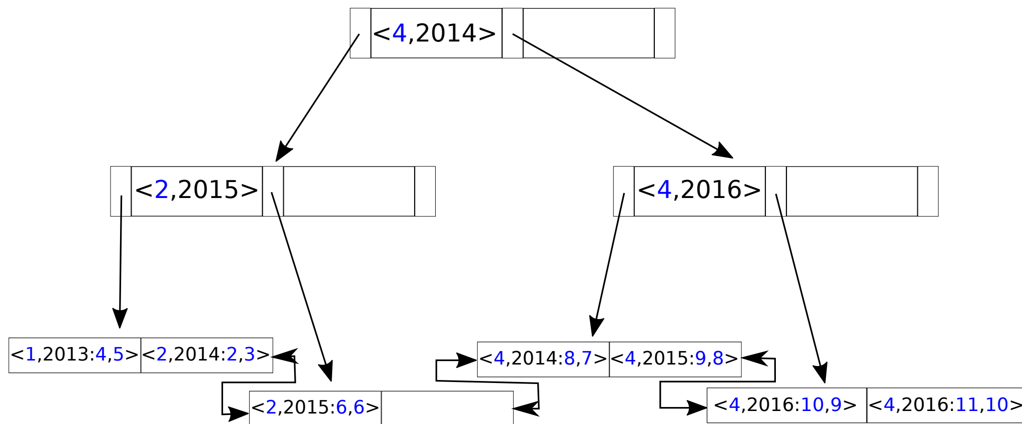
Neighborhood locality The B+ tree stores the key in sorted order which means that all paths starting in the same node will be located next to each other, both logically in the B+ tree and physically on disk. We call this feature *neighborhood locality*. Therefore we can read an entire neighborhood of paths in one or a few blocks depending on if it is spread across multiple leaf nodes, like paths starting in node 4 in figure 2.4a.

Range locality The keys can also contain one or more property values. Because of this, the keys within a neighborhood (starting with the same node id) can be sorted on some feature(s) of interest. For example creation date of the comment in the path, as in figure 2.4a. This makes it easy to read all paths that have this property value within some given range. You just need to find the first and the last path within the range and every path in between should also fit the range. We call this feature *range locality*. Note that range locality is something that exists within a neighborhood. We could achieve range locality on the entire graph by sorting first on property value and then on node id. That however will break neighborhood locality. Depending on the use case this can still be useful though.

Once we have read the first key in a range it is not much more expensive to read the key following immediately after, as long as we do not have to jump to the next leaf (which could of course be the case). As there are at least d (the order of the B+ tree) keys per leaf we only need to jump to the next leaf once for every d keys we read in sequence. Reading a range of k keys means on average that we need to read $\frac{k}{d}$ different leaves. This is not including the internal nodes that we need to read while traversing down the tree from the root.

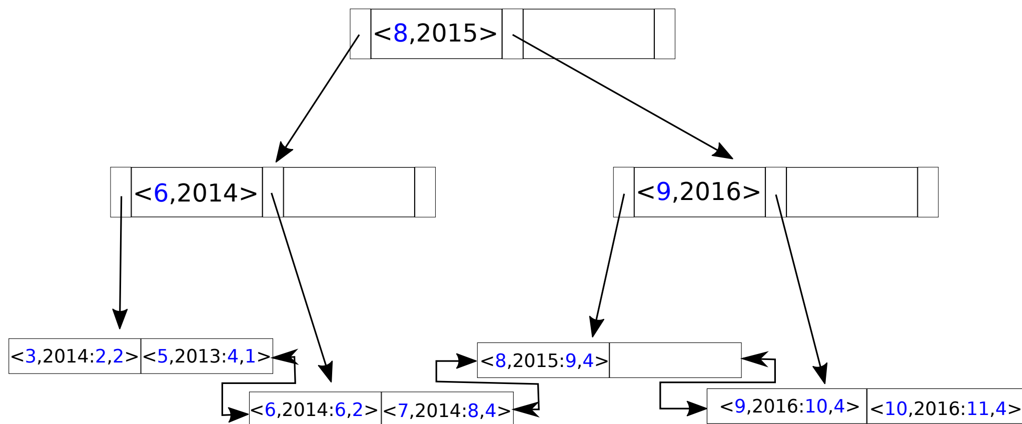
Memory utilization Memory access is fast. Disk access is slow. We want to minimize the number of disk accesses. Therefore we want to keep the most often accessed

Index on pattern: (Person)<-[HAS_CREATOR]-(Comment {date})



(a) Note that keys and values are the same as in table 2.1

Index on pattern: (Comment {date})-[HAS_CREATOR]->(Person)



(b) Shorcut index on the reversed pattern compared to (a)

Figure 2.4: Shortcut indexes on graph from figure 1.4

data in memory. The tree structure makes it easy to predict what data that is. It is of course the top internal nodes as all access to nodes further down in the tree need to pass through them. So by keeping the top internal nodes in memory you are sure to utilize it as good as possible.

Low operation overhead

Let T be a B+ tree of order d with n keys in total and k be the size in bytes of a key. The find operation must then visit at most $\log_d(n)$ tree nodes before reaching the leaves.

The cost of a find operation is dominated by the number of disk accesses needed and each node in the B+ tree will occupy one block of data or "page" and can be read into memory with one access to disk. This is covered in more detail in section 2.3.2 and 2.3.3.

If all nodes in the tree reside only on disk, which is the worst case, the cost grows proportionally to $\log_d(n)$. This would suggest that a larger d is always better. This is true to some extent, but the system restricts how large blocks of data that can be read from disk with only one disk access. So if $d \cdot k$ is larger than this block size the number of disk accesses needed is suddenly doubled. The optimal order will therefore be different on different machines.

The worst case when doing an insert is when a split occurs at every level of the tree. In this case twice as many nodes are touched compared to when only doing a find. The cost for an insert thus also grows proportionally to $\log_d(n)$. The same holds for delete which also obeys the time complexity $\log_d(n)$.

To summarize, the B+ tree has a low overhead for all relevant operations.

There are of course other data structures that could be considered; for example, a hash structure, that has constant overhead on all operations. This is obviously superior to the B+ tree, but a hash set does not keep the data in sorted order, which is a complete necessity for the use cases we are looking at. There are hash structures like the "linked hash map" that can be used to keep the insertion order of the entries but that is not good enough to be considered for our use case.

A bitmap could be considered. They are however better fitted for DSS (Decision Support Systems) and not so well for OLTP environments because of the high overhead on write, referring to [20]. One could argue that bitmaps are only useful when the number of unique values is limited and considerably less than the number of records but this is not entirely true as shown by [20].

2.2 Usefulness

In this part we discuss the usefulness of the Shortcut index in an analytic way without presenting any results. Section 2.2.1 considers the immediate question, does it solve the problem? We also reason about what demands and limitations an OLTP environment puts on the index in section 2.2.2. In section 2.2.3 we reason about how the Shortcut index could be used as a composite index.

2.2.1 Solve the problem

We attempt to solve the density problem explained in 1.4.1 with a path index, sorted by some property value. This allows us to only read the data in the range that we are actually interested in and ignore the rest of the neighborhood.

To illustrate we use "The Holy Grail", query 1.1, as an example and revisit the execution described in section 1.4.1, now assuming we have a Shortcut index on `(Person)<-[COMMENT_HAS_CREATOR]-(Comment {creationDate})`. The query will initially be executed exactly the same as if we did not have the index, but when we reach the indexed pattern we will switch from the database to instead get data from the index. This is how the execution will look step by step.

1. Seek some assumed index for `(Person id:{1})`. (1)
2. Expand all `[KNOWS]` [x35]
3. Filter out nodes that are not `(Person)` (35)
4. **Instead of expanding all `[COMMENT_HAS_CREATOR]` we perform a find and scan the index (the B+ tree) for all entries within the range and limit reads to 20** (700)
5. Sort the result
6. For results within limit, project `{friend.id}` and `{comment.id}` (20)

As seen, we reduce the number of "touched" `(Comment)` with at least one order of magnitude which should improve performance, at least in terms of response time. This is not the only benefit. As we saw in section 2.1.5 it is also less effort to read each Comment because of *neighborhood locality* and *range locality*.

2.2.2 Indexing in an OLTP environment

The number of index entries that needs to be updated when the graph is changed grows exponentially with the length of the indexed pattern. Let us illustrate this with an example.

Assume we have a graph with this schema: `(A)-[TO_B]->(B)-[TO_C]->(C)`, and the fan out on every level is f. There are f number of `(B)` connected to every `(A)` and the same for `(C)` and `(B)`. To the left in figure 2.5 you can see an example graph with $f = 4$. Let us now assume we create a Shortcut index on pattern `(A)-[TO_B]->(B)-[TO_C]->(C)` and that we have some property we can sort on, which one does not matter in this example. Every `[TO_B]` relationship is part of 4 indexed paths, so removing one of them means removing 4 entries from the index. As all of those paths start in the same node they will be located adjacent to each other in the index, so the deletion could be optimized to only use one find operation. Let us now imagine that every `(B)` node is connected to 4 `(A)` nodes as well, as shown to the right. The `[TO_C]` relationships are also part of 4 different indexed paths, however all of the paths have different start nodes which means 4 completely separated index entries will have to be removed. This will require 4 find operations.

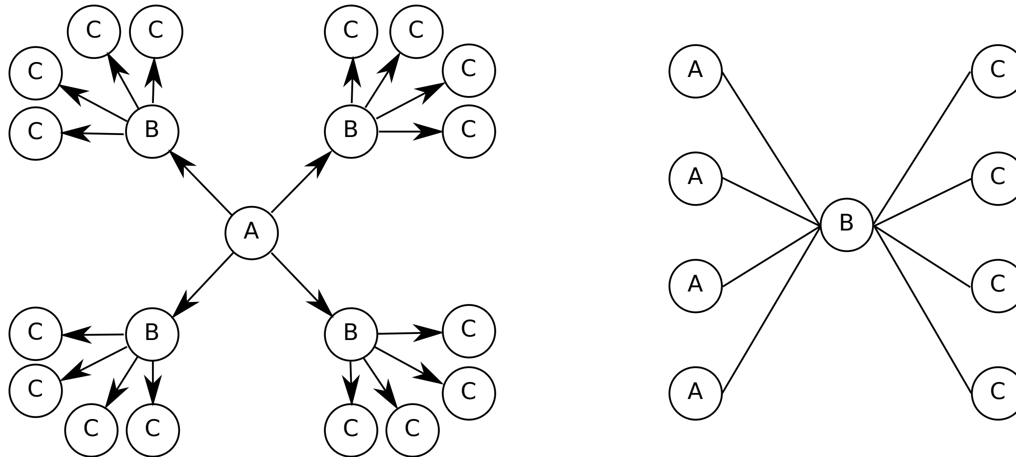


Figure 2.5: A graph with fan out 4.

Now, we assume that we have a graph with fan out f on every node and we create a Shortcut index on a pattern with length l, then every relationship is part of at least f^{l-1} indexed paths. Removing any of those relationships then of course means updating f^{l-1} index entries. Adding a relationship that fits in the pattern likewise mean adding f^{l-1} index entries.

In an online transaction processing (OLTP) environment, where nodes and relationships are constantly added and removed, such overhead for updates is unacceptable and the Shortcut index should not be used for patterns longer than one.

The index size also grows by a factor of f^{l-1} which is another reason to not index patterns longer than one.

2.2.3 Using path index as composite index

We recall the composite index used in relational databases as described in section 1.5.6. We created an index on multiple columns of a single table. This is very similar to creating an index on a pattern of length 0, that is a single node. To bridge over from the person table example, let us assume that we have `(Person)` nodes with properties lastName and firstName. We can then create a Shortcut index on pattern `(Person {lastName, firstName})`. If we instead of storing the node id as first part of the key, we store the property value of lastName and firstName, the index would be sorted on lastName and firstName. We could then store the node id as the value instead. One entry in the index would then be < "Sanzio", "Raffaello" : 1 >. We then have an index very similar to a composite index on the "person table". Of course we do not have a table as we are working with a graph database, but rather a composite index on nodes with label Person.

2.3 Implementation

In this section we describe how the Shortcut index has been implemented. In section 2.3.1 we describe what parts of the complete Shortcut index design that have been implemented.

We then go into detail about the store format in section 2.3.2. Finally we describe how we let the page cache implemented by neo4j handle all file access in section 2.3.3.

The full implementation can be found in a public repository on GitHub, see [18].

2.3.1 Partial implementation

We implement the Shortcut index with three restrictions for simplicity.

1. For reasons presented in section 2.2.2, our implementation can only index paths of length one.
2. The indexed pattern must contain one and only one property key and the property value must be of type *long*.
3. We do not implement the delete operation.

We have restriction two because a general purpose implementation would take longer to implement and would not add any knowledge about how efficient this approach is.

We have restriction three since the purpose of this work is to examine the potential performance enhancement for read queries that the Shortcut index provides. To benchmark this we do not need the delete operation and it is therefore not implemented. An investigation of the overhead for deletes is of course of great interest, however outside the scope of this project. We present suggestions for further work in section 5.7.

2.3.2 Store layout

We show an illustration of the store format in figure 2.6. In 2.6a we see a simple example of an index file containing internal and leaf nodes together with child pointers.

Page

Virtual memory is a way for an operating system to simulate that it has more memory than it actually has. It makes it possible for a program to access data as if it were loaded into memory, even if it is not. A page is a fixed size block of virtual memory. Virtual memory and paging are discussed further in [7]. This is how the page concept is described and used by neo4j in [23]:

> A "page" is a space that can fit a quantity of data, and is part of a larger whole. This larger whole can either be a file, or the memory allocated for the page cache. We refer to these two types of pages as "file pages" and "cache pages" respectively. Pages are the unit of what data is popular or not, and the unit of moving data into memory, and out to storage. When a cache page is holding the contents of a file page, the two are said to be "bound" to one another.

We think about a page as fixed sized blocks in a file that can be loaded into memory. Each page has a unique id and this id is what we use as a pointer between nodes in the B+ tree. The concept of pages is of huge interest to us as accessing a page can be expensive, if the page is not loaded into memory. We discuss how we handle pages in section 2.3.3.

One node per page

The first thing to notice is that the index files are divided into pages with unique ids ranging from 0. Every node occupies one page. The internal nodes use the page id of its children as pointers to make traversal down the tree possible.

There is no ordering among the nodes, when a new node is created (when a split occurs) the index simply allocates a new page at the end of the file.

In figure 2.6b and 2.6c we see the store layout of an internal and a leaf node. These are the components:

Type A simple one byte flag where 1 means the page holds a leaf node and 0 means internal node.

Key count A big endian integer, the number of keys the node currently has.

Right and left sibling The page id of the node's right/left sibling, stored as a big endian long, or -1 if no right/left sibling exists.

Key A key is two big endian longs. The first one is the id of the first graph node in the path and the second one is the property value of the property included in the indexed pattern.

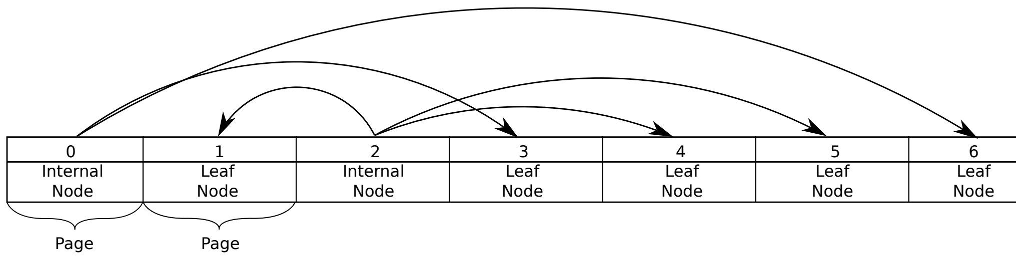
Child This is only present in internal nodes and is the page id of the child, a big endian long. If the internal node holds k keys then it also holds $k + 1$ child pointers.

Value This is only present in leaf nodes and is two big endian longs. The first long is the relationship id of the path and the second is the last graph node id in the path. If a leaf holds k keys it also holds k values.

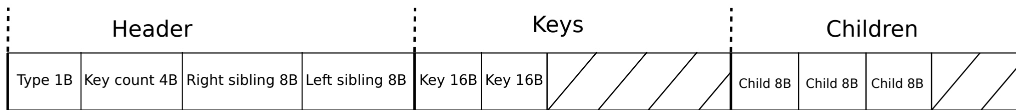
Depending on the fixed page size a node can hold a fixed number of keys, values and children blocks. This number will be different for internal and leaf nodes as value blocks are twice as large as child blocks. That means internal and leaf nodes will have different orders in this implementation.

As seen in the figure, value and child blocks are kept at an offset from the key blocks. This is so that we are not forced to move all values or all children every time a key is added. When a page is filled up, the empty space between the keys and values / children is 0. It is likely that there is some unused space left after the last value or child block. This is at most $16 + 16 - 1 = 31$ bytes, just short of one key and one value block.

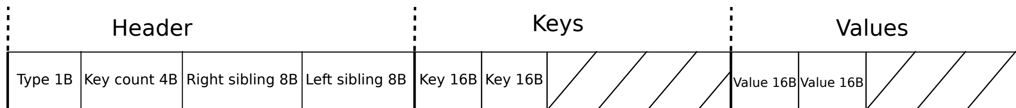
One could imagine a store format that more closely mapped to how we think about nodes in a B+ tree, with the child blocks being entwined in between the keys like in figure 2.7. This has one major disadvantage though. The typical way we visit nodes is we read the key count then we read the keys until we find what we are looking for and finally read a child block to traverse down the tree. So the majority of the reads are done on keys in sequential order. By having the keys stored next to each other we only store information in the cache that is likely to be read next. The same reasoning holds for leaf nodes as well.



(a) File with child pointers



(b) Internal node



(c) Leaf node

Figure 2.6: Store format

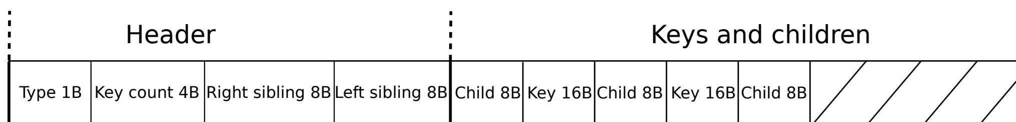


Figure 2.7: Entwined store format

2.3.3 Neo4j page cache and page fault

As explained in section 2.1.5 all operations on the tree are dominated by the number of disk accesses needed and keeping the most frequently accessed pages in memory will minimize the effort. To do this, all file access is done with the page cache implemented by neo4j.

The purpose of the page cache is to abstract away the fact that you are handling two different stores, disk and memory and simulate that everything is in memory. The algorithm makes sure that the most frequently referenced pages are always kept in memory while the rest of the pages are swapped in and out as they are accessed.

When we want to access a page that is not currently loaded into memory by the page cache this is called a *page fault*. The page cache then needs to get rid of one of the pages it currently has cached and write it back down to disk, this is called "evicting" a page. Then it has to read the newly referenced page into memory. As a page fault leads to expensive I/O operations we want to avoid this as much as possible and there are a few different page replacement algorithms that aim to do this.

The neo4j page cache implements the "GCLOCK" algorithm as presented in [21]. This is a simple explanation of the algorithm: Every page has a reference counter that is incremented on access and decremented by a dedicated "eviction thread". When the reference counter reaches zero the page can be released from memory and written back to disk. This only happens when a new page needs to be loaded from disk but the page cache does not have enough space left. In this way, reference history is kept.

It is not possible for the operating system to swap out a page that has been marked as internal memory by the page cache.

Chapter 3
Evaluation

This chapter is about the evaluation process. In section 3.1 we explain what we measure and how we measure it. Section 3.2 considers the topic of execution environment. In section 3.3 we present our environment setup. We discuss the datasets used in section 3.4. In section 3.5 we present how we evaluate usefulness in an OLTP environment by looking at response time on insert and finally in section 3.6 we discuss memory overhead for keeping an index. In chapter A in the appendix we present the queries we use to benchmark execution.

3.1 How do we evaluate performance?

In this project when we talk about performance we talk about query execution time. In particular we are interested in how much better or worse the performance gets when we use the Shortcut index compared to when we are not.

To distinguish between runs with and without the use of the Shortcut index we will use the slightly misleading terminology of Kernel and Shortcut, for example, "query x is 4 times faster on Shortcut compared to Kernel." This means execution time for query x is 4 times faster when using the Shortcut index compared to when only using the Kernel engine in neo4j. This is misleading because usually we cannot execute an entire query with only the Shortcut index, and we still use the neo4j Kernel. But we use the index as much as possible, typically to expand the last hop in the query.

3.1.1 The workload

A workload is a list of queries and a list of input values for every query, typically this is the start node id. Each query is executed once for every input value in the workload.

The workload is executed twice. The first run is used as a warm up and on the second we measure execution time. We use the warm up to simulate a production environment

where the database has been up and running for some time. This means that often used data is loaded into memory and the different caches, like the page caches used by the Shortcut index and neo4j Kernel. Before executing a workload it is assumed that all necessary indexes exist and loading the index from file is part of the warm up.

We use We are interested in two response times: Average response time and the response time for the query to return the first time, with the first input parameter.

3.2 Machine and tools

All experiments are executed with the machine described in table 3.1 with neo4j version 2.3.0-M03 and java compiler 1.8.0_60.

Machine:	MacBook Air
OS:	OS X Yosemite 10.10.2
Processor:	1.3 GHz Intel Core i5
Memory:	8 GB 1600 MHz DDR3

Figure 3.1: System specifications

3.3 Environment setup

In all of our benchmarks we use the following configuration for the Shortcut Index and the JVM.

Page size We use a page size of 8192 B. An internal node can then at most fit $(8192-21-8)/24 \approx 340$ keys and will at least contain $340/2 = 170$ keys. The leaf nodes can at most fit $(8192-21)/32 \approx 255$ keys and will contain at least $255/2 \approx 127$ keys.

Index page cache size The page cache used by the Shortcut Index is configured to be 1024 MB which in our experiments means that it will fit the entire index. This setup is not used for the "insert time" explained in section 3.5.

VM params We configure the heap size to be at least 1024 MB (-Xms 1024M) and at most 2048 MB (-Xmx 2048M).

Neo4j also uses a page cache and it is automatically configured to be half of the free memory after maximum heap size has been removed. This means the Neo4j page cache used in our experiments has a maximum size of $(8-2)/2 = 3$ GB. This is enough to fit the datasets that we use in memory.

3.3.1 No benchmarks on limited RAM

We state disc access to be the dominating cost when executing queries, if not all data can be kept in memory. Even so, all of the benchmarks that we run on queries is executed in an

environment that does fit all data in memory. The goal was to also perform these queries in an environment with less memory to see if the analysis matched reality. The limited time we had for this project did not allow us to do this however. We expand on this in section 5.4.4.

3.4 Datasets

We are working with two different types of datasets, LDBC and LAB. They are described in section 3.4.1 and 3.4.2 respectively.

3.4.1 LDBC Dataset

The Linked Data Benchmark Council (LDBC) has put together a benchmark suite called the Social Network Benchmark (SNB) that includes a way to generate datasets simulating social networks. More on LDBC and why we think it is a valid choice is discussed in section 1.4.2. The LDBC SNB datasets, from here on called LDBC datasets or just LDBC, can be generated in different sizes and the size is given by the "scale factor" (SF). SF001 means the dataset in comma separated value (CSV) representation has size 1GB. In this project we only use LDBC_SF001. The datasets are generated but much effort has been put into making them look and behave "real" [11].

In figure 3.2 we show the schema for LDBC and statistics of the dataset can be found in table C.1 in the Appendix.

3.4.2 LAB Dataset

The dataset family called LAB is used as a "laboratory environment", an environment that is not necessarily representative to real use cases but is well adapted to test specific features. It was designed by us specifically to be used in this project. It is purely synthetic and intentionally generated to be predictable. It consists of 10000 `(Person)` that have `[:CREATED]` `(Comment)`s, making up 10000 isolated islands of nodes. Every `(Person)`'s `(Comment)`s has `{date}` evenly distributed in the interval $[0, 8000)$. When generating a LAB dataset you provide a fan out value that determines the number of `(Comment)` for every `(Person)`. If fan out is 8, every `(Person)` has 8 `(Comment)`. The schema for LAB is shown in figure 3.3 and statistics for all the generated LAB datasets are shown in table C.2 in the Appendix.

The fan out of a specific LAB dataset is given by a number at the end. A LAB dataset with fan out 200 is thus called LAB200. By running our benchmarks on LAB datasets with different fan out we can see how *neighborhood density* affects response time.

The fact that the dates of the `(Comment)`s are evenly distributed in some range allows us to very accurately query for a precise percentage of the neighborhood volume. For example we can query for a range that only covers 1% of the total range and then we know that we will only get 1% of the neighborhood as a result. This is what we call *percent of neighborhood interest*.

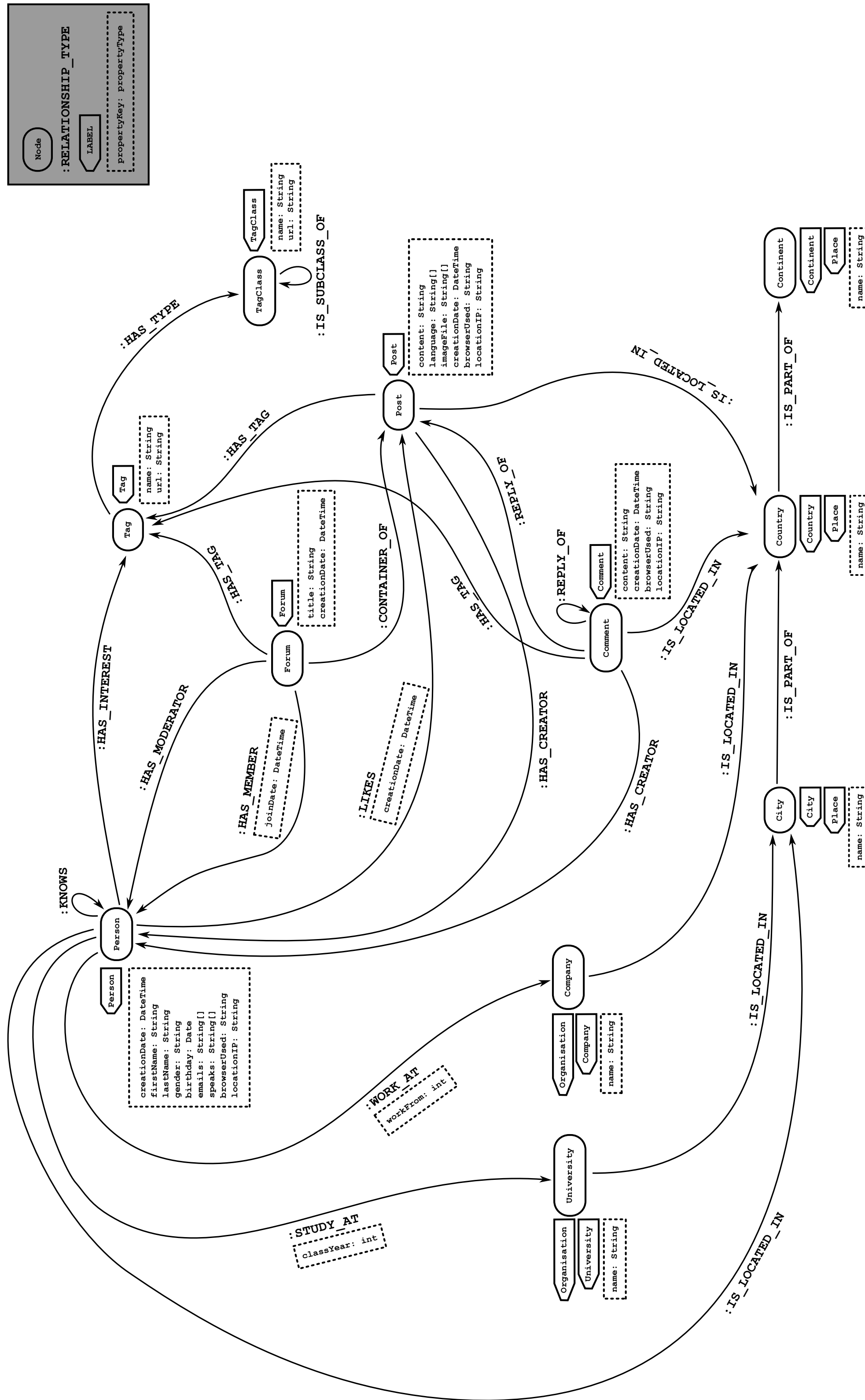


Figure 3.2: LDBC schema

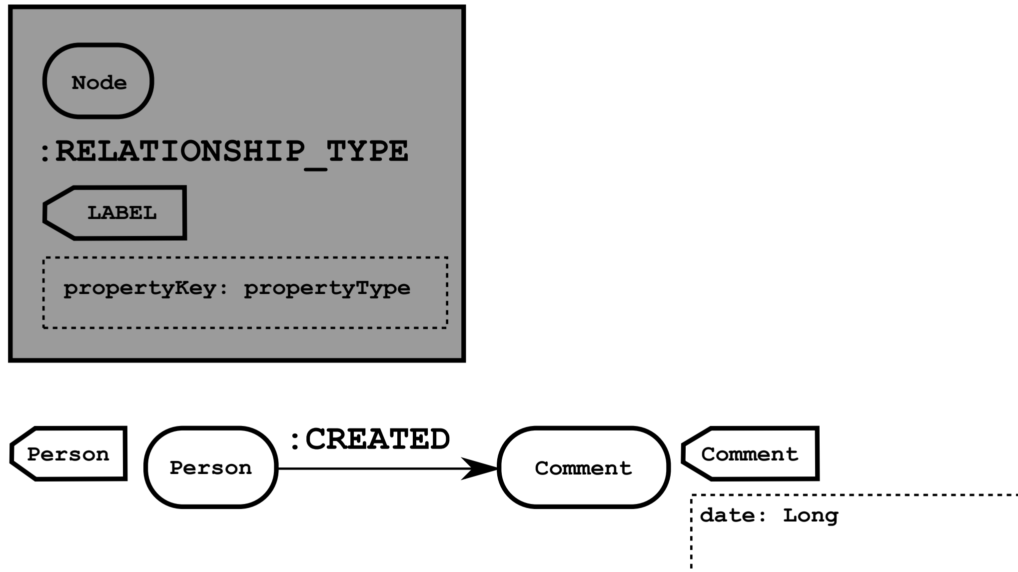


Figure 3.3: LAB schema

3.4.3 Cost analysis of range queries

We use LAB query 1 as an example query to analyze how the cost should scale for Kernel and Shortcut respectively. As we have stated in section 2.1.5 and 2.3.3 the cost of accessing the index and accessing data in the dataset is dominated by the number of disk accesses that is necessary. Therefore we analyze the cost of performing range queries with the assumption that not all data can fit inside memory.

In our experiments however, we can fit all data inside memory. The analysis made here and the results from our experiments should therefore be seen as complimentary.

We do not analyze the cost of performing queries when all data can fit inside memory because it is not very clear what dominates execution.

Not all data in memory

We analyze the cost with respect to the number of different pages we read from in the worst case. This is a reasonable approach because our assumption is that reading new pages into memory is what dominates execution time and for each new page we read from we risk a page fault. Let p be the cost of one page read from disk. Let d be the density of the neighborhood, n be the number of paths indexed in the Shortcut index and $k <= d$ the number of keys in the query range.

Kernel needs to expand the "created" relationship, read the Comment node and read the "date" property. All of this data can possibly be scattered among different pages so it adds up to 3 page reads in the worst case. Kernel needs to examine the entire neighborhood so the cost for LAB query 1 is $3dp$ for Kernel.

Shortcut needs to run a find operation on the B+ tree. Let the B+ tree have order o. The find operation then has complexity $\log_o(n)$. When the first key in the range is found we need to do a sequential scan until we find the last key in the range, that means reading k keys. The cost for this is $\lceil \frac{k}{o} \rceil p$. In total the cost for Shortcut is $(\log_o(n) + \lceil \frac{k}{o} \rceil)p$.

The order of the B+ tree is $(pageSize - headerSize)/(2*(keySize + childPointerSize))$. If we assume that we have a page size of 8192B, which is what we use in our implementation, then the order of the B+ tree is $o = \frac{8192-21}{2\cdot(16+8)} \approx 85$. If we use a LAB dataset then we know that we have $10000d$ paths indexed. The time complexity for Shortcut is then $(\log_{85}(10^4 d) + \lceil\frac{k}{85}\rceil)p$ and Kernel still has $3dp$. As $k <= d$ it is clear that Shortcut scales better than Kernel in the worst case scenario.

It is worth noting that this worst case scenario is highly unlikely to occur with warm caches, both for Shortcut and for Kernel and we should not expect the actual results to map very well to this analysis. We try to capture a "typical" production behavior with our benchmarks. Performing a theoretical analysis of a "typical" execution is hard because it depends on:

- How the data is stored on disk, which depends on the order in which it was created.
- What pages are already cached by the page cache, which depends on what data has been accessed previously.

3.5 Insert time

As discussed in section 2.2.2 overhead for index updates cannot be too high if the index is to be used in an OLTP environment. To see if our implementation could be a suitable candidate we measure response time on insert.

3.5.1 What is insert time overhead?

The overhead of the insert operation is the time it takes to insert a new entry into the index. We want to simulate the scenario where a new node or a new relationship has been added to the database that enforces a single index entry to be added. We measure the insert time in an index that has been up and running for a while, not a completely new index. So we measure insert time for 100,000 entries on an index that already has 10,000,000 entries. This way, each single insert does not make the relative difference between each insert that large and we think this is typical in a production environment. We also measure how page cache size affects insert time by letting the page cache cover a specified percentage of the index size.

The index keys and index values are produced with a random generator. We do not do any measurements on inserting sequential entries.

Because we know that the topmost nodes in the B+ tree will be visited much more frequently compared to nodes further down in the tree, especially the leaf nodes, we do not expect the insert time to scale linearly with the page cache coverage percentage.

As we only measure overhead of the actual insert operation we do not cover the entire overhead of maintaining the index on updates to the database. Other things that are also part of the overhead but not covered in this project are as follows.

- When creating a new relationship we need to see if a path is created that matches some indexed pattern.

Table 3.1: Index size compared to dataset size

Dataset	Dataset size	Index size
LAB8	7.4 MB	5.2 MB
LAB40	36 MB	26 MB
LAB200	180 MB	130 MB
LAB400	360 MB	260 MB
LAB800	720 MB	520 MB

- When we have decided that a new entry needs to be inserted into the index, all the necessary data, node ids, relationship id and property value, needs to be read from the store.

3.6 Memory overhead

Keeping the Shortcut index in memory is a cost in itself. In table 3.1 we a size comparison between the LAB datasets and respective Shortcut index. Note that all data in the LAB datasets is indexed. This is of course not usually the case as you would normally have a lot of data that should not be indexed, such as properties or nodes and relationships that are not part of dense neighborhoods and / or is not frequently queried for.

Chapter 4

Result and discussion

In this chapter we will present and discuss the results from the benchmarks described in chapter 3. We start by briefly expand on how we decided what queries to run in section 4.1. In section 4.2 we talk about the result tables that can be found in the Appendix. In section 4.3 we analyze the performance in terms of response time. This will be done one query at a time. In section 4.4 we will discuss insert times and finally in section 4.5 we will briefly comment on the memory overhead of keeping an index.

4.1 How did we decide what queries to run?

First we decided to run a small set of fairly simple queries that tested different interesting features, such as order by, limit and so on. We also decided to do this on neighborhoods with different density. Those are the LDBC queries. We ran those on LDBC_SF001 and got the results shown in table B.1 in the Appendix. We saw what seemed to be a correlation between neighborhood density and increase in query performance. For example, LDBC Query 2 touches a much denser part of the graph compared to LDBC Query 4 and the speedup when using the Shortcut index is much better for LDBC Query 2. We will expand on this in section 4.3.

To further investigate how neighborhood density actually affected performance we created the LAB environment and we discuss this more in section 3.4.2 and A.2.

4.2 Result tables

We present the benchmark results in the following tables which can all be found in chapter B in the Appendix:

Table B.1 Results for LDBC queries 1-6 on LDBC_SF001.

Neighborhood	Avg density	Touched by query
(:Person)<-[:HAS_CREATOR]-(:Comment)	202	Holy Grail, LDBC query 1, 2
(:Person)-[:LIKES_POST]->(:Post)	71.6	LDBC query 3
(:University)<-[STUDY_AT]-(:Person)	1.24	LDBC query 4
(:Company)<-[WORK_AT]-(:Person)	13.9	LDBC query 5
(:Forum)-[:CONTAINER_OF]->(:Post)	11	LDBC query 6
(:Person)-[:KNOWS]-(:Person)	35	Holy Grail

Table 4.1: Interesting neighborhood densities in LDBC_SF001

Table B.2 Results for LAB queries 1-3 on LAB8.

Table B.3 Results for LAB queries 1-3 on LAB40.

Table B.4 Results for LAB queries 1-3 on LAB200.

Table B.5 Results for LAB queries 1-3 on LAB400.

Table B.6 Results for LAB queries 1-3 on LAB800.

Table B.7 Results for the Holy Grail Query on LDBC_SF001.

For all queries except LDBC query 1, which is a complete scan, we execute the query multiple times with different input data. What we see in the tables is the response time for the query to return the first time (first) and the average response time for the query (avg). We log this both for Kernel and for Shortcut. We also include the "speedup" which simply is the Kernel response time over the Shortcut response time.

4.3 Query analysis

In this section we revisit the list of queries presented in section A in the appendix and discuss the result for each query to see if we got the result we expected. We start off by looking at the LDBC queries in section 4.3.1. We then move on to the LAB queries in section 4.3.2. We take a look at the Holy Grail Query in section 4.3.3 and finally in section 4.3.4 we discuss the "first" results from all the result tables.

4.3.1 The LDBC queries

We start with the LDBC queries. In figure 4.1 we can see a chart showing the comparison between Kernel and Shortcut in terms of response time for the different queries. Note the difference between the y-axis scales. In table 4.1 we see the densities of the neighborhoods that LDBC query 1-6 touches.

LDBC Query 1 Scan

As expected, it is much faster to scan all the leaves in the Shortcut index compared to expanding all "has creator" relationships, by more than an order of magnitude, 13x.

LDBC Query 2 Seek

We have an average performance improvement that is very similar to LDBC Query 1, 15x. Note that they touch the same neighborhood, namely `(Person)<-[COMMENT_HAS_CREATOR]-(Comment)` which has density 202.

LDBC Query 3 Order by

We have a speedup of 9x and the density is 71.6. It is not clear if we get any immediate performance gain by not having to sort the result. If we compare to query 2 we get some sense of correlation between neighborhood density and performance gain.

LDBC Query 4 Exact match

Here we see a performance regression when using the index. The interesting neighborhood, `(University) <-[STUDY_AT]- (Person)`, only has density 1.24 which is the lowest of all queries. This could be an explanation as to why we see this poor performance. We expected performance to be better because we have limit the *percent of neighborhood interest* and this should work to the advantage of the Shortcut index.

LDBC Query 5 Range

For this query we see an average speedup of 6.6x and the density of the neighborhood is 13.9. The performance improvement was expected. Although when we compare it to the speedup we got in query 2, which was 15x, it seems a little high. Recall that query 2 touched a neighborhood with avg density 202, more than one order of magnitude larger. This could be related to the fact that query 5 limits the *percent of neighborhood interest*.

LDBC Query 6 Range

The same reasoning we used for LDBC query 5 applies to LDBC query 6 as well. We see an average speedup of 2.5x and the neighborhood has average density 11.

What did we learn and how to proceed?

The LDBC queries let us "test the water". We still do not have a clear view over what affects performance and how much, but a few questions were raised.

- How does neighborhood density affect performance?
- How does sorting affect performance?
- How is performance affected when we limit the amount of data in a neighborhood that we are interested in, the *percent of neighborhood interest*?

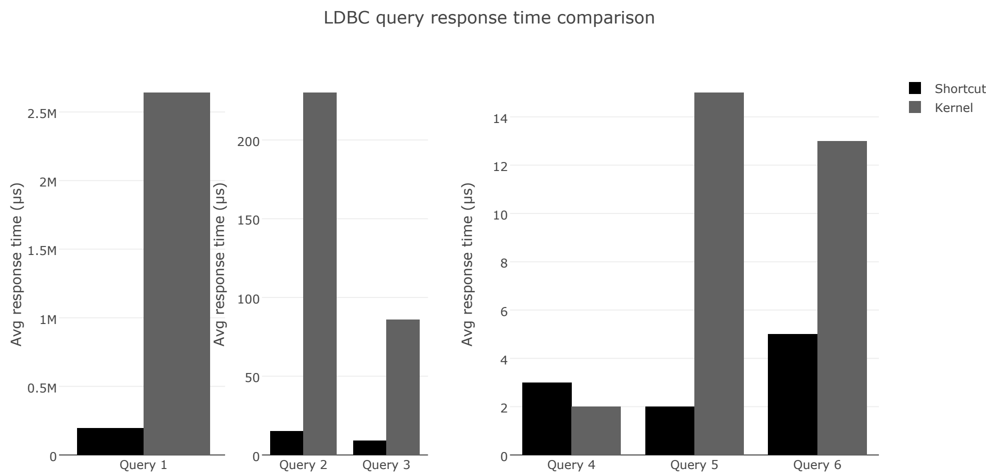


Figure 4.1: Average response time plot for LDBC queries 1-6

We will answer those questions using the LAB environment queries.

4.3.2 The LAB queries

In this section we discuss benchmark results for the LAB queries. In general we have two variables to consider when we do our benchmarks and try to make sense out of the results. Those two variables are *neighborhood density* and *percent of neighborhood interest*. This is illustrated with clarity using LAB Query 1 and 2. We also want to investigate how sorting affects performance and this is what LAB query 3 does.

LAB Query 1 "percent of range"

LAB query 1 limits *percent of neighborhood interest* and from a quick look at the result tables covering the LAB queries, table B.2-B.6 in the Appendix, we see two things. A higher density gives a higher performance gain and the more restricted *percent of neighborhood interest* the better performance gain.

In figure 4.2 we see how Kernel and Shortcut response times scale different when we increase the *percent of neighborhood interest* and fix the density. The values come from table B.4 and B.6. Kernel does not scale at all, it always hits the worst case scenario where the entire neighborhood needs to be considered. Shortcut on the other hand has linear scaling. This is because the amount of data that needs to be considered increases linearly with *percent of neighborhood interest*. Note that Kernel and Shortcut are not plotted against the same y-axis and that the scale difference is of one order of magnitude.

When we fix the *percent of neighborhood interest* and instead vary density we get the charts shown in figure 4.3 and 4.4. It is quite clear that both Kernel and Shortcut response times scale linearly with density. Although the constant of the linear scaling varies with *percent of neighborhood interest* for Shortcut, it remains the same regardless for Kernel.

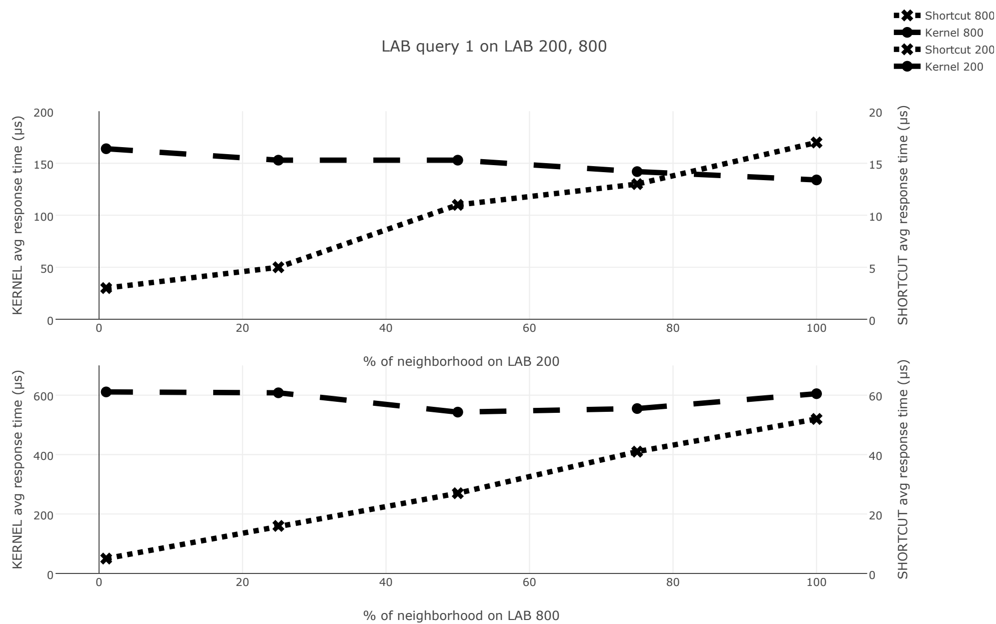


Figure 4.2: Average response time for LAB query 1 on LAB200 and LAB800. Note the scale difference between the two y-axes.

This again is due to the amount of data that needs to be considered both for Kernel and Shortcut.

LAB Query 2 "limit"

As described in section A in the appendix, LAB query 1 does not portray a likely real world use case, we are more often interested in some fixed number of "top x" values. This is what LAB query 2 does and we run it with limit 4 and limit 40, which of course means, "top 4" and "top 40" of some ordering. In figure 4.5 we see how the response time scales with density.

We are quite surprised to see that the last data points differ so much between Kernel limit 4 and Kernel limit 40 as the only difference in execution should be to limit the sorted result list to 4 instead of 40 which should not have any additional cost. This unusual behavior can have multiple explanations, but our bet is that the JVM hit a major garbage collection on this particular run. In theory, Kernel limit 4 and limit 40 should behave almost identically.

That being said, Kernel limit 40 seems to scale linearly with density which aligns well with previous experiments.

Shortcut does not seem to scale at all but is rather stable around 6µs. This further strengthens our belief that both Kernel and Shortcut response times scale with the amount of considered data.

It is surprising to us that Shortcut limit 40 does not have a higher response time than Shortcut limit 4, as the results from LAB query 1 quite clearly showed that Shortcut response time scales linearly with the amount of considered data. Although, if we look at

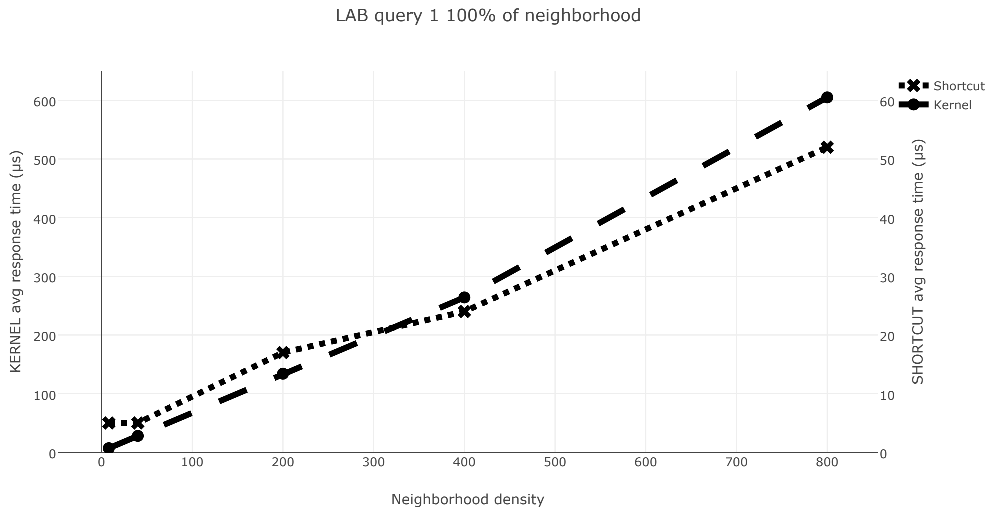


Figure 4.3: Average response time for LAB query 1 100% on varying density levels. Note the scale difference between the two y-axes.

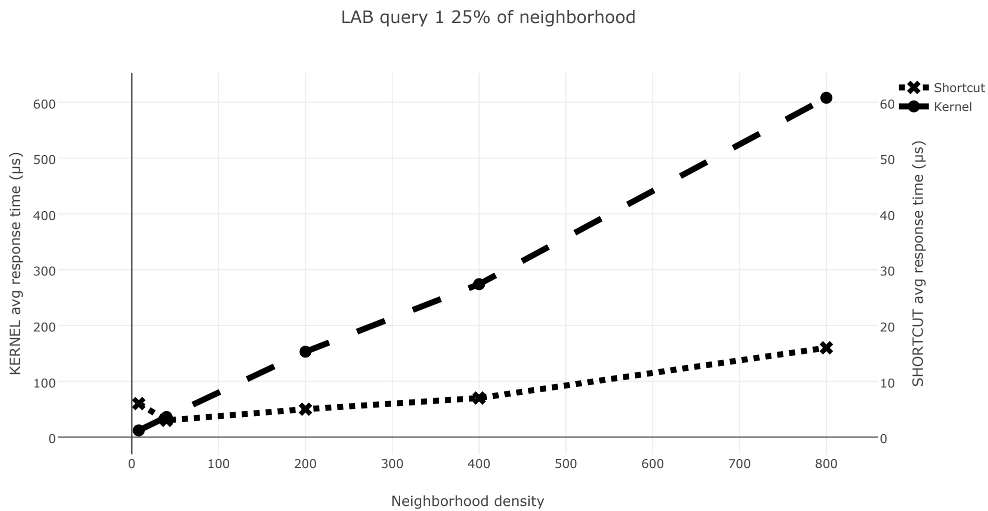


Figure 4.4: Average response time for LAB query 1 25% on varying density levels. Note the scale difference between the two y-axes.

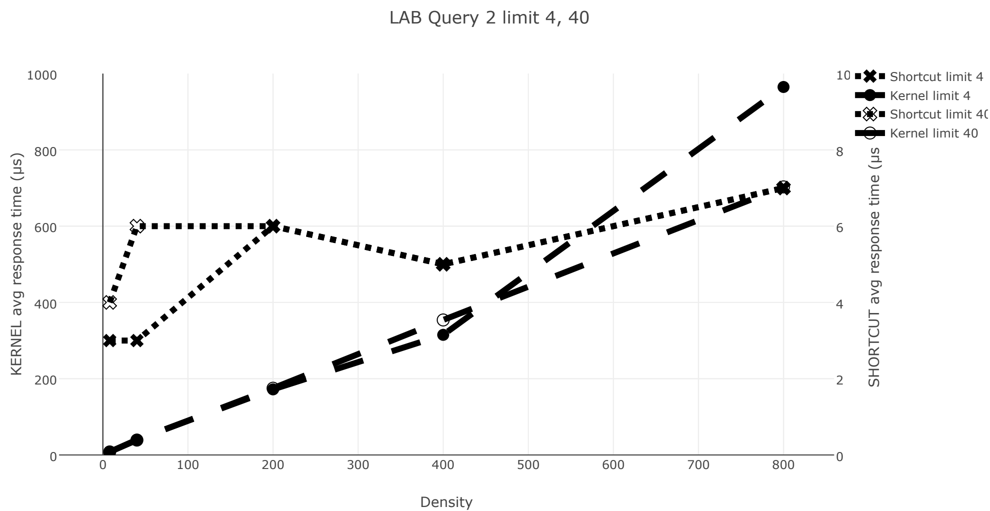


Figure 4.5: Average response time for LAB Query 2 with limit 4 and 40. Note the scale difference between the two y-axes.

the first two data points for Shortcut in figure 4.3 we see that the scaling actually does not start until after density 40. It seems likely that up until that density the response time is dominated by something other than traversing the index tree. That would explain both the results for LAB query 2 and LAB query 1.

LAB Query 3 "order by"

The goal of LAB query 3 is to examine how sorting affects response time. Recall that we expected that only Kernel's response time would be affected by sorting. In figure 4.6 we have plotted the response times for LAB query 3 and LAB query 1 100%. The only difference between those is that LAB query 3 sorts the result. Looking at the plot we see that Kernel query 3 has a higher response time compared to Kernel query 1 on all densities. We also see that the difference seems to grow with the density. As sorting has time complexity $O(n \log(n))$ we can safely say that this is how the difference grows as well. For Shortcut however, response time seems to be the same for Lab query 3 and 1, which is expected because execution is identical.

4.3.3 The Holy Grail

The Holy Grail touches the neighborhood `(Person)-[KNOWS]-(Person)<-[COMMENT_HAS_CREATOR]-(Comment)` which has density $35 \cdot 202 = 7070$. We do not have the whole neighborhood indexed though, only the last hop: `(Person)<-[COMMENT_HAS_CREATOR]-(Comment)`. In section 1.4.1 and 2.2.1 we explained how Shortcut only needs to consider $35 \cdot 20 = 700$ Comments while Kernel needs to consider all 7070. As explained in section 3.4.3 the cost of reading a small range of keys is often the same as reading only one key due to *range locality*. To read 20 keys at most two B+ tree leaves will have to be visited

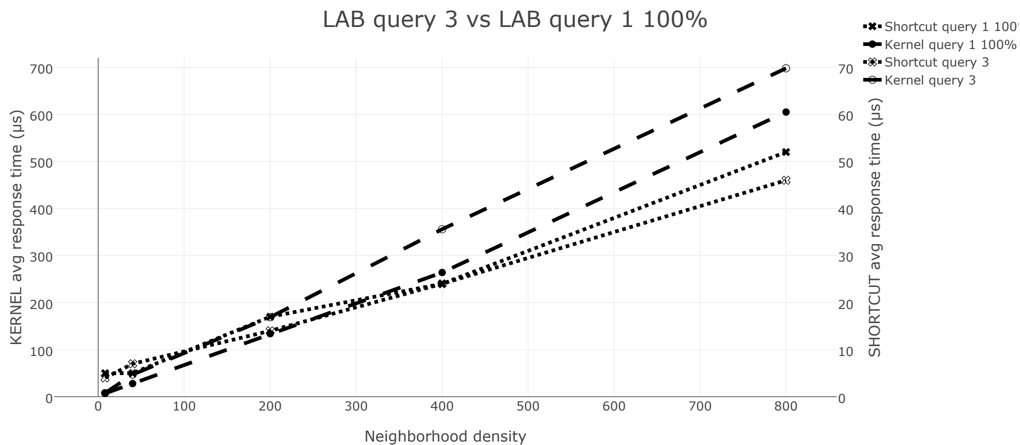


Figure 4.6: Average response time for LAB query 3 and LAB query 1 100%. Note the scale difference between the two y-axes.

and in most cases only one. So while Kernel in the worst case scenario needs to read from 7070 pages, only including the last and dominant hop in the query, Shortcut only needs to read from at most $35 \cdot 2 = 70$ pages. This maps very well to the average speedup of 103x that we see in table B.7.

4.3.4 Result for first query to return

If we compare the "first" with "avg" response times for Shortcut, in tables B.1-B.7 in the Appendix, we see that "first" consistently has a significantly higher response time. We unfortunately cannot explain what causes this. It is not caused by the data not being loaded into the caches because we run the exact same workload as warm up right before we start measuring execution.

It could be the case that the JVM decides to do a just in time compilation after the first query returns but it seems unlikely that this happens for every query, every time, independently of what queries and what dataset is being used.

It could also be the case that the JVM optimizes some object allocation that we cannot foresee.

The code is available in a public repository on GitHub [18].

4.4 Insert times

This section covers the overhead for performing insertions into the Shortcut index and how it relates to the size of the used page cache. We talk about page cache size in terms of how large a percentage of the entire index it can cover. We call this page cache coverage.

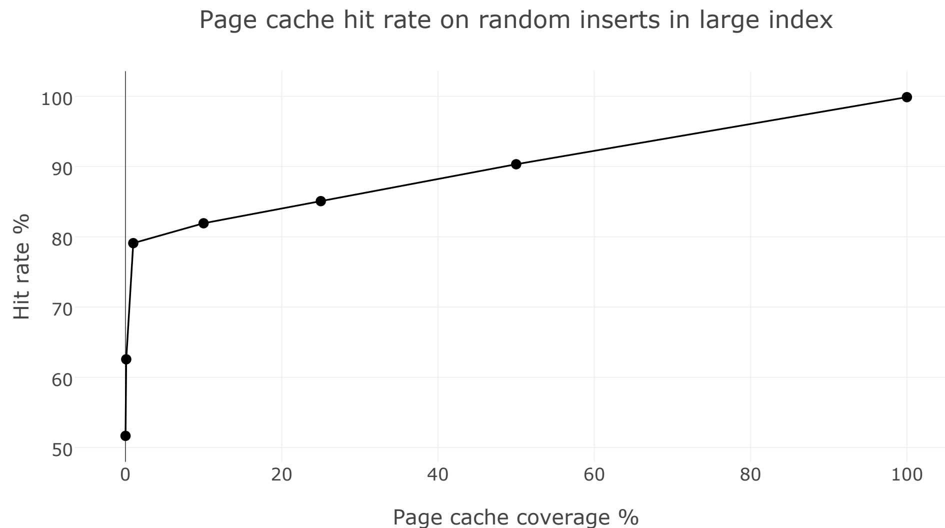


Figure 4.7: Page cache hit rate on random inserts in large index.

4.4.1 Page cache hit rate vs page cache coverage

In figure 4.7 we see the page cache hit rate plotted against the page cache coverage. The values come from table B.8. We see that the hit rate is already quite high, at 80%, for a cache coverage of only 1% and with cache coverage of 50% we get a hit rate of 90%. This is of course due to the structure of the B+ tree. The topmost nodes in the tree will be visited for every insert and thus kept in the page cache. They are very few in number compared to the leaf nodes and can therefore be kept in memory even if the size of the page cache is relatively small compared to the size of the entire index.

4.4.2 Insert time vs page cache coverage

In figure 4.8 we see how the insert time is affected by the page cache coverage. We plot the average insert time together with the 99th, 95th, 90th and 50th percentile. In figure 4.9 only the average insert time is plotted. There are some interesting things that we can deduce from these plots and the table.

- The average insert time is almost halved when increasing the page cache coverage from 1% to 100%.
- The 99th percentile is roughly quartered when increasing the page cache coverage from 1% to 100% and almost cut to one third when going from 50% to 100%.
- When increasing the page cache coverage from 0.01% to 1% the 50th, 90th, 95th percentile and average insert time drops to between one half and one third. The 99th percentile however almost does not drop at all. It is not until the page cache covers 50-100% of the index size that we see a larger drop in insert time. In particular when moving from 50% to 100%, we see a huge improvement for the 99th percentile. When looking at figure 4.8 and 4.9 it is quite apparent that the improvement in insert

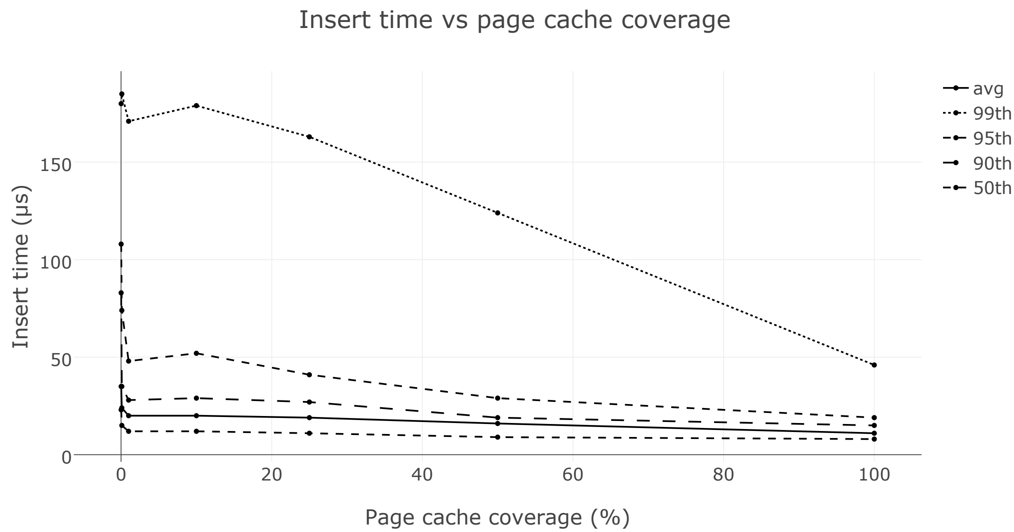


Figure 4.8: Insert time vs page cache coverage.

time is quite large when going from 0.01% to 1%. This also matches well with the "hit rate plot" in figure 4.7. This tells us that we can get an average performance that is quite reasonable even if that page cache can only hold parts of the index in memory but to get rid of the worst case scenarios you need to have it all in memory.

An average insert time of between 11 and 35 µs should be acceptable in most use cases.

4.5 Memory overhead

As shown in section 3.6, the size of the index does not grow beyond the size of the dataset, even if all data is kept in the index. Of course, the scenario where all data is indexed is very extreme and is not likely to occur in a real use case.

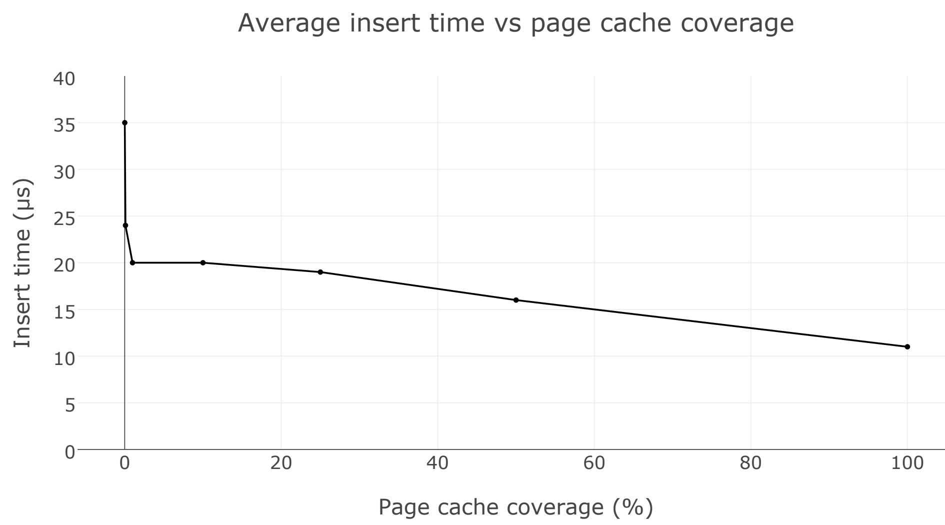


Figure 4.9: Average insert time vs page cache coverage.

Chapter 5

Conclusions

In this project we have made an attempt at solving the node density problem by indexing paths. We have presented a novel path index design and evaluated our implementation on a small number of queries and datasets. Here are our conclusions.

5.1 Response time improvement

Our results suggest that our design could improve query execution performance in terms of response time and for reasonably dense neighborhoods an improvement of between one and two orders of magnitude is not unusual.

In particular, a path index of this kind would be beneficial for queries for which the following statements are true.

- The query limits the *percent of neighborhood interest* with a range predicate on a property value or by having the result sorted on the property value and only return some limited number of result rows.
- The query touches a neighborhood with large density.

The response time can be limited because we can limit the number of different pages that we need to read data from by utilizing the *neighborhood locality* and *range locality*. Those are concepts introduced by us and they simply mean that all data that belongs to the same neighborhood is stored together and within that neighborhood all data within the same property range is stored together on disk.

5.2 Scaling with percent of neighborhood interest

What makes our solution extra appealing is that it lets response time scale with the *percent of neighborhood interest* of the query by actually limiting the amount of data that needs to be considered. By contrast, without the index you would always hit the worst case scenario of considering all the data in the neighborhood.

5.3 Insert time

Depending on how much of the entire Shortcut Index the page cache could keep in memory at the same time we saw an average insert time between 11 and 35 µs. For most use cases this should be an acceptable trade off and we conclude that the Shortcut Index could be used in a real production environment.

5.4 Limitations and unknowns

We have made limitations to our implementation, how we measure performance and what we consider performance to be. This is what we discuss in this section.

5.4.1 Implementation

Our implementation of the Shortcut Index is purely single threaded and cannot handle concurrent work of any kind. This is of course a complete necessity if it should be used in a real production environment. This means that the workloads we use to benchmark performance is also single threaded and we only execute one query at a time. It is not realistic to assume that this is the typical case for a real workload on a database. It is not very clear how this will affect performance of the Shortcut Index and more investigation should be done in this area.

5.4.2 Linear vs binary search

Our design has an obvious flaw in that it uses linear search when finding keys in the tree nodes. It would be interesting to know how much we could improve performance by instead using binary search.

5.4.3 Response time as performance

We have only measured performance in terms of response time. It is not obvious however that this is the most reasonable approach. Response time is affected by things like, but not limited to, the following.

- How high memory pressure is from the rest of the system.

- How high pressure is on the CPU from the rest of the system.
- How warm the different components of the system are.

Another way to measure performance could be to count the number of page accesses and page faults needed for each query. This would make sense since we assume that this is the dominating cost during execution.

The most sensible thing would be to measure both response time and number of page accesses and page faults. That would give us the possibility to actually correlate the response time and disk access. As far as we reached in this work, the assumption that disk access is what dominates execution time is still just an assumption.

5.4.4 Configuration

When testing the response time on insert we varied page cache size. This should have been done when we benchmarked response time on query execution as well to get a better understanding of how the index works when varying the availability of memory. This would be especially interesting if we also measured the number of page accesses and page faults as discussed in the section above.

We should also have made more adjustments to the size of the page cache used for Neo4j. It is not obvious that we can keep the entire database or index in memory in a production environment and this should have been reflected in our benchmarks. Especially since disc access is assumed to be the dominating cost when executing queries.

5.4.5 Datasets and environment

Our design and implementation is only tested on a very limited set of queries and datasets, both in terms of structure and size, and only tested on one single machine. It is not necessarily the case that the same performance gain can be expected on other machines as well. What we feel is missing in particular are benchmarks on a much more powerful machine, like something typically used in a real production environment, on much larger datasets of different kinds.

5.4.6 Response time for "first" query

In section 4.3.4 we discussed a result that we do not understand. This is the fact that it is always slower for a query to return for the first input data compared to the average. This remains an unknown for us and it makes us doubt if our results really are as good as they seem. It could be that we execute our workload in much too "streamlined" a way that makes optimization easy for the JVM and thus we see results not realistic under a more randomized workload.

Further testing with a more randomized workload could be part of future work.

5.5 Did we solve the problem?

One of the main goals and motivations for this work was to improve the Holy Grail query, see query 1.1. From the result shown in table B.7 in the Appendix and from the discussion in section 4.3.3 we can conclude that this goal was achieved.

5.6 Final conclusion

Within our limitations we can conclude that the Shortcut Index improves response time at a reasonable cost and is especially useful when indexing dense neighborhoods that is often queried with limitations on some property value. It is our firm belief that further research in this area could bring great academic value and open up many opportunities for graph database vendors. In section 5.7 we list suggestions for future work.

5.7 Future work

There are of course some questions that have not been addressed in this work for different reasons. The questions we would suggest for future work in this area are as follows.

Concurrent environment How does the Shortcut index perform in a multi-threaded environment? Every reasonable database allows concurrent transactions and it is not completely obvious that the same performance can be expected on parallel workloads as on single threaded execution.

Page size How does the page size of the page cache affect performance? From the analysis made in this work it is clear that the page size has some impact on how the time complexity scales for the Shortcut index. We do not investigate how large this impact is and we feel that this is a shortcoming in our work.

Page faults, page accesses and profiling Measure the number of page faults and page accesses on a per query basis and correlate it with the execution time. This should be done to confirm that the number of page faults and page accesses is what actually dominates execution time and potentially find other bottlenecks not discovered in this project. As an extension to this, profiling execution could also be done to further analyze where optimization would be most useful.

Garbage collection How much does garbage collection affect performance for the Shortcut index? Some efforts not presented in this work have been made to limit the number of java objects created during execution and the results indicate that garbage collection has a significant impact on execution time. This is not confirmed and the question could serve as a basis for a new project.

Execution environment We only execute our experiments on a single machine with a very limited number of datasets and queries and with a quite "streamlined" workload. It would be interesting to see if the same performance gain manifests on a more

powerful machine and larger datasets on a more randomized and perhaps more realistic workload.

Chapter 6
Bibliography

[1] Db-engines ranking by database model. http://db-engines.com/en/ranking_categories. Accessed: 2015-09-28.

[2] Db-engines ranking of graph DBMS. http://db-engines.com/en/ranking/graph+dbms. Accessed: 2015-09-28.

[3] Information about how the ldbc organization works. http://ldbcouncil.org/industry/organization. Accessed: 2015-10-02.

[4] Neo4j repository. https://github.com/neo4j/neo4j. Accessed: 2015-09-28.

[5] Neo4j website. http://neo4j.com/. Accessed: 2015-09-28.

[6] Alex Averbuch and Arnau Prat. Benchmark design for navigational pattern matching benchmarking. http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf, Sep 2014. Accessed: 2015-10-01.

[7] Jack Belzer, Albert G. Holzman, and Allen Kent. *Encyclopedia of Computer Science and Technology*. Marcel Dekker Inc., 1980. Accessed 2015-11-09 at https://books.google.se/books?id=KUgNGCJB4agC&printsec=frontcover&hl=sv&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false.

[8] Chee-Yong Chan and Chee-Yong Chan. Bitmap index design and evaluation. *ACM*, 1998.

[9] Ted Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), june 1970.

[10] Douglas Comer. The ubiquitous b-tree. *Computing Surveys*, 2(2), June 1979.

[11] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload, June 2015.

[12] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems The Complete Book*. Pearson Education Inc., 2nd edition, 2009.

[13] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *ACM*, 2010.

[14] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM*, 1984.

[15] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. *CIDR*, 2007.

[16] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 2011.

[17] Oracle. Using indexes and clusters. `http://docs.oracle.com/cd/B28359_01/server.111/b28274/data_acc.htm#i2773`, 2015. Accessed: 2015-11-10.

[18] Anton Persson. Shortcut index repository. `https://github.com/burqen/Shortcut-index/tree/0.2`, 2015. Accessed: 2015-12-03.

[19] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2nd edition, 2015.

[20] Vivek Sharma. Bitmap index vs. b-tree index: Which and when? `http://www.oracle.com/technetwork/articles/sharma-indexes-093638.html`, 2005. Accessed: 2015-11-09.

[21] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[22] Jonathan Maxwell Sumrall. Path indexing for efficient path query processing in graph databases. Master's thesis, Eindhoven University of Technology, 2015. Accessed: 2015-11-03 at `http://alexandria.tue.nl/extra1/afstversl/wsk-i/Sumrall_2015.pdf`.

[23] Chris Vest. Neo4j page cache package info. `https://github.com/neo4j/neo4j/blob/3.0/community/io/src/main/java/org/neo4j/io/pagecache/package-info.java`, 2014. Accessed: 2015-11-09.

[24] Harry K.T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed file. *VLDB*, 1985.

[25] John Wu. Annotated references on bitmap index. `http://www-users.cs.umn.edu/~kewu/annotated.html`, 2013. Accessed: 2015-11-09.

Appendices

Appendix A

Target queries

In this chapter we present the queries used to benchmark the Shortcut index. We discuss what is interesting about them and how they are executed by Shortcut and Kernel respectively.

A.1 LDBC Queries

First, to explain how the different queries are executed we need to know what Shortcut indexes we have. Here are the patterns indexed on LDBC. They will be referred to as index 1-5:

1. `(Person)<-[COMMENT_HAS_CREATOR]-(Comment {creationDate})`.
2. `(Person)-[LIKES_POST {creationDate}]->(Post)`
3. `(University)<-[STUDY_AT {classYear}]-(Person)`
4. `(Company)<-[WORKS_AT {workFrom}]-(Person)`
5. `(Forum)-[CONTAINER_OF]->(Post {creationDate})`

LDBC query 1-6 is used to "test the grounds" for the Shortcut index. In what areas does it seem to work well? When does it not work well? Here is an overview of the LDBC queries, how they are executed on Kernel and Shortcut respectively and why they where chosen:

LDBC Query 1 Scan Find all Persons and all Comments they have created. This is the only query that does not take a start node as input and is therefore only executed once. *Query A.1*
For **Shortcut** this correspond to an index scan on index 1, see the list of indexed patterns above. That is, we need to go straight to the leaves in the B+ tree and scan

```
// QUERY 1 - SCAN
// All comments written by all persons
MATCH (p:Person) <-[r:COMMENT_HAS_CREATOR]- (c:Comment)
RETURN id(p), id(r), id(c), c.creationDate
```

Query A.1: LDBC Query 1

```
// QUERY 2 - SEEK
// All comments written by person
MATCH (p:Person) <-[r:COMMENT_HAS_CREATOR]- (c:Comment)
WHERE id(p) = {1}
RETURN id(p), id(r), id(c), c.creationDate
```

Query A.2: LDBC Query 2

through all of them.
Kernel on the other hand finds all Persons and expands all `[:COMMENT_HAS_CREATOR]` relationships to find all Comments.
Why this query?: This is the simplest query we can imagine but it is in general not an easy query to execute because of the amount of data that is asked for. We expect that scanning all leaves in the B+ tree is a lot faster than expanding all of the "has creator" relationships.
How many times?: We run this query only once as it does not take any input data.

LDBC Query 2 Seek Find Person with some given id and find all Comments created by that Person. *Query A.2*.
For **Shortcut** this is a find on index 1 for the first key with the given id, then a range scan until the last key with that id has been found.
As for LDBC Query 1 **Kernel** needs to expand the "has creator" relationships attached to Person with the given id to find the Comments.
Why this query?: This is a slightly more complex query compared to LDBC Query 1. It reads less data but has the important difference that **Kernel** will execute it in a similar manner as LDBC Query 1 while **Shortcut** will use a different approach, namely search the tree for a specific key instead of just scanning the leaves. We want to investigate how this affects performance.
How many times?: We run this query with 9987 different input parameters.

LDBC Query 3 Order by Find all Posts liked by Person with the given id and order the result by creationDate in descending order. *Query A.3*.
Shortcut index is sorted in ascending order so Shortcut does a find on index 2 for the last key with the given id and then range scan "backwards" to find the first key with the id. In this way, the result will be delivered in descending order.
Kernel uses the same approach as for LDBC Query 2 but we need to sort the result list.
Why this query?: The "order by" feature makes this query interesting. As the

```
// QUERY 3 - ORDER BY
// Most recently liked post by person
MATCH (p:Person) -[r:LIKES_POST]-> (s:Post)
WHERE id(p) = {1}
RETURN id(p), id(r), id(s), r.creationDate
ORDER BY r.creationDate DESC
```

Query A.3: LDBC Query 3

```
// QUERY 4 - EXACT MATCH
// All students studying at university 2010
MATCH (u:University) <-[r:STUDY_AT]- (p:Person)
WHERE id(u) = {1} AND r.classYear = 2010
RETURN id(u), id(p), r.classYear
```

Query A.4: LDBC Query 4

Shortcut index is already sorted, we expect better performance compared to Kernel which has to sort the result.
How many times?: We run this query with 9987 different input parameters.

LDBC Query 4 Exact match For University with the given id find all Persons who studied there during 2010. *Query A.4*.
Shortcut needs to do a find on index 3 for the first key with the given id and property equal to 2010 then range scan for all of the similar keys.
Kernel needs to expand all `[:STUDY_AT]` relationships to find the ones with `{classYear}` equal to 2010
Why this query?: By using `r.creationDate = 2010` we limit the part of the neighborhood that we are actually interested in. Our expectation is that this should work in favor of Shortcut as it can limit the total amount of data considered while Kernel still needs to touch everything in the neighborhood.
How many times?: We run this query with 6421 different input parameters.

LDBC Query 5 Range Find all Persons who started working at Company with the given id before 2010. *Query A.5*.
Shortcut performs a find on index 4 for the first key with the given id and range scan up to the first key with `{workFrom}` equal to or greater than 2010.
Kernel expands all `[:WORKS_AT]` relationships and filters out the ones outside the range.
Why this query?: The range limitation limits the amount of interesting data in the neighborhood, just as LDBC Query 4. LDBC Query 5 target a different part of the graph with a different density though. We expect this to affect performance gain.
How many times?: We run this query with 1575 different input parameters.

LDBC Query 6 Range Find all Posts created between two given times that is part of Forum with the given id. *Query A.6*.

```
// QUERY 5 - RANGE prop <
// Employees since before 2010
MATCH (c:Company) <-[r:WORKS_AT]- (p:Person)
WHERE id(c) = {1} AND r.workFrom < 2010
RETURN id(c), id(p), r.workFrom
```

Query A.5: LDBC Query 5

```
// QUERY 6 - RANGE <= prop <
// Posts posted to a forum in a time interval
MATCH (f:Forum) -[r:CONTAINER_OF]-> (p:Post)
WHERE id(f) = {1} AND {2} <= p.creationDate < {3}
RETURN id(f), id(r), id(p), p.creationDate
```

Query A.6: LDBC Query 6

Shortcut performs a find on index 5 for the first key with the given id and property higher than or equal to the lower bound of the range, then range scan for the last key with the id and property lower than the upper bound of the range.
Kernel expands all `[:CONTAINER_OF]` relationships and filters out all Posts that do not fit the range.
Why this query?: Same reason as for LDBC Query 4 and 5 but targeting another different part of the graph. This query is very similar to LDBC Query 5 and the major reason for using this is to test the feature of having a double sided range.
How many times?: We run this query with 10000 different input parameters.

A.2 LAB Queries

On LAB we have obviously indexed pattern `(Person)-[CREATED]->(Comment {date})`. All of the LAB queries are executed with 10000 different input parameters.

LAB Query 1 percent of neighborhood For a Person with the given id, find all the Comments written by that Person with a date less than some given value. QueryA.7.
As the date property is evenly distributed we can easily control how large a percentage of the Person-Comment neighborhood we are querying for by changing this range. In our experiments we query for 100, 75, 50, 25, 1% of the range, ergo neighborhood. Querying for only some part of a neighborhood is something that we call *percent of neighborhood interest* and we expand on this in section 3.4.2.

Query execution is similar to how LDBC Query 5 is executed, both for Shortcut and for Kernel.

LAB Query 2 limit For a Person with the given id, find all the Comments written by that Person and return the `{2}` oldest. QueryA.8.
In a real use case you would rarely query for a fixed percentage of the result set, you

```
// LAB Query 1
// Range covers % of total range
// 100%, 75%, 50%, 25%, 1%
MATCH (p:Person)-[r:CREATED]->(c:Comment)
WHERE id(p) = {1} AND c.date < {2}
RETURN id(p), id(r), id(c), c.date
```

Query A.7: LAB Query 1

```
// LAB Query 2
// Limit result count to 4
MATCH (p:Person)-[r:CREATED]->(c:Comment)
WHERE id(p) = {1}
RETURN id(p), id(r), id(c), c.date
ORDER BY c.date ASC
LIMIT {2}
```

Query A.8: LAB Query 2

would instead ask for some "top x". This query lets you do that and therefore we believe it has more resemblance with a real use case. We run this query with "top 4" and "top 40". The "limit" feature still restricts the *percent of neighborhood interest* just as LAB query 1 does. Query execution is similar to that of LDBC Query 3.

LAB Query 3 order by For a Person with the given id, find all the Comments written by that Person and return them in ascending order by date. QueryA.9.
This query should clarify how "order by" affects response time. Our expectation is that Kernel should suffer from sorting and that Shortcut should not be affected at all. Execution is similar to LAB Query 2 except we return the entire result set.

```
// LAB Query 3
// Order by
MATCH (p:Person)-[r:CREATED]->(c:Comment)
WHERE id(p) = {1}
RETURN id(p), id(r), id(c), c.date
ORDER BY c.date ASC
```

Query A.9: LAB Query 3

Appendix B
Result tables

Table B.1: Result table for LDBC_SF001

Query		Kernel (µs)	Shortcut (µs)	Speedup
LDBC Query1	first	2,641,692	199,476	13.24x
	avg	2,640,536	196,624	13.43x
LDBC Query2	first	362	112	3.23x
	avg	230	15	15.37x
LDBC Query3	first	139	57	2.44x
	avg	86	9	9.35x
LDBC Query4	first	23	45	0.51x
	avg	2	3	0.78x
LDBC Query5	first	43	37	1.16x
	avg	15	2	6.62x
LDBC Query6	first	34	27	1.26x
	avg	13	5	2.52x
Setup				
Dataset		LDBC_SF001		

Table B.2: Result table for LAB8

Query		Kernel (µs)	Shortcut (µs)	Speedup
Lab Query1 001%	first	115	148	0.78x
	avg	9	5	1.67x
Lab Query1 025%	first	78	62	1.26x
	avg	12	6	1.93x
Lab Query1 050%	first	48	34	1.41x
	avg	9	5	1.59x
Lab Query1 075%	first	28	54	0.52x
	avg	8	6	1.26x
Lab Query1 100%	first	31	28	1.11x
	avg	7	5	1.40x
Lab Query2 limit 4	first	29	70	0.41x
	avg	8	3	2.85x
Lab Query2 limit 40	first	127	82	1.55x
	avg	8	4	2.11x
Lab Query3 order by	first	147	400	0.37x
	avg	8	4	2.28x

Table B.3: Result table for LAB40

Query		Kernel (µs)	Shortcut (µs)	Speedup
Lab Query1 001%	first	188	116	1.62x
	avg	33	4	9.31x
Lab Query1 025%	first	76	53	1.43x
	avg	36	3	13.32x
Lab Query1 050%	first	50	27	1.85x
	avg	33	4	9.26x
Lab Query1 075%	first	57	48	1.19x
	avg	33	3	9.42x
Lab Query1 100%	first	46	42	1.10x
	avg	28	5	6.03x
Lab Query2 limit 4	first	94	66	1.42x
	avg	40	3	12.57x
Lab Query2 limit 40	first	148	58	2.55x
	avg	39	6	6.77x
Lab Query3 order by	first	146	429	0.34x
	avg	47	7	7.02x

Table B.4: Result table for LAB200

Query		Kernel (µs)	Shortcut (µs)	Speedup
Lab Query1 001%	first	188	118	1.59x
	avg	164	3	59.16x
Lab Query1 025%	first	172	112	1.54x
	avg	153	5	28.89x
Lab Query1 050%	first	148	39	3.79x
	avg	153	11	14.12x
Lab Query1 075%	first	145	51	2.84x
	avg	142	13	11.04x
Lab Query1 100%	first	150	66	2.27x
	avg	134	17	8.00x
Lab Query2 limit 4	first	376	211	1.78x
	avg	172	6	29.83x
Lab Query2 limit 40	first	231	139	1.66x
	avg	175	6	27.88x
Lab Query3 order by	first	316	251	1.26x
	avg	169	14	12.19x

Table B.5: Result table for LAB400

Query		Kernel (µs)	Shortcut (µs)	Speedup
Lab Query1 001%	first	309	185	1.67x
	avg	270	3	84.97x
Lab Query1 025%	first	527	126	4.18x
	avg	274	7	39.51x
Lab Query1 050%	first	327	81	4.04x
	avg	274	12	21.96x
Lab Query1 075%	first	243	46	5.28x
	avg	324	17	19.33x
Lab Query1 100%	first	276	72	3.83x
	avg	264	24	10.80x
Lab Query2 limit 4	first	1,418	176	8.06x
	avg	315	5	61.76x
Lab Query2 limit 40	first	397	78	5.09x
	avg	354	5	72.06x
Lab Query3 order by	first	657	245	2.68x
	avg	356	24	15.09x

Table B.6: Result table for LAB800

Query		Kernel (µs)	Shortcut (µs)	Speedup
Lab Query1 001%	first	547	153	3.58x
	avg	611	5	121.05x
Lab Query1 025%	first	528	84	6.29x
	avg	608	16	38.25x
Lab Query1 050%	first	486	109	4.46x
	avg	543	27	19.94x
Lab Query1 075%	first	552	71	7.77x
	avg	555	41	13.39x
Lab Query1 100%	first	488	77	6.34x
	avg	605	52	11.59x
Lab Query2 limit 4	first	1,288	145	8.88x
	avg	965	7	138.19x
Lab Query2 limit 40	first	584	134	4.36x
	avg	703	7	100.28x
Lab Query3 order by	first	1,140	264	4.32x
	avg	698	46	15.23x

Table B.7: Result table for Holy Grail on LDBC_SF001

Query		Kernel (μs)	Shortcut (μs)	Speedup
Holy grail	first	22,796	693	32.89x
	avg	26,495	257	102.99x

Table B.8: Insert time and page cache hit rate depending on page cache coverage

Cache coverage	Page cache hit rate	Avg (μs)	Percentile (μs)			
			99th	95th	90th	50th
0.01%	51.67%	35	180	108	83	23
0.1%	62.57%	24	185	74	35	15
1%	79.10%	20	171	48	28	12
10%	81.93%	20	179	52	29	12
25%	85.08%	19	163	41	27	11
50%	90.33%	16	124	29	19	9
100%	99.88%	11	46	19	15	8

Appendix C
Dataset statistics

Statistic	**SF001 (1.6 GB)**
Nodes	3,158,994
Relationships	16,800,936
(:Comment)	2,015,590
(:Post)	1,015,594
(:Forum)	92,210
(:Tag)	16,080
(:TagClass)	71
(:Person)	9,987
(:Company)	1,575
(:University)	6,421
(:City)	1,349
(:Country)	111
(:Continent)	6
(:Person)-[:LIKES_COMMENT]->(:Comment)	1,219,614
(:Comment)-[:HAS_CREATOR]->(:Person)	2,015,590
(:Forum)-[:CONTAINER_OF]->(:Post)	1,015,594
(:Person)-[:LIKES_POST]->(:Post)	714,592
(:TagClass)-[IS_SUBCLASS_OF]->(:TagClass)	70
(:Comment)-[HAS_TAG]->(:Tag)	2,597,375
(:Forum)-[HAS_MEMBER]->(:Person)	1,599,016
(:Post)-[HAS_CREATOR]->(:Person)	1,015,594
(:Tag)-[HAS_TYPE]->(:TagClass)	16,080
(:Comment)-[:IS_LOCATED_IN]->(:Country)	2,015,590
(:Forum)-[HAS_MODERATOR]->(:Person)	92,210
(:Person)-[:HAS_INTEREST]->(:Tag)	233,336
(:Person)-[STUDY_AT]->(:University)	7,983
(:Post)-[:HAS_TAG]->(:Tag)	693,027
(:Comment)-[:REPLY_OF]->(:Comment)	1,024,042
(:Forum)-[:HAS_TAG]->(:Tag)	315,559
(:Person)-[:IS_LOCATED_IN]->(:City)	9,987
(:Person)-[WORK_AT]->(:Company)	21,930
(:Post)-[IS_LOCATED_IN]->(:Country)	1,015,594
(:Comment)-[:REPLY_OF]->(:Post)	991,548
(:Person)-[:KNOWS]->(:Person)	177,149

Table C.1: Statistics for LDBC SF001 dataset

Statistic	**LAB008 (7.4 MB)**
Nodes	90,000
Relationships	80,000
(:Comment)	80,000
(:Person)	10,000
(:Person)-[:CREATED]->(:Comment)	80,000
Statistic	**LAB040 (36 MB)**
Nodes	410,000
Relationships	400,000
(:Comment)	400,000
(:Person)	10,000
(:Person)-[:CREATED]->(:Comment)	410,000
Statistic	**LAB200 (180 MB)**
Nodes	2,010,000
Relationships	2,000,000
(:Comment)	2,000,000
(:Person)	10,000
(:Person)-[:CREATED]->(:Comment)	2,000,000
Statistic	**LAB400 (360 MB)**
Nodes	4,010,000
Relationships	4,000,000
(:Comment)	4,000,000
(:Person)	10,000
(:Person)-[:CREATED]->(:Comment)	4,000,000
Statistic	**LAB800 (720 MB)**
Nodes	8,010,000
Relationships	8,000,000
(:Comment)	8,000,000
(:Person)	10,000
(:Person)-[:CREATED]->(:Comment)	8,000,000

Table C.2: Statistics for the LAB datasets

EXAMENSARBETE The Shortcut Index
STUDENT Anton Persson
HANDLEDARE Krzysztof Kuchcinski (LTH), Johan Svensson (Neo Technology)
EXAMINATOR Per Andersson

Snabb sökning i kopplad data

POPULÄRVETENSKAPLIG SAMMANFATTNING **Anton Persson**

Den data som skapas, hanteras och analyseras idag är allt mer "kopplad". För att hantera den sortens data krävs grafdatabaser. Vi visar i detta arbete hur vi kan göra svarstiderna 10100 gånger snabbare.

Typen av data som vi hanterar har förändrats mycket sedan slutet av 90´talet. Framför allt på grund av internet. Då använde vi datorer för att digitalisera lönelistor och svar på formulär. Denna typen av data är isolerad och har ett eget värde i sig självt. Men den data som skapas på internet ser annorlunda ut. Den är "kopplad" och mycket av informationen ligger i hur datan relaterar till sin omvärld.

Ta Facebook som exempel. Jag har en profil med lite information om mig själv. Men du är mer intresserad av vilka gemensamma vänner vi har. Dvs, våra relationer till andra människor. Jag har också skrivit kommentarer, men vad som är mer intressant är vad jag har kommenterat på. Alla dessa kommentarer, profiler och bilder är små noder av data och de relaterar till varandra med relationer. T.ex likes och vänrelationer. Dessa noder och relationer bildar tillsammans ett nätverk av data. När vi hämtar data från detta nätverk så "traverserar" vi från en nod längst med dess relationer ut till nya noder. T.ex när du öppnar min Facebookprofil så vill du även se vilka mina vänner är. Då traverserar du från min "profil"nod via "vän"relationer ut till andra "profil"noder. På så sätt kan du sätta mig i ett sammanhang och lära dig mer om mig.

För att traversera i detta nätverk av information i realtid krävs det att datan är sparad på ett smart sätt. Traditionella databaser, så kallade relationsdatabaser, använder tabeller med rader och kolumner för att spara data. Men den modellen passar dåligt för den här typen av "kopplad data". Det är här som grafdatabaser kommer in i bilden. Ordet graf är det matematiska namnet på ett nätverk av noder och relationer och det är så grafdatabaser sparar all data, som noder och relationer. Det gör det enkelt att "traversera" i informationsnätverket.

MEN! Även grafdatabaser stöter på problem när noder har extremt många relationer. Speciellt svårt är det om vi endast är intresserade av en liten del av alla "grann"noder. T.ex kanske det endast är de tio nyaste vännerna som skall visas på profilsidan. Men för att hitta de tio senaste så måste vi först plocka fram alla vänner och sedan sortera dem efter när vänrelationen skapades. Om jag har 200 vänner så innebär det en otrolig mängd onödigt arbete och försämrade svarstider för databasen.

I detta arbete visar vi hur vi kan komma runt problemet genom att hålla ett sorterat register, eller index, över de mest kritiska traverseringarna. Svarstiderna kan på så vis bli mellan 10100 gånger snabbare. Tricket för att nå denna förbättring är att vi använder oss av ett B+ träd, en datastruktur som länge har använts för indexering i databaser. Vad som är nytt är hur vi använder den för att indexera en hel substruktur av grafen istället för bara en enda nod.

Genom att minska svarstiden för täta grafdatabaser öppnar vi upp möjligheter att utföra mer avancerad analys av data i realtid. Vilket är mycket värdefullt eftersom data i sig själv inte är värt något förrän vi kan tolka den.