Eindhoven University of Technology

MASTER

MAGPIE - a Maintainable Graph Pattern Indexing Engine
towards a versatile path index for the industrial graph database

de Jong, N.

Award date:
2019

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

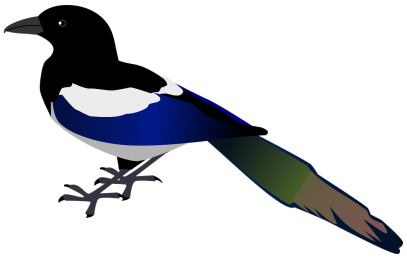
- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 14. Sep. 2026

MAGPIE - a Maintainable Graph Pattern Indexing Engine

Towards a Versatile Path Index for the Industrial Graph Database



Master's Thesis
Database Group
Faculty of Mathematics and Computer Science
Eindhoven University of Technology

NIELS DE JONG

Academic Supervisors:
dr. G.H.L. Fletcher
dr. N. Yakovets

Industry Supervisor:
A. Averbuch

Examination Committee:
dr. G.H.L. Fletcher
dr. N. Yakovets
dr. D. Fahland

Eindhoven, March 2019

Niels de Jong: *MAGPIE - a Maintainable Graph Pattern Indexing Engine,* Towards a Versatile Path Index for the Industrial Graph Database, © March 2019

ABSTRACT

Even though the benefits of using a path index for evaluating pattern matching queries have been proven empirically by years of research, no industrial graph database has adopted the path index as a core feature. One of the reasons for this is the fact that the *maintainability* of existing path indexing methods is not considered by previous work. Thus, these indexing techniques are unusable for general purpose database systems, which deal with changing data. This report summarizes the design of MAGPIE, the first Maintainable Graph Pattern Indexing Engine, which is tailored to be used in the industrial graph database.

First, we identify four requirements that a path index must meet such that it can be used in an industrial database. Given these requirements, we perform a survey of existing structural indexing techniques, and select the k-path index [24] as our candidate for extension. We extend k-path indexing with two novel maintenance algorithms, of which one exploits the B+ tree prefix-search operation to achieve efficient index updates by avoiding expensive graph traversals. We also introduce the concept of batching index-update translations, as to speed up the processing of multi-update transactions.

To motivate developing a fast maintenance technique for a path index, we need insights on the frequency of updates that can create or delete paths in a graph. To investigate this, we perform the first-ever large-scale analysis of Cypher logs from industrial applications. In addition to recording the number of updates, we also investigate the structural properties of the query graphs, as well as the frequency of sub-patterns in a single log. The latter will also act as evidence that long patterns exist frequently for some applications, and can be evaluated faster by a path index. This, to our knowledge, has not been shown by previous research.

To test the capabilities of MAGPIE, we have implemented the new maintenance techniques as an integrated part of the Neo4j graph database. In our experimental study, we find that (in the best case) our new maintenance technique allows for a number of magnitudes speed-up as opposed to traversal-based updates. From our study on Cypher logs, we find that some log instances contain over 99% queries that can generate path index updates. In addition, we find that many of the logs contain long frequent sub-patterns that can be evaluated faster by using a path index.

ACKNOWLEDGEMENTS

This thesis has been a collaborative effort between the Database Group at Eindhoven University of Technology and Neo4j in Malmö, Sweden. I want to thank both parties for providing the time and resources to be able to come to a successful completion of this thesis. Working on this project has been a unique experience and an excellent opportunity to work with both academia and industry.

I would like to thank Dr. George Fletcher and Dr. Nikolay Yakovets for their active involvement and frequent feedback during the project. I would also like to thank Dr. Fahland for serving on my assessment committee. I very much appreciate the time and effort put in by Dr. Fletcher and Dr. Yakovets for meeting frequently and pointing me in the right direction throughout the past months. It has been a great experience working with the Database Group, who have guided me with humour and expertise through my internship and thesis period. A special thanks also to Max Sumrall for taking the time to share his thoughts on this study.

I would also like to thank Alex Averbuch for his everlasting patience and help with benchmarking the index implementation, Satia Herfert for his help with the Neo4j parser, as well as Anton Persson for his time explaining the Neo4j indexing infrastructure. To everyone in the Neo4j Cypher team who has been actively involved in my project, I would also like to give thanks for their time and effort. Last, a special thanks to everyone else at Neo4j for providing a warm home in the cold Swedish winter.

Last but not least I would like to thank my parents, my family, and friends for their support throughout my thesis and my entire education. I will never forget my time in the TU/e Common Room, where I have shared amazing lunches, dinners and an uncountable number of coffees over the past years with people from all corners of the world.

CONTENTS

1 INTRODUCTION 1
1.1 Use-Cases for Path Indexes . . . 1
1.1.1 Query Evaluation . . . 2
1.1.2 Graph Algorithms . . . 2
1.1.3 (Regular Path) Query Planning . . . 2
1.1.4 Special Operators . . . 3
1.1.5 Views . . . 3
1.2 Context . . . 3
1.3 Research Question . . . 3
1.4 Contributions . . . 4
1.5 Overview of Contents . . . 4
2 PRELIMINARIES 7
2.1 Semi-Structured Data Models . . . 7
2.2 Property Graph Model . . . 8
2.3 Paths & Label-Paths . . . 8
2.4 Pattern Matching Queries . . . 8
2.5 Indexing . . . 9
2.6 Notation . . . 10
3 QUERY LOG ANALYSIS 11
3.1 Problem Definition . . . 11
3.2 Related Work . . . 12
3.3 Data . . . 13
3.4 Query Log Analysis Method . . . 14
3.4.1 Parsing . . . 14
3.4.2 Frequency of Updates . . . 14
3.4.3 Query Graph Structure . . . 15
3.4.4 Frequent Sub-Patterns . . . 15
3.5 Results . . . 16
3.5.1 Frequency of Updates . . . 17
3.5.2 Query Graph Structure . . . 17
3.5.3 Frequent Sub-Patterns . . . 18
3.6 Conclusion & Future Work . . . 20
4 INDEXING GRAPH STRUCTURE 23
4.1 Structures to Index . . . 23
4.1.1 Indexing Paths . . . 23
4.1.2 Indexing (Frequent) Sub-Graphs . . . 23
4.1.3 Indexing Trees . . . 24
4.1.4 Choice of Structure to Index . . . 24
4.2 Requirements for a Path Index . . . 24
4.3 A Survey of Path Indexing Techniques . . . 25
4.3.1 DataGuides . . . 25
4.3.2 The T-Index . . . 26

4.3.3 The D(k) Index . . . 27
4.3.4 The GRIN Index . . . 28
4.3.5 Language-Based Indexing . . . 28
4.3.6 *k*-Path Indexing . . . 29
4.3.7 Comparison of Techniques . . . 30
5 MAINTAINING A PATH INDEX 33
5.1 Types of Updates . . . 34
5.2 Handling Updates . . . 35
5.3 Traversal-Based Translation (TBT) . . . 36
5.4 Inverted Index Translation (IIT) . . . 37
5.5 Self-Maintaining Translation (SMT) . . . 38
5.6 Batching Indexing Maintenance . . . 40
5.6.1 Motivation . . . 40
5.6.2 Batching Method . . . 41
5.7 Comparison of Techniques . . . 42
6 IMPLEMENTATION 45
6.1 Index Design . . . 45
6.1.1 The GB+ Tree . . . 45
6.1.2 Key Design . . . 46
6.1.3 Multiple Trees for Multiple Paths . . . 46
6.2 Index Maintenance . . . 46
6.2.1 Overview . . . 47
6.2.2 Traversal-Based Translation . . . 48
6.2.3 Self-Maintaining Translation . . . 48
6.2.4 Concatenating Paths . . . 48
6.2.5 Writing Updates to Index . . . 49
7 EXPERIMENTAL EVALUATION 51
7.1 Experiments on Synthetic Data . . . 51
7.1.1 Graph Layout . . . 51
7.1.2 Results . . . 53
7.1.3 Summary . . . 54
7.2 Experiments on Real Data . . . 55
7.2.1 Experiment Design . . . 55
7.2.2 Results . . . 56
8 CONCLUSION 59
8.1 Summary . . . 59
8.2 Future Work . . . 60
A APPENDIX A: QUERY LOG DATA 63
A.1 Types of Queries . . . 64
A.2 Query Shapes . . . 66
A.3 Frequent Pattern Analysis . . . 68
B APPENDIX B: EXPERIMENT RESULTS 71
B.1 Synthetic Data . . . 71
B.2 Real Data . . . 72

BIBLIOGRAPHY 73

LIST OF FIGURES

Figure 1 Popular data models as ordered by their degree of structure. . . . 7
Figure 2 Distribution of the number of write queries for the 66 query logs. . . . 17
Figure 3 Distribution of read/write queries and custom procedure calls for the 66 query logs. . . . 18
Figure 4 The 66 query logs and their distributions of query graph size. . . . 19
Figure 5 The 66 query logs and their percentages of unbounded variable length query graphs. . . . 19
Figure 6 The top 10 frequent patterns for log $D30$. . . . 20
Figure 7 Left: An example dataset with information about a university department. Right: A corresponding DataGuide. . . . 25
Figure 8 A simple node and edge-labeled graph. . . . 29
Figure 9 An example graph layout resulting in path index updates. The dashed line represents the added/deleted edge $a \xrightarrow{X} b$. . . . 36
Figure 10 The different sub-indexes needed to efficiently maintain an index on paths of the shape $A \xrightarrow{X} B \xrightarrow{Y} C \xrightarrow{Z} D$. . . . 39
Figure 11 Adding three edges to an existing graph. Here, the dotted lines represent new edges. . . . 40
Figure 12 Three updates in a single transaction that imply an ordering on visiting edges for path index updates. . . . 41
Figure 13 Two updates in a single transaction while maintaining $A \xrightarrow{X} B \xrightarrow{Y} C$ and $C \xrightarrow{Y} B \xrightarrow{X} A$. No edge-at-a-time processing of update translation is possible. . . . 41
Figure 14 A high-level overview of the index maintenance implementation. . . . 47
Figure 15 An edge addition with label `[:X]` that creates four new indexed paths. The traversal-based update method will perform no unnecessary work. . . . 52
Figure 16 An edge addition with label `[:X]` that creates 1 new indexed path. The traversal-based update method will do extra work, as it runs into nodes with label `(:E)`. . . . 53

Figure 17 A flame graph for the experiment performing SMT on a tree with 10000 paths, where no unnecessary traversals can be skipped. 54
Figure 18 A high-level label schema for the Geospecies Knowledge Base RDF graph. 56

LIST OF TABLES

Table 1 An overview of the notation used in this thesis. 10
Table 2 Keys for a k-path index where $k = 2$, for the graph in Figure 8. 30
Table 3 An overview of the path indexing techniques in Section 4.3. 31
Table 4 The number of indexed sub-paths needed to update a path index using self-maintaining translations. 39
Table 5 Overview of the maintenance technique for path updates. 42
Table 6 Results of the maintenance experiments on synthetic data. 71
Table 7 Results of the maintenance experiments on real data. The reported mean and error are over the entire workload (100,000 queries). 72

1 INTRODUCTION

Increased use of semi-structured and unstructured data have made non-relational data stores an increasingly popular means of data storage and retrieval. In particular, graph databases have seen a surge in popularity over the last decade, largely due to their natural network-like representation and data flexibility. Pattern matching query languages such as *Cypher* are often used to query graph databases. These languages inherently require a large number of joins to be executed on the data store, which can get computationally expensive, even given the fact that native graph databases have been shown to outperform relational databases on long, pattern-matching queries [27]. To tackle this problem, we explore structural indexing as a means of accelerating this type of queries.

Ever since the first database systems were developed, indexing has been widely used for speeding up data retrieval. Traditionally, index structures for relational databases rely on a predefined schema that defines the structure of the tables in the database. [16] Semi-structured data such as graphs, however, have no explicit schema, and therefore indexing techniques are not easily translated from a relational context. [25] For this reason, indexing structural information of a graph is a non-trivial task, especially in an industrial database (OLTP system) where data changes dynamically over time. Even though a variety of structural indexing techniques have been proposed in the literature, very limited attention is given to maintenance after addition and deletion of graph elements, consequently making existing techniques not applicable to industrial graph databases. To our knowledge, as of yet, no industrial graph database exists that supports structural indexing as a core functionality, even though relevant research on this type of indexing is very promising.

1.1 USE-CASES FOR PATH INDEXES

A database index speeds up data retrieval by keeping a redundant copy of data in a specialized data structure, such as a B+ tree. Indexing has already been investigated extensively for labels and properties of nodes and edges, and is available in most industrial graph database systems, such as Neo4j. To motivate the choice of adding structural indexing as a core functionality to a graph database, we highlight five possible use cases in the following sections.

1.1.1 *Query Evaluation*

A structural index can aid in evaluation of long, complex pattern matching queries by avoiding random disk IO operations. Given that these type of queries are at the core of graph analytics, this type of index could make a large difference in evaluation time. One of the problems of this category is the following query: Given a node n in a graph, efficiently identify the set of paths p intersecting that node that match a given pattern. An example Cypher query expressing this is as follows:

```
MATCH p=(a:A)-[:X]->(:B)-[:Y]->(:C)
WHERE a.id = n
RETURN p
```

These type of operations can be made more efficient by using a structural index - given that our index allows for fast retrieval of all paths originated from node a. More generally, we might also be interested in just finding all `(:A)-[:X]->(:B)-[:Y]->(:C)` patterns. The indexing technique we select in Section 4.3.6 supports both of these operations.

1.1.2 *Graph Algorithms*

A structural index can additionally be used to speed up graph algorithms involving many traversals. As an example, consider the PageRank algorithm. Computing the PageRank of a node in a network is dependent on the node's neighbours, which can efficiently be done by the use of a structural (path) index. For a higher-level PageRank, the PageRank could even be computed by looking at a more extended neighbourhood of a node, e.g., finding its neighbourhood with nodes up to a distance two away.

1.1.3 *(Regular Path) Query Planning*

In [7], Fletcher et al. illustrate how a path index can be used to generate better execution plans for Regular Path Queries (RPQs). In the experimental evaluation of [7], it is demonstrated that a k-path index of all paths up to length $k = 2$ may already generate physical plans that speed up execution by a number of magnitudes. In fact, there is evidence that query planning for Cypher queries can also be improved by the use of a path index: given that we have exact counts on the number of paths of a given length two pattern, we can produce significantly better cardinality estimations as opposed to having simple graph statistics. [13]

1.1.4 *Special Operators*

Some specialized graph database operations can benefit significantly from the presence of a path index. The triadic closure operator, as frequently used in recommendation engines, originated from the following principle from sociology: if A has a connection with B, and B is connected with C, then A is likely also connected with C. The Neo4j database contains a special operator for these type of operations, due to their frequency of appearing in query workloads. The fact that this operator exists is a strong motivation that these types of queries appear frequently, and are thus interesting candidates for optimization. A path index can aid in this type of query, as a triadic closure can be modeled as a path.

1.1.5 *Views*

A path index is a possible implementation of a materialized view for chain pattern matching queries. In Chapter 3 we find that for some query logs, over 90% of the pattern matching queries have a chain shape, thus covering a significant part of some query workloads. The problem of maintaining and storing the view is then translated to the problem of maintaining and storing the index.

1.2 CONTEXT

From previous research, we find that the use of a structural index can greatly speed up query evaluation, in some cases evaluation is sped up by a factor greater than 1000. [26] [24] The path index has, however, never been implemented in an industrial graph database as a core feature. We find the main reason for this is that no existing technique has directly addressed *maintenance* - an essential requirement for a database that deals with changing graphs. In addition, there is limited empirical evidence supporting the occurrence of updates, as well as the fact that (long) paths appear frequently in query logs. In this thesis, both of these issues will be addressed.

1.3 RESEARCH QUESTION

In our study, we aim to provide an answer to the following research question: How can we design a structural path index suited for the industrial graph database, such that it can efficiently be maintained? Before we tackle this problem, we also investigate:

- Is there a need for efficient maintenance in industrial graph applications?

- How often can a path index be utilized for a given application, and which paths are worth indexing?

To motivate that our solution can be used in an industrial database, we implement MAGPIE as an integrated part of Neo4j. With this, we can experimentally investigate the performance of the index in terms of maintainability. To investigate the frequency of updates and frequent query patterns, we investigate 66 Cypher query logs as used by six of Neo4j's customers.

1.4 CONTRIBUTIONS

This report provides a thorough investigation of the maintenance of path indexes and the use of path indexing in general. We make three main contributions by our research:

- We perform a theoretical and empirical evaluation of two novel methods to efficiently translate graph updates to index updates. One of the methods, which we call *Self-Maintaining Translation* (SMT), allows us to avoid expensive graph traversals by using shorter paths as present in the index.
- We provide an implementation of the new approach as an integrated part of the Neo4j database engine, supporting the claim that MAGPIE can be implemented in an industrial graph database. Neo4j is, at the time of writing this paper, the world's leading graph database according to the DB-engines ranking [1], and is an excellent candidate for a demonstration of an implementation for a maintainable path indexing technique. Our experimental evaluation shows that SMT can provide a large speed-up as opposed to a straightforward traversal-based translation.
- By analyzing a wide variety of query logs, we discover opportunities for using a path index by investigating the structure of Cypher queries. In addition, we investigate the frequency of queries that can trigger a path index update. We developed an analysis script that parses and processes the Cypher queries as present in the Neo4j log files in order to extract the query graphs, and performs post-processing on these query graphs to discover frequent patterns. To our knowledge, we are the first to perform such an analysis on Cypher queries.

1.5 OVERVIEW OF CONTENTS

Chapter 2 summarizes preliminary knowledge for the remainder of the thesis, as well as provides an overview of the used notation and

1 https://db-engines.com/en/ranking/graph%20dbms

abbreviations. Chapter 3 describes our query workload analysis on Cypher logs, as well as discusses the results in relation to the aforementioned research questions. Chapter 4 contains a discussion of different structural indexing techniques, as well as a survey of the state-of-the-art in path indexing. Chapter 5 goes into detail on the challenges that are introduced when building a maintainable path index and introduces a number of novel solutions for this type of maintenance. Next, Chapter 6 describes implementation specifics for embedded our new techniques into the Neo4j codebase. Chapter 7 describes a number of experiments to empirically evaluate the new maintenance techniques, and finally, Chapter 8 presents concluding remarks and future work.

2 PRELIMINARIES

First, we provide a short introduction to the landscape of databases for semi-structured and unstructured data and show where graphs fit into the bigger picture of these types of data stores. Then, we provide the reader with an overview of concepts inherent to graph databases, including nodes, edges, and paths. We introduce the concept of *indexing* to accelerate query evaluation, and discuss existing techniques for indexing graph structures. Lastly, we provide a concise overview of the notation used in this report.

2.1 SEMI-STRUCTURED DATA MODELS

Since the introduction of relational databases in the early 1970's [6], many types of data stores have been proposed to fit the needs for different shapes of data. Non-relational data stores (often dubbed NoSQL stores) come in many forms and can be sorted in order of structural strictness as in Figure 1. In their 2011 paper, Strauch et al. provide a similar ordering, even though graphs are not present in their overview [22].

In general, less structured data provides more flexibility, but limits optimizations derived from the data's structural properties. Thus, structural indexing is a problem that is fundamentally different based on the structure of the underlying data store. Even though a large variety of indexing techniques exist for hierarchical data such as XML and JSON stores, indexing techniques for graphs are far less common, as discussed in Chapter 4.

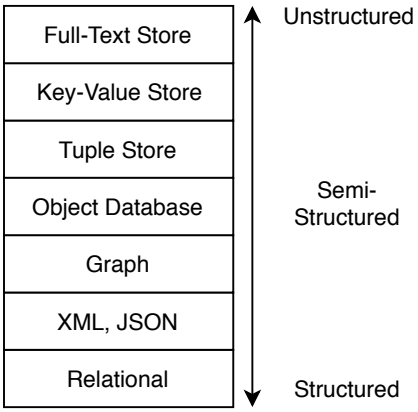


Figure 1: Popular data models as ordered by their degree of structure.

2.2 PROPERTY GRAPH MODEL

Even though a variety of data models exist for graph-shaped data, the (labeled) property graph model is most frequently by graph databases [21]. This model, which allows for the augmentation of nodes and edges with a set of properties, has proven effective for many industrial applications. The property graph model as used by Neo4j has a specific set of semantics that should be taken into consideration when developing optimization techniques for graph querying. The most important constraints are the following:

- A node may have zero or more labels.
- An edge must have exactly one label.
- A path, as defined in Neo4j, cannot contain the same edge twice. It can, however, contain the same node twice.

2.3 PATHS & LABEL-PATHS

A path in a graph is defined as a fixed-length ordered sequence of nodes and edges. We define the length of a path to be the number of edges contained in a path. A label path is a sequence of node labels and edge labels, similar to a pattern. Given a graph, a label path will have a set of matching paths - paths that have the exact same sequence of labels. If any of the labels in a label path are unspecified, this will match with any node (or edge) regardless of its labels. In addition, a node will be matched in a label path even if the label path does not specify all of the labels. That is, if a node has two labels, A and B, the node can appear in label paths that contain either A or B.

2.4 PATTERN MATCHING QUERIES

As of the time of writing this thesis, no standardized language exists for querying graphs. The declarative Cypher, SparQL and PGQL languages are popular due to their SQL-like syntax and are therefore used frequently by commercial graph databases. Section 1.1.1 provides an example of a pattern matching query in Cypher, where a pattern is expressed using a label path. The set of patterns inside a single query can also be represented in a graph format - this is referred to as a *query graph*. Query graphs can also contain *variable length patterns*, which are denoted by the Kleene star (*). As an example, the Cypher pattern `(:A)-[:X*]-(:B)` will match all paths from A nodes to B nodes that are a sequence of X edges, regardless of its length.

2.5 INDEXING

Database indexing is a widely used technique to speed up the evaluation of queries in databases. By having fast access to a redundant copy of some of the data, expensive read-operations to the underlying data structure can be avoided or minimized. Indexing has been investigated for many different shapes of data and ranges from simple, single property indexes to structural indexing as discussed in Chapter 4. Specialized data structures, such as the B+ tree, have been extensively researched to fit common operations such as reading and writing the index. Hash-based indexing has also been investigated thoroughly, and continues to be used, even for graph patterns [20]. Recently, with the rise in popularity of AI and deep-learning solutions, self-learned index structures have also been proposed [15]. In our study, we do not go in-depth on the design of index structures themselves, but rather on how to effectively use existing techniques to achieve efficient maintenance.

2.6 NOTATION

The table below summarizes the different notations as used in the remainder of this report. In some cases, mostly relating to the Cypher log analysis, we will used Cypher notation. A concise overview of the Cypher notation can be found on the Neo4j website.[1]

Graph	$G(V, E)$	A graph is defined as a set of nodes V, and a set of edges E.
Nodes	$a \in V$	Nodes are denoted by the lowercase letters a, b, c... a is sometimes refered to as the ID of a node.
Edges	$a \xrightarrow{x} b \in E$	an edge x from node a to node b. Edges are denoted by the lowercase letters x, y, z, q...
Path	$a \xrightarrow{x} b \xrightarrow{y} c$	A path is a sequence of connected nodes and edges.
Labels	$\mathcal{L}(a) = A$	the label of node a is A. Labels are denoted by uppercase letters.
Label-Paths	$A \xrightarrow{X} B$	$A \xrightarrow{X} B$ refers to the set of all edges $a \xrightarrow{x} b$, where $\mathcal{L}(a) = A$, $\mathcal{L}(b) = B$ and $\mathcal{L}(x) = X$.
Path Shape	shape($a \xrightarrow{x} b$) $= A \xrightarrow{X} B$	The shape of a individual path p is the label path with matching labels to the given path.
Var. Length Label-Path	$A \dashrightarrow B$	Denotes all paths from a node with label A to a node with label B of unspecified length.
Set of Labels	$\mathcal{L}(V)$ and $\mathcal{L}(E)$	These denote the sets of all node and edge labels, respectively.

Table 1: An overview of the notation used in this thesis.

1 https://neo4j.com/docs/cypher-refcard/current/

3 QUERY LOG ANALYSIS

In recent years, an increasing amount of work has been done on the topic of understanding the structure of semi-structured data. Understanding how people *interact* with semi-structured data stores, on the other hand, has only recently received attention, even though it has a lot to offer. Considering our topic of path indexing, understanding logs is especially important. Developing elaborate path indexing techniques only becomes useful if they can be used to evaluate queries that contain these paths. In this chapter we answer exactly this question: can a path index be of use for evaluating real-world database queries? If so, how can we determine which paths appear frequently in queries, and are they the most valuable to index? As we are also interested in maintaining these indexes, it is also useful to gain insights into the frequency of update queries that can potentially create new paths.

To answer these questions, we performed an extensive study on real-world Cypher query logs from Neo4j customers. Section 3.1 goes in more detail on the exact artefacts we want to extract from these logs. Section 3.2 summarizes related work on the analysis of graph queries. Section 3.3 elaborates on the investigated query logs. Section 3.4 describes the procedure of reading, parsing and processing the query logs to extract the interesting information. Sections 3.5 and 3.6 summarize the findings from the logs and makes some concluding remarks.

3.1 PROBLEM DEFINITION

Even though research on the analysis of graph queries exists, no such analysis has ever been done on Cypher queries in particular. In addition, little is known about the occurrence of frequent subpatterns in query graphs, which play an essential part in selecting the right paths to index. In the following sections, we describe the results of an analysis of 66 datasets used by six of Neo4j's customers. By analyzing over five million queries, we answer the following questions:

- What query graph shapes are common for real-world Cypher queries? How often do large query graphs appear?
- How can we extract frequent patterns from Cypher queries? What can these patterns tell us about which paths are interesting to index?

- How often do updates occur which may create or delete paths?

We perform this analysis by means of a script that handles the end-to-end processing from query logs to summary data about the structure of the workload. The script is written in Java and uses Scala to hook into the Neo4j parser. The source code for this script is available open-source.[1]

3.2 RELATED WORK

Understanding query logs for graph-shaped data has seen relatively little attention in academia. Most of the current research has been focused on SparQL logs, likely due to a large amount of open data containing SparQL queries. In this section, we summarize relevant findings from these studies and indicate what information is missing for our purposes.

To our knowledge, Möller et al. were the first to analyze large SparQL query logs. [17]. Their 2010 study focused mainly on the agents used to do the queries, as well as the frequency of queries over time. Some attention is given to the number of triple patterns as present in the SparQL queries, which prove to be highly variable depending on the dataset. Still, for all investigated datasets, more than 99% had less than four triple patterns per query (with two of the datasets even having as many as 99% single-pattern queries). The shapes and structure of the queries are not investigated in [17], and neither is the frequency of updates.

A 2011 study by Gallego et al. went in-depth on some of the missing information from the Möller study. This study focused mainly on understanding the frequencies of joins and patterns in real-world SparQL queries [2]. Gallego found that most of the 3 million analyzed DBPedia queries are relatively short and do not contain many triple patterns. The two query workloads examined contain 66.41% and 97.25% queries with a single pattern respectively. Interestingly, almost all (>99.9%) of the observed queries with more than two joins have an (almost) star-shaped structure. Star-patterns are common for the RDF data structure, as all properties of a node are expressed using triples, which materialize as edges in the graph. In the property graph model, these type of star-shaped structures can be replaced by property fields on the nodes and edges in the graph, and will likely appear less often.

In the same year, Picalausa and Vansummeren also performed a study on SparQL queries from a DBPedia endpoint. [19] The types of queries

1 https://github.com/nielsdejong/neo4j-log-parser

(SELECT, ASK, CONSTRUCT and DESCRIBE), the number of operators and structural properties of the data were thoroughly documented by Picalausa and Vansummeren, as well as the number of queries that contain combinations of operators. Even though the size and shape of the queries was not directly addressed, the time complexity of evaluating the queries was induced from structural properties of the query. Picalausa and Vansummeren found that many of the analyzed queries can be evaluated in polynomial time.

In 2017, Bonifati et al. perform an analytical study on roughly 180 million SparQL queries. [3] In their study, a number of open data sets from DBPedia, WikiData and a number of other open data sources, were analyzed. The study contains a summary of a wide range of query graph properties such as the number of triples per query, the distribution of operators and even a detailed structural analysis. Bonifati et al. summarized the number of chain queries, trees, star queries as well as the treewidth of the analyzed queries. As opposed to the study by Gallego, Bonifati et al. found many more complex (long) SparQL queries, as well as discovered that for the analyzed datasets more than 90% of the queries are chain-shaped. Unfortunately, the analysis did not contain an investigation into frequent sub-patterns, nor the frequency of updates.

Even though previous research provides an insight into the shape and size of graph queries, we find the following to be missing:

1. No research has been done into Cypher queries as used in an industrial database setting. Given that SparQL and Cypher are designed for different data models, the shapes of Cypher queries may be entirely different.

2. Little is known about the frequency of updates, and its relation to the number of read-only queries.

3. To learn about frequently occurring label-paths, we need to examine frequent query patterns, which have not been investigated by previous research.

3.3 DATA

Our experiments were performed on 66 query logs from different Neo4j customers, totalling 5,249,471 queries. The datasets range not only in size (from 24 to 1.5 million queries) but also in contents, as the data comes from a variety of industries: telecom, consultancy, travel, banking, computer hardware and analytics. For confidentiality reasons, we do not disclose any details about the contents of the logs. For the remainder of this chapter, we refer to the logs by a number ranging from D1 to D66.

3.4 QUERY LOG ANALYSIS METHOD

This section describes the design of a query log analyzer for Cypher queries that extract structural information from query graphs. We summarize the entire log analysis procedure, from parsing Cypher to extracting patterns, and ultimately mining frequent sub-patterns. The results are discussed in the next section.

3.4.1 *Parsing*

An essential part of our query analysis is examining the structure of the Cypher query graphs. As the Neo4j logs only contain the Cypher string, the parser is required to parse the queries in the logs to extract query patterns from the graph. In the remainder of this chapter, query patterns are referred to as a collection of label-path *blocks*, which have the following shape: **(***a***:**$\mathcal{L}1$**)-[***t***:**$\mathcal{T}$******i***..***j***]->(***b***:**$\mathcal{L}2$**)**. Here, a and b are node identifiers, t is an edge identifier, $\mathcal{L}1$ and $\mathcal{L}2$ are sets of allowed node labels, $\mathcal{T}$ is a set of allowed edge labels, and i and j are the minimum and maximum number of $\mathcal{T}$ edges to match, respectively.

For confidentiality reasons, we anonymize the names of nodes and edges in the query graph, as well as label names.

The task of the query parser is to convert a Cypher string into a set of blocks, which can be fit together to form a query graph. As Cypher supports nested queries, we are required to iterate over all nested queries and add these to the set of blocks. Our implementation hooks into the Neo4j parsing module to accomplish this, as to avoid having to write a standalone parser. The output from the Neo4j parsing module is used by the remainder of the analysis.

3.4.2 *Frequency of Updates*

To determine whether a query performs updates, we check whether the Cypher query contains any of these statements: 'CREATE', 'SET', 'MERGE', 'DELETE' or 'REMOVE'. In the Cypher language, these are the only five statements that are able to modify nodes, edges, and their labels.

- 'CREATE' and 'DELETE' statements should always be considered updates that can create or delete new paths for indexed label-paths.
- 'SET' and 'REMOVE' statements can sometimes be considered updates that influence paths in the index. In Cypher, these statements can either modify labels or properties, but only label modifications can cause updates in our path index, as we do not index paths with properties. By utilizing the Cypher parser, we extract only the 'SET' and 'REMOVE' statements that modify labels.

- 'MERGE' is a special case of an updating statement, as it either matches a pattern or creates the pattern if it does not exist. As Neo4j's query logs do not capture whether any writes have been made to the graph store, we cannot be sure whether MERGE statements have caused any updates. In the discussed of our results, we categorize these types of queries as an update query.

Neo4j's Cypher supports another statement that can potentially perform updates to the graph. The 'CALL' clause can be used to call custom procedures as an extension to Neo4j. These procedures can perform any operation on the graph, including updates. In most cases, however, this is unknown. In our study, we do not consider 'CALL' queries as updating queries.

Missing Patterns

In some cases, the logs do not contain the patterns that have been traversed during query execution. If no edge labels are specified in a query pattern, all edges connected to a certain node need to be traversed, regardless of label, which will not appear in the logs. Another example of these missing patterns is caused by the DETACH DELETE operator in Neo4j. This operator deletes a node along with the edges connected to it. The labels of these edges are not specified in the Cypher query, and therefore not present in the query logs, even though they will have to be traversed in order to process the update.

3.4.3 *Query Graph Structure*

As a first step towards structural analysis, we examine the shape of the query graphs similar to the work of Bonifati et al. [3] Here, we only consider the read and write queries, and ignore queries that perform a custom procedure. First, we transform the parsed blocks into a graph format, such that we can identify which blocks are connected. Using a simple implementation of a breadth-first search, we can easily determine whether a query graph is a chain, has loops, is a tree, or even a forest. If a query graph contains a Kleene star (*), we mark it as an unbounded variable length pattern in our analysis (**var_len**). Appendix A.2 provides a summary of the query graph structure of each of the analyzed logs.

3.4.4 *Frequent Sub-Patterns*

To identify which exact patterns could be indexed to speed up query evaluation, (linear) frequent sub-patterns are extracted from the query graphs. Simply looking at the size of query graphs is insufficient to give insights on where path indexes are applicable, instead, we should identify which patterns are frequent enough to be stored to

Sub-pattern extraction can be accelerated, for example by the use of the Apriori algorithm.

index. Thus, we extract the frequent sub-patterns from the query graphs in a query log, as well as the number of times they appear. Given that the number of sub-patterns scales exponentially in the size of the query graph, we extract only the patterns up to length five, especially since some of the queries contain unbounded variable length patterns. Let V_q be the set of nodes in the query graph q. Let $\mathcal{P}_0$ be a set containing a single zero-length path. That is, for any path P, concatenation with $\mathcal{P}_0$ returns the same path: $\mathcal{P}_0 \cup P = P$. Then, let $F(P)$ be a mapping from sub-patterns to a frequency value. Algorithm 1 contains the procedure used for frequent sub-pattern extraction up to length 5.

Algorithm 1 Extract linear frequent sub-patterns up to length n

```
1: P ← ∅
2: F ← ∅
3: for each v ∈ V_g do
4:     EXTRACT_FSP(P_0, v, 5)
   return F
5: function EXTRACT_FSP(P_in, v, n)
6:     if n = 0 then
7:         return P_in
8:     P_new ← ∅
9:     for each b ∈ BlocksConnectedTo(v) do
10:        for each P_in ∈ P_in do
11:            P ← P_in ∪ b
12:            F(P) ← F(P) + 1
13:            P_new ← P_new ∪ P
14:        return P_in ∪ EXTRACT_FSP(P_new, b.other, n − 1)
```

In Algorithm 1, subroutine `BlocksConnectedTo(`v`)` returns all edges connected to a node v in the query graph, including permutations with unspecified node labels. Consider a query graph with a single Cypher pattern `(a:A)-[x:X]->(b:B)`. Then, `BlocksConnectedTo(`a`)` will return not only `(:A)-[:X]->(:B)`, but also patterns `()-[:X]->(:B)`, `(:A)-[:X]->()` and `()-[:X]->()`. These are patterns that could also be indexed, and can also appear by themselves in query graphs, as query graphs do not always contain labeled nodes. The implementation of Algorithm 1 also considers variable length patterns as a special case, but this was omitted for ease of understanding.

3.5 RESULTS

This section summarizes the results gathered by the log analysis as described in the previous section. Here, we highlight a number of interesting cases in relation to the research questions as specified in

Section 3.1. The complete list of aggregated results can be found in Appendix A.

3.5.1 *Frequency of Updates*

Appendix A.1 lists for each of the logs the number of read-only, write and custom procedure queries. We additionally record the number of unique queries as observed in the logs. Figure 2 contains a plot of the distribution of read/write queries for the analyzed logs. Even though 59 out of 66 logs contain only 0-10% write queries, there are also logs with a large number of mutating queries. An example of this is *D*15, which contains 99.8% write queries. There appears to be no direct relation between write queries and custom procedure calls. Figure 3 shows the distribution of normal read/write Cypher queries and the number of custom procedure calls. Log *D*4 is an example of a log that is largely dominated by custom procedure calls (99.9%) and contains very few actual Cypher queries. Custom procedure calls are common throughout all logs, 10 out of 66 logs contain more than 50% of these queries. In terms of the number of unique queries, there are also large differences. *D*3 has 425811 queries of which 127 unique, and *D*15 has 918556 queries, of which almost all queries are unique. Our log analysis thus shows highly heterogeneous data. To able to handle applications that perform many data mutations, we must ensure a highly efficient solution to deal with regular update queries.

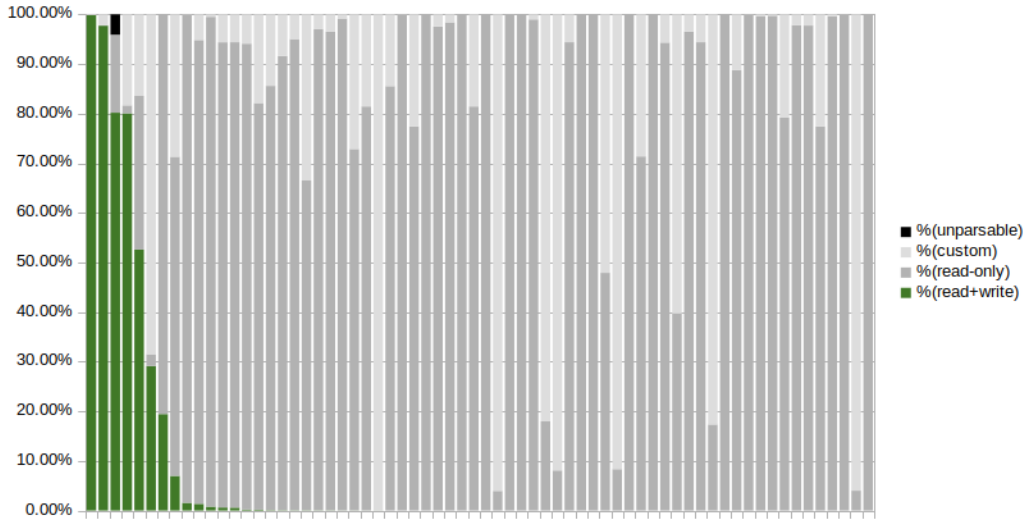


Figure 2: Distribution of the number of write queries for the 66 query logs.

3.5.2 *Query Graph Structure*

From Appendix A.2 we find the same degree of data heterogeneity in query shapes and in query types. Figure 4 contains a plot the distribu-

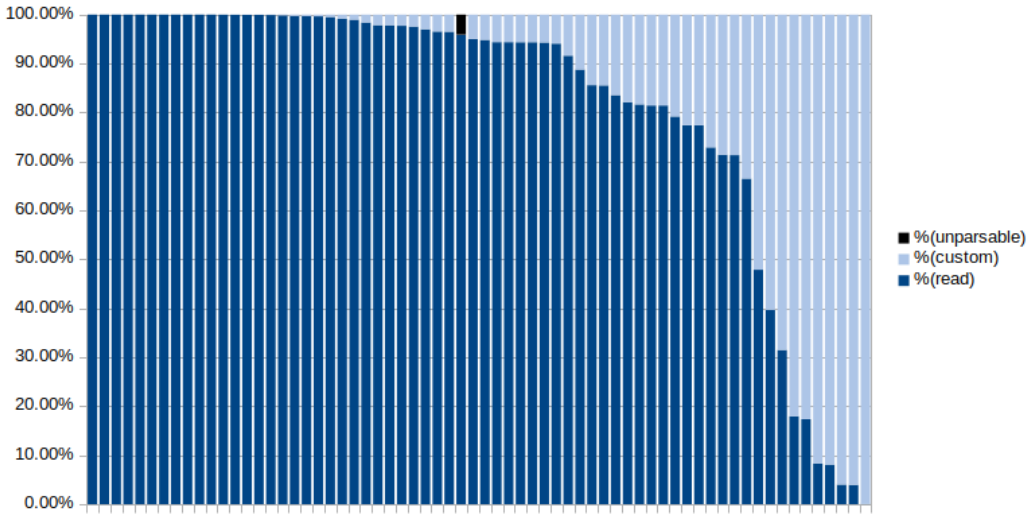


Figure 3: Distribution of read/write queries and custom procedure calls for the 66 query logs.

tions in query graph size for the 66 datasets. From the appendix, we find that logs *D*15 and *D*26 contain almost exclusively query graphs of size zero and one, and are thus less interesting candidates for using a path index. Log *D*12 however could be especially interesting to apply a path index to, as it contains over 95% query graphs with an unbounded variable length pattern. Log *D*57 contains 30% queries with an unbounded variable length pattern, but also contains 7% updating queries. Even though a path index could be of use to speed up query evaluation for the queries in this log, index maintenance can also be expensive, due to the high frequency of updates. This log is a good example where a path index can potentially be useful, but only if it can be maintained efficiently. *D*57 also proves that applications with frequent updates and applications with complex query graphs are not necessarily disjoint.

Figure 5 shows that variable length patterns are common in the query logs, with three of the logs containing exclusively queries with an unspecified-length pattern. These types of patterns can benefit from a specific shape of path index, where one or zero node labels are specified. Consider a query pattern `(:A)-[:X*]->(:B)`, which can be partially evaluated by reading an index on path `(:A)-[:X]->()-[:X]->()-[:X]->()-[:X]->()`, bootstrapping the evaluation. In our frequent sub-pattern analysis, we should therefore also consider this shape of patterns.

3.5.3 *Frequent Sub-Patterns*

To discover which patterns could be valuable to index, we use the frequent patterns as extracted by Algorithm 1. The result of this algo-

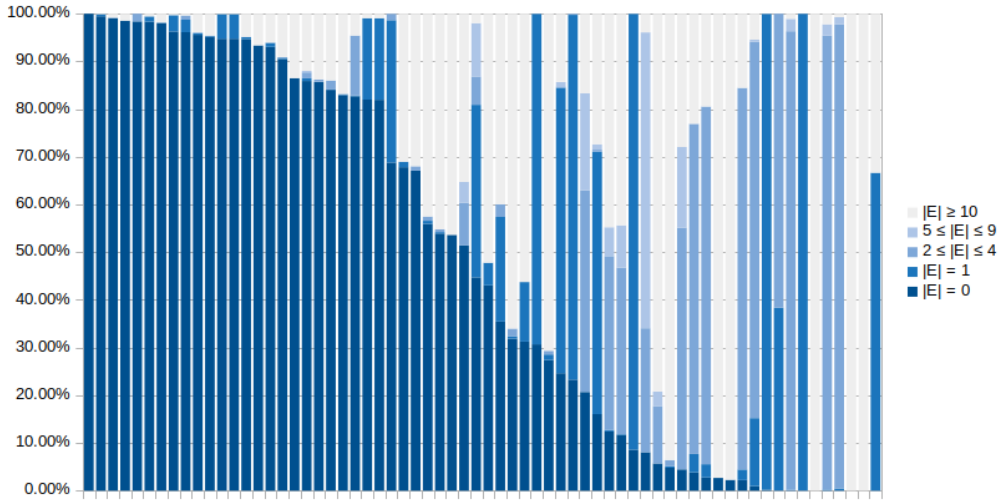


Figure 4: The 66 query logs and their distributions of query graph size.

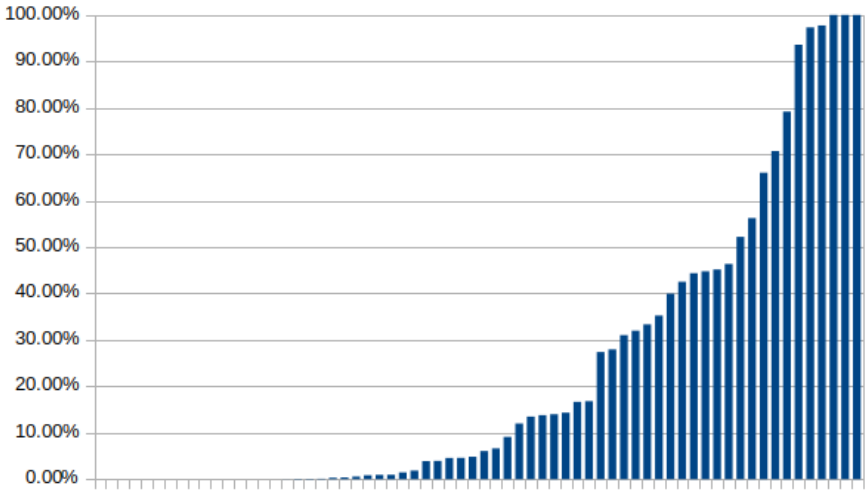


Figure 5: The 66 query logs and their percentages of unbounded variable length query graphs.

rithm is the appearing sub-patterns together with their frequencies. The frequency can be seen as a basic predictor for which paths to index. If a subpattern is observed frequently, the indexed path can be frequently used to aid query evaluation. From the study by Sumrall [24], we find that indexes on length one paths can already speed up query evaluation, but we are mainly interested in longer paths, which provide the more interesting use-cases for path indexing. For this reason, we also distinguish between different lengths of frequent patterns.

The relation between the executing times of (parts of) frequent patterns is discussed as future work.

Figure 6 contains the top 10 frequent patterns for log $D30$ as produced by the frequent sub-pattern extraction algorithm. For this log, with a total of 3579 Cypher queries, we find pattern `(:K)-[:K]->(:J)`

appears 3666 times - on average, more than once per query. This pattern, as well as `()-[:K]-(:J)-[:K]->(:J)` are thus great candidates for indexing. Appendix A.3 summarizes, for each log, the number of

pattern	length	count
(:K)-[:K]->(:J)	1	3666
(:)-[:K]->(:J)	1	3666
(:K)-[:K]->(:)	1	3666
(:)-[:K]->(:)	1	3666
(:)-[:K]-(:)	1	3280
(:)-[:K]-(:J)-[:K]->(:J)	2	1640
(:)-[:K]->(:J)-[:K]-(:)	2	1640
(:)-[:K]-(:J)-[:K]->(:)	2	1640
(:K)-[:K]->(:J)-[:K]-(:)	2	1640
(:)-[:K]-(:)-[:K]->(:J)	2	1640

Figure 6: The top 10 frequent patterns for log *D*30.

frequent patterns that appear often. For this, we count the number of frequent patterns whose counts are more than 10% of the total number of queries in the log. Even though this is an arbitrary number, it acts as a heuristic to determine how useful an index on a sub-pattern can be. In the future, given that we know the relative speed-up produced by indexing a pattern, a better heuristic can be used. By distinguishing between different lengths of frequent patterns, we can also investigate whether long frequent patterns also appear regularly.

From Appendix A.3 we learn that many of the logs contain frequent patterns that can be indexed. 32 out of the considered logs contain frequent patterns of length two or greater that appear more than 10% of the total number of queries. 26 of the logs contain even a length 5 frequent pattern that is an interesting indexing candidate. It is important to take into account that these counts include permutations of patterns that contain unspecified node labels. The pattern `(:J)-[:P]->(:O)-[:R]->(:N)-[:O]->(:M)-[:S]->(:P)-[:Q]->(:I)` is contained in *D*30 for a total of 458 times, but there are also 127 different permutations of this pattern which do not specify node labels that could be indexed. The total count as reflected in the appendix is thus 128.

3.6 CONCLUSION & FUTURE WORK

From the results as discussed in Section 3.5 we conclude that there is strong evidence that path indexing can be applied to a number of industrial applications. Due to the fact that there are many patterns of length two or greater that appear frequently in many of the analyzed logs, there are many queries which could have been evaluated faster given that these patterns were indexed. In addition, we have discovered that seven of our logs contain over 10% mutating queries.

For these applications, a path index can only be effectively used if we can maintain it with minimal overhead. Thus, an investigation into maintenance methods can be of significant value.

As future work, we would like to investigate how the length and size of a query graph correspond to the execution time of the query. In the analyzed logs, we only know the total execution time of a query, thus it is hard to say which joins are most expensive. Given that we can identify the most expensive patterns to evaluate, along with a good estimate of maintenance cost for each indexed path, we can more accurately predict which of the paths are affordable to index.

4

INDEXING GRAPH STRUCTURE

The increased use of graph-shaped data has resulted in the birth of a completely new class of indexing techniques, vastly different from those used in relational databases or those used for tree-shaped data. With the introduction of these new data stores and query languages, existing techniques cannot directly be translated to a graph context [25]. As querying graphs is done by specifying *patterns*, our index should naturally resemble and be designed to support pattern-matching queries. In this chapter, we explore the landscape of structural indexing methods for semi-structured data.

Section 4.1 introduces three types of structural indexing, as well as provides a motivation for the choice of indexing paths as opposed to other substructures. In Section 4.2, we define four requirements that a path-index must fulfil to be able to be used in an industrial database. Next, Section 4.3 provides an analysis of existing literature on path indexing techniques with respect to these four requirements.

4.1 STRUCTURES TO INDEX

First, we briefly summarize the types of graph structures that can be indexed. We refrain from the discussion of basic indexes, such as node and edge indexes, as well as indexes for (composite) properties. Instead, we focus on indexing structural properties of the graph.

4.1.1 *Indexing Paths*

The simplest, non-trivial graph structure to index is a path. Recall that a path is defined as an ordered sequence of directed edges. Given that we want to index paths in a directed graph, there should be a possibility to index patterns with reversed edges, e.g. $A \xrightarrow{X} B \xleftarrow{Y} C$. The largest advantage of indexing only paths is that they are easier to process and store due to their linear shape. Most of the existing literature on structural indexes focuses on paths.

Patterns with reversed edges ($A \xrightarrow{X} B \xleftarrow{Y} C$) also appear frequently in the Cypher query logs investigated in Chapter 3.

4.1.2 *Indexing (Frequent) Sub-Graphs*

One may find that just storing simple paths cannot express enough information about the real structure of a graph - detailed structural information is lost [28]. In graphs that contain many sub-graphs of the same shape, such as chemical or biological networks, it may be more

effective to index frequently occurring sub-graphs instead of paths - consider indexing 100 different sub-graphs as opposed to millions of different paths. Indexing sub-graphs, however, introduces many challenges. Given a graph, which sub-graphs do we index? What is the largest substructure size we want to store? How can we store these sub-graphs in an efficient manner? Unfortunately, these questions are only very briefly discussed in the current state-of-the-art on structural indexing.

4.1.3 *Indexing Trees*

A third option for structural indexing is the indexing of tree-shaped patterns. In [29], Zhang et al. claim that tree structure indexes can preserve almost as much structural information as a sub-graph index. Yet, to our knowledge, very limited research on tree indexes for graph-shaped data exists.

4.1.4 *Choice of Structure to Index*

We select paths as the best candidate for structural graph indexing for the following reasons:

1. Path indexes have been most extensively studied as opposed to other graph structures, and a variety of different algorithms for the construction of path indexes exist. In addition, an implementation of the *k*-path index has already been shown to decrease query evaluation time by a number of magnitudes [24].
2. The linear structure of paths allows for the use of efficient storage techniques on well-established data-structures such as the B+ tree.
3. Maintenance of more complex graph structures may get complicated quickly. In [28], it is mentioned that frequent substructures may need to be recomputed after a number of graph updates, given that the internal graph structure has possibly changed too much. For dynamic graph databases, this could potentially pose a problem in terms of efficiency.

4.2 REQUIREMENTS FOR A PATH INDEX

We define four requirements that an index must meet for it be usable in an industrial database setting. These requirements are as follows:

- **Construction Time** - We should be able to quickly construct the index from an existing graph.
- **Memory Usage**- The index should not consume too much memory when stored on disk.

- **Maintenance Time** - The index should be efficiently maintained as our graph changes over time.
- **Query Evaluation Time** - The index should speed up the evaluation of pattern matching queries significantly.

In our survey of existing path indexing techniques, these metrics play an important role when selecting our indexing technique. Although not measurable, a fifth requirement often set by industrial databases is to be ACID compliant - the data in the index should still be valid after unexpected errors. In Chapter 6, we show how we integrate our path index into a generation-aware B+ tree implementation that ensures ACID compliance.

4.3 A SURVEY OF PATH INDEXING TECHNIQUES

We now provide an overview of path indexing techniques for graph-shaped data. In this section, we summarize the contents of a wide variety of indexing methods by selecting techniques that are fundamentally different in underlying mechanisms and supporting data structures. To provide an overview of the complete landscape of existing work, we list techniques based on five concepts: state automata, bi-simulation, center-nodes, language expressivity, and paths. Ultimately, we provide a concise summary of the discussed techniques, highlighting the strengths and weaknesses of each approach.

4.3.1 *DataGuides*

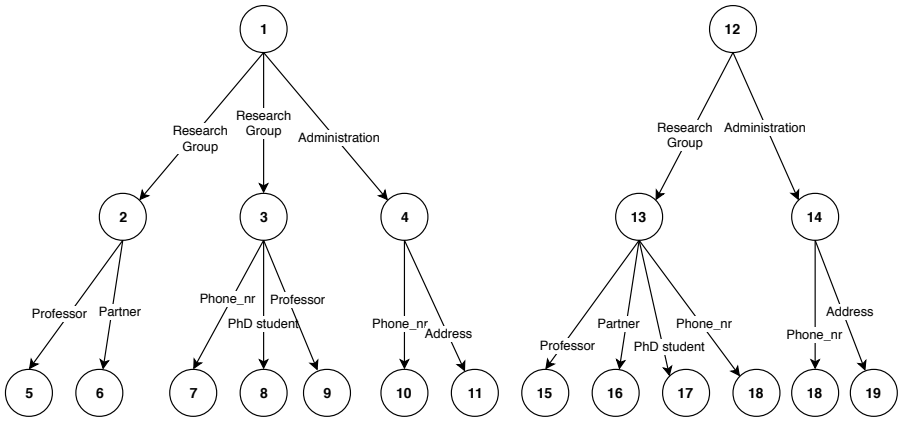


Figure 7: Left: An example dataset with information about a university department. Right: A corresponding DataGuide.

To our knowledge, the use of DataGuides [11] as proposed by Goldman & Widom is one of the first indexing techniques that captures information about semi-structural data. In [11], a *DataGuide* is defined as a structural summary of the label-paths appearing in a tree or

graph dataset, which can be used for a variety of use-cases, including a path index. The DataGuide is based on the concept of a state automaton - it embeds the allowed transitions (edges) that can be made from a given state (set of nodes). Figure 7 provides an example of a DataGuide for a database containing information about a department of a university. By the definition of a DataGuide, we find that a path in the original graph always corresponds to a path in the DataGuide on the right. Note, however, this DataGuide could be simplified, e.g., node 13 and 14 could be merged into one node. This results in node 7 and 10 in the original graph being indistinguishable by the DataGuide, and as such this new DataGuide would not be usable as a path index. In [11], the original DataGuide as in Figure 7 is called a *Strong DataGuide*, which implies a one-to-one mapping between nodes in the source graph and nodes in the DataGuide. For this very reason, only strong DataGuides can be used as a path index.

In [11], Goldman & Widom show that strong DataGuides can be efficiently constructed from a given tree. For most of the experiments conducted on constructing DataGuides, the size of the DataGuide is significantly smaller than the original dataset, but Goldman & Widom do note that the size of a DataGuide for non tree-shaped data is worst-case exponential in the size of the graph. Maintenance of DataGuides is also addressed in detail, and a number of algorithms are proposed to update a DataGuide on edge insertion and deletion. Goldman & Widom find that maintenance for tree-structured data can be done with reasonable speed, but graph-shaped data may require a large number of sub-DataGuides to be re-evaluated. No experimental evaluation of incremental updates is performed in [11], but it appears that the provided algorithms are not easily applied to (dense) graph data.

4.3.2 *The T-Index*

The 1999 paper "Index Structures for Path Expressions" by Milo and Suciu introduces a path index for regular path queries (RPQs) on rooted graphs, called the T-Index (Template-Index) [16]. The T-Index is a generalization of two other indexing structures as described in [16], called the 1-Index and the 2-Index, which are constructed in a similar fashion. The T-Index, in particular, allows for construction of an index of all RPQs of a certain shape (or template). For example, a T-Index can be created for all paths that contain a person owning a house: $() \dashrightarrow person \xrightarrow{owns} house \dashrightarrow ()$. Construction of the 1-, 2- or T-index is based on the following idea: Given that two nodes in a graph are in the same equivalence class with respect to a query language L, these nodes are indistinguishable by that language. Concretely, this implies that these nodes are either both present in the result of a query, or both missing in the result. Thus, if we can effi-

ciently compute the equivalence classes of nodes with respect to L, we can use these as an index. The T-index is constructed by computing an approximation of a language equivalence relation of a graph, called a *bi-simulation*. Given that two nodes a and b are in the approximation relation $\approx$, we know that $a \approx b \implies a \equiv b$, where $\equiv$ is the equivalence relation of L. Constructing a bi-simulation is shown to be possible in $O(|E| \log |V|)$ time, as opposed to constructing an equivalence relation, which is a *PSPACE* problem [18].

The storage space required for a T-index is not discussed in [16], this naturally depends on the degree of the generality of the index templates. Maintenance is very briefly mentioned, a possible algorithm for incremental T-index updates is referenced in the conclusion of [16]. This algorithm and its supporting data structure is described in [4], and involves constructing a mapping from every node in a graph to all the paths that originate from that node, similar to an *inverted index*. This additional data structure will naturally also need to be stored and maintained. Finally, deletion is only possible in linear time for directed acyclic graphs.

4.3.3 *The D(k) Index*

The D(k) Index as proposed by Chen et al. is another simulation-based indexing technique, designed to allow for efficient updates on graph changes. [5] In addition, the D(k) index is adaptive - the index can be adapted based on query workloads. The foundations of the D(k) index and the T-index as discussed in the previous section are similar, both are generalizations of the 1-index as described in [16]. Essential to the D(k) index is the concept of *local similarity*, a local similarity value is assigned to each node in the index (simulation) graph. Local similarities are determined by graph updates and determine for each node in the index graph, the maximum length k of paths starting from that node that can be retrieved using the index. To ensure that infrequently queried paths are no longer stored in the index, the local similarity values slowly decrease by a process called demoting. Similarly, the promoting process increases the local similarity values for the index nodes. The authors of [5] suggest that both promoting and demoting should be done periodically, but leave the exact specifications of when to execute these processes as future work.

As the D(k) index is based on the A(k) index [14], the size of the D(k) index can never exceed the size of the data graph. Construction can be done efficiently in $O(|E| \cdot k)$ time, where k is the maximum length of paths in the index. Index updates are empirically shown to be significantly faster than those on the A(k) index. The evaluation time of queries is also much lower than the A(k) index, even

though the updating process could potentially influence evaluation performance. Unfortunately, the algorithms for the construction and updating heavily rely on the fact that our graph is rooted, which is not the case for our data model as described in Section 2.2.

4.3.4 *The GRIN Index*

While not directly storing paths, the GRIN Index as described by Udrea et al. in [26] can be used to evaluate path queries in RDF data. The GRIN Index is based on the concept of centrality of nodes - a number of nodes are chosen as "center" nodes, and these are associated with the local neighbourhood of nodes appearing close to the center. Then, given that our query evaluation engine runs into a center node, we can efficiently retrieve its direct neighbourhood nodes and their distances from the center node.

Udrea et al. provide a concise summary of the experimental performance of the GRIN index with respect to construction time, size and query evaluation time. Theoretical bounds on size are not discussed, however. An empirical study shows that GRIN performs better than three other RDF indexes, which were at the time state-of-the-art. Unfortunately, maintenance is not discussed, but a number of issues will likely come into play when considering maintenance. The most straightforward is that the centrality of a node can change over time. In [26], an inter-cluster distance measure is used to define centrality, but this distance is expensive to re-compute on every graph update.

4.3.5 *Language-Based Indexing*

A 2018 survey by Fletcher and Theobald [10] provides an overview of structural indexing based on query language expressivity. Given that two entities in a graph are indistinguishable given a query language L, they are *structurally equivalent* in terms of L. Thus, the elements are either both present in the result of a query, or both absent. An index based on this principle can thus group these structurally equivalent elements together, and return them efficiently.

There is a close link between indistinguishability in terms of a query language and bi-simulation [9], and it has been shown that for many fragments of a language, indistinguishability is decidable in polynomial time. In 2009, Fletcher et al. published a study that links the concepts of language indistinguishability and indexing XML data, by investigating for which XPath expressions structural indexes can be utilized. [8] This paper provides a theoretical ground for the P(k)-index, which based on partitions of upward paths in an XML tree. The investigation of a data structure that supports efficient operations

for this type of indexing is left as future work. Even though there are many promising opportunities for applying language-based indexing to practical applications, structural (graph) indexing has only received limited attention in academia.

4.3.6 *k-Path Indexing*

The *k*-path index, as described by Sumrall in [24], is to our knowledge the first path index designed for the property graph model. By using a B+ tree with keys that contain a label-path and a path identifier, path queries can be evaluated a number of magnitudes faster than by directly using the graph store. Figure 8 shows an example property graph, and Table 2 contains the index keys for some paths of length two. The design of the index keys in [24] is based on the fact that a path can be identified by (at least) the ids of the edges in the path. This key design allows for *prefix-searching* the B+ tree, given that the keys are sorted. Then, without specifying the entire key, we can retrieve all paths of the shape $A \xrightarrow{X} B \xrightarrow{Y} C$.

The empirical study in [24] has shown that the *k*-path index can greatly decrease query evaluation time. In addition, there is a possibility for compression of the keys in the B+ tree, reducing the total memory consumed by the *k*-path index. Experiments on workload-driven indexing have also been performed in [24], which additionally reduce memory consumption. The B+ tree data structure allows for the efficient construction of the *k*-path index due to its bulk-loading capabilities. As discussed later, the basic *k*-path indexing method as described by Sumrall is not suited for incremental maintenance, especially for updates that generate many (long) paths.

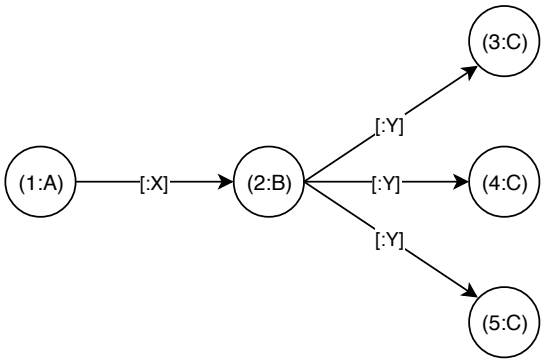


Figure 8: A simple node and edge-labeled graph.

N_LBL	E_LBL	N_LBL	E_LBL	N_LBL	NODE_ID	NODE_ID	NODE_ID
A	*X*	*B*	*Y*	*C*	1	2	3
A	*X*	*B*	*Y*	*C*	1	2	4
A	*X*	*B*	*Y*	*C*	1	2	5

Table 2: Keys for a k-path index where $k = 2$, for the graph in Figure 8.

4.3.7 *Comparison of Techniques*

We have investigated six structural indexing methods for graph-shaped data, based on five conceptual ideas: state automata, bi-simulation, center-nodes, language expressivity, and paths. Table 3 summarizes how each technique performs on the metrics of construction time, size of the index ($|\mathcal{I}|$), ease of maintenance and query evaluation speed. We make the following observations when comparing these techniques:

- The original DataGuides was mainly designed to be used for tree-shaped object databases, and most of the operations can only be done on graphs with no or minimal cycles. Even though graphs are supported, operations on a DataGuide can take exponential time if an update triggers rebuilding of a large number of sub-DataGuides.
- Language-based structural indexing shows promising opportunities, however, there is limited empirical evidence of its effectivity. In addition, further investigations into efficient data structures that support this type of indexing are yet to be done.
- The T-Index and the D(k) Index are designed to be used on rooted graphs, where all nodes are reachable from a designated root node. In our data model as described in Section 2.2, we cannot make this assumption.
- Upper bounds for the retrieval of data from an index are not discussed in detail in any of the surveyed techniques. However, we know that DataGuides, the T-Index and the D(k) index all have a graph as the internal index representation. Thus, in the worst case, the entire index graph needs to be examined, we denote this as $|\mathcal{I}|$. The GRIN Index is shaped like a balanced binary search tree, which requires $\log |\mathcal{I}|$ comparisons. The k-path index is based on the B+ tree, which requires $\log_B |\mathcal{I}|$ comparisons, where B is the order of the B+ tree.

In conclusion, from Table 3, we learn that none of the existing techniques are suited for maintainability of non-rooted graphs. For all of the discussed approaches, graph updates result in a traversal of the stored data graph which can potentially be very expensive, due to

random IO operations both in the index representation and the original graph. The k path index, however, has the advantage of utilizing the venerable B+ tree data structure, as well as shows promising results in terms of the experimental evaluation. As the full k-path index is very expensive to construct and store, we choose the workload-driven k-path index as a viable candidate for an industrial path index.

Language-based structural indexing was omitted from Table 3, due to limited information about the time complexity of the indexing operations.

Table 3: An overview of the path indexing techniques in Section 4.3.

Technique	**Construction**	**Size**	**Updates**	**Retrieval**
DataGuides[a]	$EXP(\|E\| + \|V\|)$	$EXP(\|E\| + \|V\|)$	$EXP(\|E\| + \|V\|)$	$O(\|\mathcal{I}\|)$
T-Index[a]	$O(\|E\| \log \|V\|)$	Unknown	$O(\|E\| + \|V\|)$[b]	$O(\|\mathcal{I}\|)$
D(k) Index[a]	$O(\|E\| \cdot k)$	$O(\|E\| + \|V\|)$	$O(\|E\| + \|V\|)$	$O(\|\mathcal{I}\|)$
GRIN Index	$O(\|E\|^4 \log \|E\|)$	Unknown	Unknown	$\log_2 \|\mathcal{I}\|$
k-Path Index[c]	$O(\|E\|^k)$	$O(\|E\|^k)$	$O(\|E\|^{k-1})$	$\log_B \|\mathcal{I}\|$

[a] DataGuides, the T- and D(k)-index are designed for rooted graphs.
[b] The T-Index can be updated in linear time if there is an additional inverted index mapping all nodes and edges to their paths. (In practice, such an inverted index is extremely hard to maintain and store)
[c] This corresponds to the complete, non workload-driven k-path index.

5 MAINTAINING A PATH INDEX

As opposed to simple node, edge, or property indexes, path indexes are not trivial to maintain. Even in a small graph, a single edge can already be part of thousands of paths, even when only indexing paths up to length two. This results in a serious maintenance problem:

EXAMPLE Consider maintaining an index on all length two paths. Let n be a node with degree D. Given that we add a new edge e to n, we need to add D new paths in our index, as we need to examine all combinations of e and existing edges attached to n. Then, a large number of entries in the index structure need to be updated.

This example introduces one of the issues related to maintenance of path indexes: apart from the cost of updating the index structure, there is another costly factor: identifying which paths are created by the addition/deletion of graph elements. We call this *translation* from graph updates to path updates. Then, we can divide the index maintenance process into two parts:

1. Translating node and edge updates to path updates.
2. Updating index data structures, i.e., writing to a B+ tree.

Even though these two steps are inherent to all path indexing methods, this chapter will introduce a number of maintenance techniques for the k-path index in particular. Recall from Chapter 4 that few indexing techniques consider handling graph updates as a design choice, as such, research into index maintenance is limited. In this chapter, we do not consider ourselves with index-implementation specific updates, e.g., the balancing of a B+ tree. Instead, we focus on how to translate graph modifications into index updates efficiently.

In Section 5.1 we list the types of updates of interest for path index maintenance. Section 5.2 dissects the maintenance process in detail. In Section 5.3, we describe the most straightforward method of converting graph updates to paths, as well as the problems introduced by this approach. Section 5.4 introduces the inverted index as a means of speeding up a translation, but we find that this method also introduces some problems. In Section 5.5, we show how we perform translations efficiently by means of *Self-Maintaining Translation* which exploit a property of the B+ tree data structure. Next, in Section 5.6, we discuss batching graph updates together, such that translations

can be performed more efficiently for transactions with multiple updates. In fact, we show that batching is required to be able to handle multi-update transactions. Section 5.7 recaps the advantages and disadvantages of each method.

5.1 TYPES OF UPDATES

Recall from the property graph model as defined in Section 2.2 that a node may have 0 to n labels, and an edge always has a single label. We distinguish between six types of updates to this type of graph:

1. Node Insertion - node insertions do not need to be considered when maintaining the path index. Naturally, when a new node is added to the graph, it is completely disconnected from any other component. Only after edges are inserted that connect to a node, new paths are created in the graph.

2. Node Deletion - node deletion does additionally not influence the paths in a graph. This is due to the fact that a connected node can only be deleted together with its edges.

3. Node Label Changes - A node label change on node n could possibly result in new paths behind created in the graph. On such a change, we must:
   - For all removed labels, delete the existing paths going through n from the index.
   - For all added labels, index the new paths going through n.

4. Edge Insertion - Consider that we are adding an edge x from node a to b. Then, we should add to the index all new paths created that contain this edge x. Thus, starting from nodes a and b, we explore the graph to identify the paths x is contained in.

5. Edge Deletion - deleting an edge x will result in all paths containing x to be removed from an index.

6. Edge Label Changes - An edge label change could result in both additions and deletions. Considering the property graph model in 2.2 which states that an edge has exactly one label, an edge label change can be seen as deletion of an edge followed by an addition.

In Section 5.2 we focus on the two most complex type of updates, edge additions and deletions. Later, we show that the algorithm to handle node label changes is a simplified version of the edge-update algorithm.

5.2 HANDLING UPDATES

Considering edge insertions and deletions, the following must be done to maintain a path index when inserting/deleting edge $a \xrightarrow{X} b$:

1. Find the set of paths $\mathcal{P}_{in}$ ending in a.
2. Find the set of paths $\mathcal{P}_{out}$ starting from b.
3. Concatenate all paths in $\mathcal{P}_{in}$ with $a \xrightarrow{X} b$ and $\mathcal{P}_{out}$, and insert/remove these paths from the index data structure.

A similar method can be applied to node label changes. In this case, we simply find all paths $\mathcal{P}$ originating from node n and join them with each other. Concatenating incoming and outgoing paths is still required for processing deletions, this is the most efficient method to identify the paths a deleted edge is contained in. When we are dealing with a k-length path index, the paths incoming into a and b are naturally length-bounded by length $k-1$.

Algorithm 2 describes the translation procedure in a simplified manner. This algorithm uses subroutines `PathsComingInto` and `PathsComingOutOf` that return a list of paths coming into or out of a specified node. Concatenation is denoted by using the 'union' symbol ($\cup$). An actual implementation of this algorithm should also take into account that we only get the right paths incoming and outgoing into a certain edge, as we are only interested in paths that are stored in our index. In addition, nodes with multiple labels should be considered.

Algorithm 2 Translate a single edge update on $a \xrightarrow{X} b$ to path updates

1: $\mathcal{P} \leftarrow \{\}$
2: $\mathcal{P}_{in} \leftarrow$ `PathsComingInto(`$a \xrightarrow{X} b$`)`
3: $\mathcal{P}_{out} \leftarrow$ `PathsComingOutOf(`$a \xrightarrow{X} b$`)`
4: **for each** $P_{in} \in \mathcal{P}_{in}$ **do**
5: **for each** $P_{out} \in \mathcal{P}_{out}$ **do**
6: $\mathcal{P} \leftarrow \mathcal{P} \cup (P_{in} \cup (a \xrightarrow{X} b) \cup P_{out})$
7: **return** $\mathcal{P}$

We now list three techniques for finding all paths starting/ending from a given node - different alternatives for the `PathsComingInto` subroutine. We propose three methods to solve this problem: TBT (Traversal-Based Translation), IIT (Inverted Index Translation), and SMT (Self-Maintaining Translation), each come with their individual benefits and costs. In addition, we discuss the possibility of batching index updates to increase maintenance speed.

5.3 TRAVERSAL-BASED TRANSLATION (TBT)

In Sections 5.3, 5.4 and 5.5 we consider adding/removing a single edge. Section 5.6 discusses multi-update transactions in more detail.

First, we discuss index maintenance directly accessing the graph store. That is, whenever an edge is added to/removed from the graph, we perform a local search to find the paths the edge is contained in. Consider the following example:

EXAMPLE Consider maintaining a forward k-path index on a graph, where we add/remove an edge $a \xrightarrow{X} b$. Figure 9 shows a part of this graph. After this update, we are required to modify our index, such that all new paths (of length $\leq k$) of the shape $() \dashrightarrow a \xrightarrow{X} b \dashrightarrow ()$ are added, or old paths are deleted. This can be accomplished by running a local search of depth (at most) $k-1$ starting from node a, storing the set of paths $\mathcal{P}_{in}$ ending in node a. We call this traversal-based translation (TBT). A similar search should be done from node b, storing the set of paths $\mathcal{P}_{out}$. Then, we concatenate every path in $\mathcal{P}_{in}$ with edge $a \xrightarrow{X} b$ and then all paths in $\mathcal{P}_{out}$. On edge insertion, these paths should be added to the index, and on deletion, these paths should be removed from the index. Naturally, we ensure that the total length of the newly generated paths is at most k. In practice, this means an incoming path p_{in} into a and an outgoing path p_{out} from b are only concatenated when $|p_{in}| + |p_{out}| + 1 \leq k$.

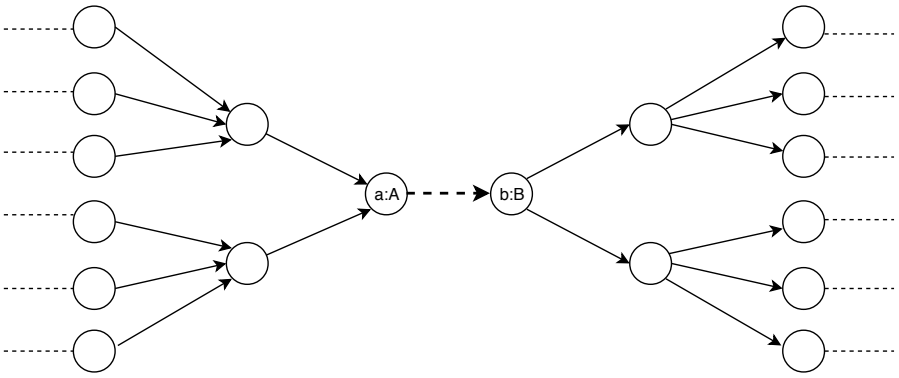


Figure 9: An example graph layout resulting in path index updates. The dashed line represents the added/deleted edge $a \xrightarrow{X} b$.

The largest advantage of TBT is that updating the index can be done only using only the data in the graph, no additional data is required. In addition, if the local-search is implemented through a pruning BFS, we can avoid some unnecessary traversals. The following main issues can, however, be identified with this approach:

- The number of edges traversed in the graph by TBT is exponential in k. More precisely, $\mathcal{O}(|D_{max}|^{k-1})$, where D_{max} is the maximum degree in the graph. Additionally, there are unavoidable unnecessary traversals that need to be performed. In the

experimental evaluation of Chapter 7, we provide an example with these kinds of unnecessary traversals.

- Running a local-search (BFS) on the graph may result in many random IO operations to read graph information from disk, which can get expensive.
- For each insertion and deletion in a transaction, we require two calls to the `PathsComingInto` subroutine, even though several new edges can be added to a single node in a transaction. This problem is further discussed in Section 5.6, where we discuss batching index updates and reusing the output from the subroutines.

In the next two sections, we describe how local-searches can be avoided by using either an inverted index or by using existing information stored in the k-path index.

5.4 INVERTED INDEX TRANSLATION (IIT)

A possible means of avoiding local searches in the graph is by using an inverted index. An inverted index can be seen as a reversed k-path index: instead of mapping paths to nodes, we store for every node the paths this node is a part of. This idea is very similar to the proposed solution to T-Index maintenance, as discussed in Section 4.3.2. With this inverted index, it is simple to find all paths ending in or starting from a given node. Nevertheless, such an inverted index introduces a number of new issues:

- The inverted index needs to be stored. Consider again that a single node can be a part of thousands of paths, even for small k-values. Then, storing an inverted index on all nodes in the graph will take up a large amount of memory.
- The inverted index needs to be constructed and maintained, creating a huge overhead for such a supporting data structure.

Alternatively, we can also consider a partial inverted index. It might be interesting to index only nodes with high degrees or nodes that appear in many paths. Then, for these nodes, we can use the inverted index, and for the lower-degree nodes, use a simple local search. Such an inverted index can even be maintained based on query workloads - we only construct an inverted index for nodes that are queried frequently. Yet, this is very expensive to maintain, defeating the purpose of efficient path index maintenance. Using an inverted index as a separate supporting data structure is therefore not a good choice.

5.5 SELF-MAINTAINING TRANSLATION (SMT)

A third solution to finding paths ending in a node is using the information present in the index itself. This method, which we call self-maintaining translation, borrows its name from a self-maintaining view. A self-maintaining view, as described in [12], is a materialized view that does not require access to the underlying data tables in order to stay updated. Instead, given a query and its own contents, it updates the data in the view. Similarly, given that we need to update an index entry for a path P of length i, can we use the fact that we already store P's sub-paths of length $i-1$ in our index? Then, which sub-paths of P should be present in the index? Consider the previous example again:

EXAMPLE Again, we update the index entry for all paths of the shape $() \dashrightarrow a \xrightarrow{X} b \dashrightarrow ()$, when a new edge $a \xrightarrow{X} b$ is added to a graph. Rather than exploring the local neighbourhoods of depth (at most) $k-1$ starting from node a and b, we consult our index to find the set of paths $\mathcal{P}_{in}$ that end in a and b. This requires the index to contain all sub-paths of the lengths at most $k-1$. Similarly, we concatenate every path in $\mathcal{P}_{in}$ with $a \xrightarrow{X} b$ and then $\mathcal{P}_{out}$, and add/remove these to/from the index. Using this method allows the index to be maintained *without* accessing the graph store, avoiding a local search like a BFS. Note this is only possible due to the fact that we can prefix-search the B+ tree the paths are contained in, by specifying only part of the search key.

To elaborate on the method of using sub-paths to accomplish self-maintaining translations, we introduce a concrete example. Consider adding a new edge `[y:Y]`. between node `(b:B)` and node `(c:C)`. We need to perform the following steps when updating an index on `(:A)-[:X]->(:B)-[:Y]->(:C)-[:Z]->(:D)`:

1. $P_{in} \leftarrow$ read from the index all `(:B)<-[:X]-(:A)` coming from node b.
2. $P_{in} \leftarrow$ read from the index all `(:C)-[:Z]->(:D)` coming from node c.
3. Add to `(:A)-[:X]->(:B)-[:Y]->(:C)-[:Z]->(:D)`, for all $p_1 \in P_{in}$ and $p_2 \in P_{out}$: $p_1 \cup y \cup p_2$.

Figure 10 illustrates for `(:A)-[:X]->(:B)-[:Y]->(:C)-[:Z]->(:D)` which shorter indexed paths need to be available for reading sub-paths using SMT. Note that, depending on which edge in the path we are updating, different sub-indexes will need to be searched. Table 4 summarizes the total number of indexed sub-paths needed to update

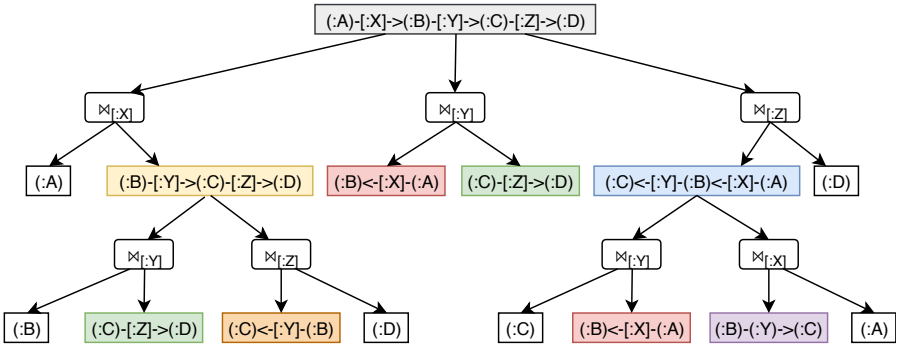


Figure 10: The different sub-indexes needed to efficiently maintain an index on paths of the shape $A \xrightarrow{X} B \xrightarrow{Y} C \xrightarrow{Z} D$.

a k-path index of a given length. From this table, we make the observation that we need $n(n-1)$ sub-indexes to be present in order to update an index of length n. This property also allows us to update the indexes in a structured manner - we start from the shortest $k = 1$ indexes, and work our way up towards longer paths, as discussed in the next section.

Naturally, there is a trade-off between storing longer indexed paths directly or combining several shorter indexed paths during evaluation time. In Chapter 8, we discuss this further.

	k=1	**k=2**	**k=3**	**k=4**	**k=5**	**k=6**	**k=7**
#(1-label paths) needed		2	4	6	8	10	12
#(2-label paths) needed			2	4	6	8	10
#(3-label paths) needed				2	4	6	8
#(4-label paths) needed					2	4	6
#(5-label paths) needed						2	4
#(6-label paths) needed							2
#Total	0	2	6	12	20	30	42

Table 4: The number of indexed sub-paths needed to update a path index using self-maintaining translations.

5.6 BATCHING INDEXING MAINTENANCE

Rather than directly translating index updates for every single node or edge update in a transaction, index updates can also be batched. We motivate the choice of batching translation by two examples as described in the next section.

5.6.1 *Motivation*

Consider the existing graph in Figure 11. Now, we add three new edges in the same database transaction, such that the edges have the same label Z, and share a common starting node c. Given that we maintain a path index on paths of the shape $A \xrightarrow{X} B \xrightarrow{Y} C \xrightarrow{Z} D$, a clever maintenance algorithm should only find the incoming paths into node c once, and use this information to update the path index. Batching translations can solve this issue, by only performing the sub-path retrieval once for a given node.

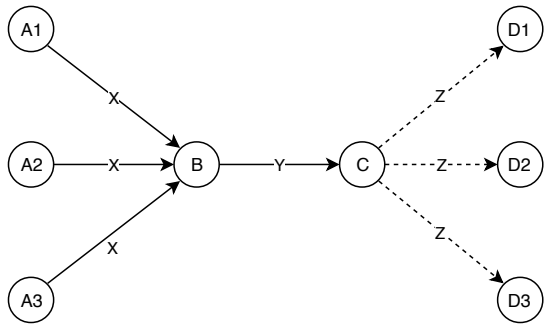


Figure 11: Adding three edges to an existing graph. Here, the dotted lines represent new edges.

Batching can also resolve issues inherent to self-maintaining translation. Consider the problem of maintaining an index on paths of the shape $A \xrightarrow{X} B \xrightarrow{Y} C$. Now, we add three edges to a graph during a single transaction. Figure 12 illustrates this example. If we are to use self-maintaining translation as described in Section 5.5, this will require use to also index paths $B \xrightarrow{Y} C$ and $B \xleftarrow{X} A$. In fact, we require the index to be up to date on these two sub-paths *before* we can update $A \xrightarrow{X} B \xrightarrow{Y} C$, thus implying an ordering on handling edge updates - processing added y edges before the added x edge. There are also scenarios where edge-at-a-time index update translations are impossible. Consider maintaining the indexes $A \xrightarrow{X} B \xrightarrow{Y} C$ and $C \xrightarrow{Y} B \xrightarrow{X} A$ after a transaction with two edge insertions as shown in Figure 13. Here, there is no ordering possible where both indexed paths can be maintained when considering the edges one-at-a-time. Instead, we should consider edge updates length-at-a-time.

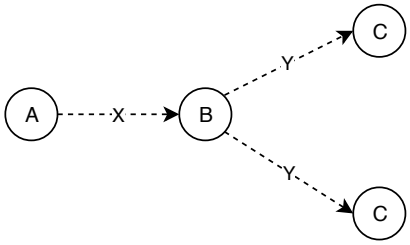


Figure 12: Three updates in a single transaction that imply an ordering on visiting edges for path index updates.

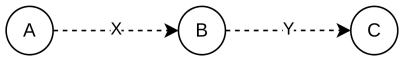


Figure 13: Two updates in a single transaction while maintaining $A \xrightarrow{X} B \xrightarrow{Y} C$ and $C \xrightarrow{Y} B \xrightarrow{X} A$. No edge-at-a-time processing of update translation is possible.

5.6.2 *Batching Method*

We propose a batching method that solves the ordering problem introduced in the previous section, as well as avoids redundant executions of the `PathsComingInto` subroutine. To solve the update ordering problem, recall one of the properties of self-maintaining updates: An index on paths of length n can be maintained given that we have an up-to-date index on sub-paths of length at most $n - 1$. Thus, as opposed to translating updates edge-at-a-time, we should collect index updates incrementally based on path length. Given that writing to the index mid-transaction can potentially be expensive, we collect the updated sub-paths of lengths $\leq n$ in a cache. This cache can then be used to perform self-maintaining translations for longer paths in the same transaction.

Algorithm 3 shows how such a cache can be utilized while processing path updates incrementally by length. Here, we let U be the set of edge updates (additions/removals) in a single transaction. An element $u \in U$ represents the addition or deletion of a single edge $a \xrightarrow{X} b$). The main differences between this algorithm and Algorithm 2 are:

- The use of cache $\mathcal{C}$ allows us to get a set of paths $\mathcal{P}_\mathcal{T}$ that were created in the same transaction. Once a new path is concatenated together, we add it to this cache.
- Length-wise processing of updates. Given that we only want generate updated paths of length i, we ensure that the total length of $(P_{in} \cup (a \xrightarrow{X} b) \cup P_{out})$ equals i.

	Reads graph store	**Multi-update transactions**
TBT + Single Update	Yes	No
SMT + Single Update	No	No
TBT + Batch	Yes	Yes
SMT + Batch	No	Yes

Table 5: Overview of the maintenance technique for path updates.

- We use different variations of the `PathsComingInto` and the `PathsComingOutOf` subroutines that return only paths of a given length.

Algorithm 3 Batch translation of edge updates U to path updates

1: $\mathcal{P} \leftarrow \{\}$
2: $\mathcal{C} \leftarrow \{\}$
3: **for** $i = 1$ to max_length **do**
4: **for each** $u \in U$ **do**
5: **for** $in_length = 0$ to i **do**
6: $out_length = i - in_length - 1$
7: $\mathcal{P}_{in} \leftarrow$ `PathsComingInto(`u`,` in_length`)`
8: $\mathcal{P}_{in} \leftarrow \mathcal{P}_{in} \cup \mathcal{C}(u, in_length)$
9: $\mathcal{P}_{out} \leftarrow$ `PathsComingOutOf(`u`,` out_length`)`
10: $\mathcal{P}_{out} \leftarrow \mathcal{P}_{out} \cup \mathcal{C}(u, out_length)$
11: **for each** $P_{in} \in \mathcal{P}_{in}$ **do**
12: **for each** $P_{out} \in \mathcal{P}_{out}$ **do**
13: $P_{new} = (P_{in} \cup u \cup P_{out})$
14: $\mathcal{P} \leftarrow \mathcal{P} \cup P_{new}$
15: $\mathcal{C}(start_node(P_{new}), i) = P_{new}$
16: $\mathcal{C}(end_node(P_{new}), i) = invert(P_{new})$
17: **return** $\mathcal{P}$

5.7 COMPARISON OF TECHNIQUES

Table 5 summarizes the four combinations of techniques that can be used for path index maintenance. Batch translation is a requirement for almost all transactions that contain more than one update. Only in the cases where the updates are completely disconnected in the graph, single update translation will still work correctly. A small advantage of these type of updates is that using the cache $\mathcal{C}$ can be avoided, possibly creating a small advantage in terms of running time.

Much more interesting is the difference in performance between traversal based translation (TBT) and self-maintaining translation (SMT)

in different scenarios. Chapter 7 will investigate these differences in both the worst- and best-case settings for both translation methods. In all cases, when selecting a preferred translation method, the performance overhead of maintaining $n \cdot (n-1)$ sub-path indexes for SMT should also be taken into consideration, which becomes significant for long indexed paths.

6 IMPLEMENTATION

This chapter describes the implementation of MAGPIE as an embedded part of the Neo4j graph database. Section 6.1 summarizes how we have implemented the *k*-path index as designed by Sumrall in [24] into the existing Neo4j indexing data structures. Section 6.2 adds how the proposed solutions for path index maintenance have been added to the transaction processing pipeline. Both TBT and SMT were implemented into Neo4j, as to benchmark the performance of both translation methods in different scenarios. Evaluating both methods provides insights into which method is most effective in a given setting. As using a path index during query evaluation was not integrated into the Neo4j query planner, TBT is used as a baseline to measure maintenance time. Evaluation of TBT and SMT against query workloads with both reads and writes is discussed as future work.

6.1 INDEX DESIGN

To demonstrate that our newly proposed path indexing technique can be used in an industrial database, we have constructed a prototype path index integrated into the Neo4j database. We exploited the existing infrastructure for node, edge and property indexes in Neo4j, and built the path index on top of the existing GB+ tree native index. The next subsections elaborate on the design choices made in our implementation.

6.1.1 *The GB+ Tree*

The design of the GB+ tree (Generation-aware B+ tree) as implemented in the Neo4j database is inspired by the work of Sullivan and Olson in [23], which describes the design of a B+ with efficient recoverability. The GB+ tree is lock-free as well as recoverable, thus an excellent choice for a recoverable database. The GB+ tree's internal nodes are categorized into three groups: stable, unstable and crashed. Once an edit is made to a node in the tree, the node is copied to a new, unstable generation. Edits will be made in such an unstable generation, which will be marked stable when the updates are completed. Given that the database crashes during updates, the tree will be recovered to its latest stable generation. On the API surface, the GB+ tree is used just like a B+ tree, thus *k*-path indexing can be implemented without any extra work.

6.1.2 *Key Design*

Recall that a path in a graph can be uniquely identified by at least the identifiers of the edges along the path. Given that we want to exploit the prefix-search property of the B+ tree for maintenance as in Section 5.5, we are also required to store the ID of the first node in the path. Although not needed, we also store the identifiers of the other nodes in the path. During query evaluation, these could also be needed, and if not stored in the index, additional information must still be read from the graph store.

6.1.3 *Multiple Trees for Multiple Paths*

In Neo4j, every index is stored in a different instance of a B+ tree. Even though the *k*-path index can be used to store multiple label-paths in the same B+ tree, we opted to store every indexed path in a different tree. This design choice was made for the following reasons:

- Storing each label-path in a different tree ensures that a tree is directly linked to a given label-path, which fits the Neo4j indexing architecture. Also, no additional data structures are required to keep track of which paths are indexed.
- When the decision is made to no longer index a given label path, we can delete the entire tree in memory as opposed to executing a large number of delete operations, which is significantly faster. Similarly, bulk-loading can be done if we decide to index a new label-path.
- Storing all indexed paths in a single tree may be disadvantageous to infrequently appearing patterns. A larger B+ tree can result in longer lookup times, which also result in slower write operations to the tree.

6.2 INDEX MAINTENANCE

Recall from Chapter 5 that we divide the index maintenance process into two steps:

1. Translation from entity updates to path updates.
2. Updating the index data structures (i.e., writing to the B+ tree)

In the last chapter, we proposed two translation methods to implement: TBT (Traversal-based translation) and SMT (Self-maintaining translation). This translation in itself can also be subdivided into two parts:

1a. Given an updated entity, collect the sets of paths P_{in} and P_{out} connected to that entity. This can be done either by prefix-searching the index of a shorter path or by traversing the graph.

1b. Concatenating the sets together. In the case of an added edge $a \xrightarrow{X} b$, the newly generated paths are $P_{in} \cup (a \xrightarrow{X} b) \cup P_{out}$.

Section 6.2.1 provides a high-level overview of the MAGPIE implementation in Neo4j. It is important to mention that bulk-loading of the index (index construction) was not implemented in the prototype due to time constraints. In [24], this was accomplished with promising results, proving that such a construction method is viable for path indexes. For this reason, we have decided to focus solely on the maintenance of the indexes.

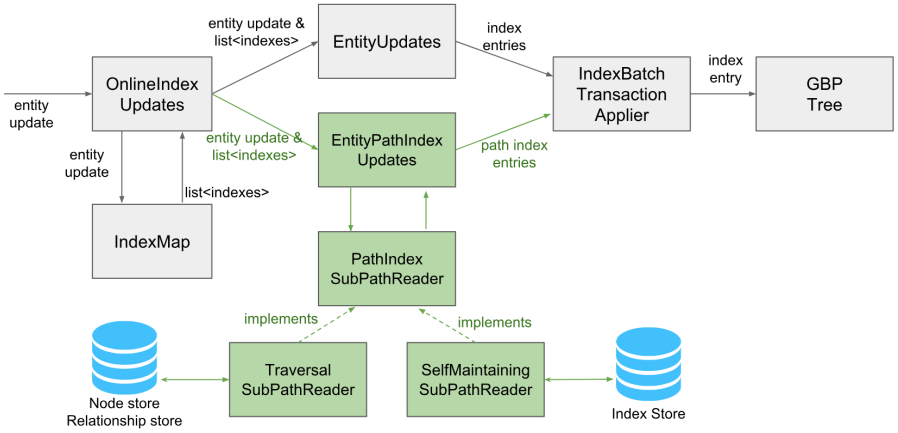


Figure 14: A high-level overview of the index maintenance implementation.

6.2.1 *Overview*

Figure 14 contains a schematic overview of the path index maintenance pipeline as implemented in Neo4j. The components in this figure have the following tasks:

- OnlineIndexUpdates - the main class responsible for index maintenance. An IndexMap object is used to retrieve the relevant indexes to update, based on a set of update commands.
- EntityUpdates - the class that converts commands to a set of logical index updates for Neo4j's property indexes. (This corresponds the translation step as described in Section 5.2)
- EntityPathIndexUpdates - the class that converts commands to logical index updates for the path indexes by implementing Algorithm 2. The class BatchEntityPathIndexUpdates (not in this diagram) performs the same functionality but allows for multi-update database transactions, as described in Algorithm 3.

- PathIndexSubPathReader - the interface responsible for step 1a of the maintenance process. This interface is implemented by two classes that perform TBT and SMT.
- IndexBatchTransactionApplier - writes logical index updates to the B+ trees, corresponding to step 2 of the maintenance process.

6.2.2 *Traversal-Based Translation*

The implementation of TBT executes a pruning breath-first search starting from a specified node, recording all paths up to a given depth. Two optimizations were made to the implementation:

- **Pruning** - When executing the translation step, we are only interested in a limited number of label-paths originating from a given node. For example, given that we only maintain an index on $A \xrightarrow{X} B \xrightarrow{Y} C$, and an edge $a \xrightarrow{X} b$ is added, we are only interested in finding all $B \xrightarrow{Y} C$ edges attached to b. Given that we have the prefixes of the label-paths we want to find, we can prune the BFS when we encounter irrelevant edge labels. Unfortunately, unnecessary traversals can never be completely avoided, as discussed later in Chapter 7.
- **Dense Node Optimization** - Given that we encounter a dense node in the traversal, we can use Neo4j's dense node optimization to avoid unnecessary traversals. In Neo4j, if a node has 50 or more edges, it is considered dense and contains references to relationship group records. These records allow us to get edges attached to a node by a specific label, as opposed to reading all edge records from disk one-by-one.

6.2.3 *Self-Maintaining Translation*

The implementation of the Self-Maintaining Translation method uses only the indexes to discover paths coming into a node. We then do not only avoid many random read operations from disk, but also avoid unnecessary traversals.

6.2.4 *Concatenating Paths*

In [24], it was shown that indexing all k-paths is infeasible for any $k \geq 2$. For this reason, we default to indexing only some of the k paths.

Algorithm 2 in Section 5.2 describes the basic logic behind a single update translation. This algorithm maps a list of entity updates to a set of materialized path entries for our B+ tree. To implement Algorithm 2 into Neo4j, there are a few other adjustments that need to be made:

- We need to take into account Neo4j path semantics. This semantics dictate that a path cannot contain the same edge twice, which should be checked before adding the new path to the index. That is, if P_{in} and P_{out} contain the same edge, we do not add this path entry to the index being maintained.

- A check whether the path $P_{in} \cup (a \xrightarrow{X} b) \cup P_{out}$ is one of the paths we want to index. If this specific label path is not in the list of specified indexes, we should skip this combination.

- If applicable, we also index the inverted version of $P_{in} \cup (a \xrightarrow{X} b) \cup P_{out}$. That is, if the newly added edge is part of an indexed label path with a backward X edge, we must also add the inverted path to our index.

When batching entity updates in a single translation, we do not only gather the sets of paths $\mathcal{P}_{in}$ and $\mathcal{P}_{out}$ before the concatenation step, but also generate updates incrementally by path length. Algorithm 3 in Chapter 5 describes the batch update process. When implementing the batch update translation, we take into the same considerations as when implementing single entity updates.

6.2.5 *Writing Updates to Index*

The IndexBatchTransactionApplier class is responsible for converting logical index updates into B+ tree updates that can be written to disk. Similar to the translation step, these updates are also batched to improve performance.

7 EXPERIMENTAL EVALUATION

To test the effectiveness of our new index maintenance methods as implemented in MAGPIE, we designed two experiments that measure maintenance time. The synthetic, controlled experiment in Section 7.1 investigates how TBT and SMT perform in different graph layouts. Here, we generate the graph layouts with high data locality, favoring traversal-based translation. Data locality is thus not variable, such that we can investigate the shape of the graphs in detail. The first experiment aims to measure maintenance time in the following scenarios:

1. The best case for traversal-based updates. This is when TBT does not read any unnecessary information. The difference in maintenance speed is thus determined by the time it takes to read from the data structures (reading the graph store versus reading the B+ tree).
2. This worst case for traversal-based updates, when TBT is required to do unnecessary traversals that cannot be avoided.

The experiment in Section 7.2 measures the speed of path index maintenance on a real data graph, given a realistic query workload. We measure the effect of using TBT or SMT given a workload that creates edges given a preferential attachment placement model.

7.1 EXPERIMENTS ON SYNTHETIC DATA

In our first experiment, we want to measure the impact of a single edge addition to a graph. This experiment is executed for transactions that create different numbers of paths, as to measure if one of the methods performs better in large updating transactions. All experiments are repeatedly executed, as to minimize and measure the error in the latency measurements. We influence the following parameters:

- The translation method to use (TBT or SMT).
- The number of paths generated by the update.
- The number of unnecessary traversals performed by TBT.

7.1.1 *Graph Layout*

The graph used by the synthetic experiment consists of 10 disconnected trees. Every one of these trees is a balanced tree of depth two,

where every non-leaf node has the same amount of children. We also add a disconnected node a with label A next to every tree. To measure maintenance time, we add an edge with label X between the a and the root of each tree (b), creating new paths in the graph. After the edge is added and the transaction is completed, we delete the edge to return to the initial state of the graph. This allows us to repeat the experiment with the same instance of the graph.

We use two different graph layouts to perform the experiment. Figures 15 and 16 show the shapes of these trees for the two layouts. Before the update, we also require an up-to-date index on label-path `(:B)-[:Y]->(:C)-[:Z]->(:D)`, as this is required for self-maintaining translation. The underlying motivation for the choice for these graph layouts is as follows:

- Both tree-structures will require the traversal-based translation method to read all edges from the graph store. The TBT method will need to traverse the `(:C)-[:Z]->(:E)` edges before discovering that these edges are irrelevant for the index update - node label E is not contained in the path index. The self-maintaining translation will not have this problem, and thus be in an advantage.
- Using 10 trees will allow us to cycle the trees and add/delete edges from different trees in one instance of the experiment. This does not only disable the database from doing unwanted optimizations (for example, cancelling out addition and deletion of the same edge), but also allows us to scale the size of the total graph without increasing the complexity of the experiment.

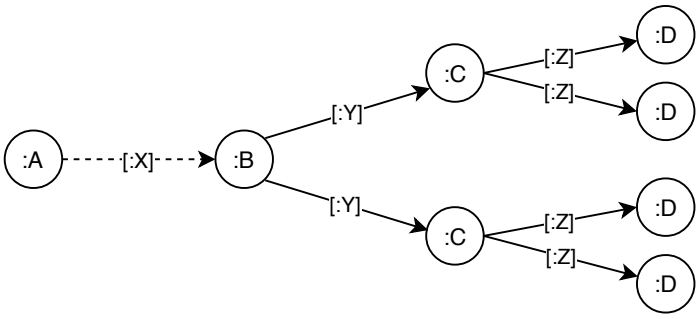


Figure 15: An edge addition with label `[:X]` that creates four new indexed paths. The traversal-based update method will perform no unnecessary work.

Due to the fact that the graph is restored to its original state after each execution of the experiment, the time measured corresponds to the time taken to add and delete an edge (two updates in a single

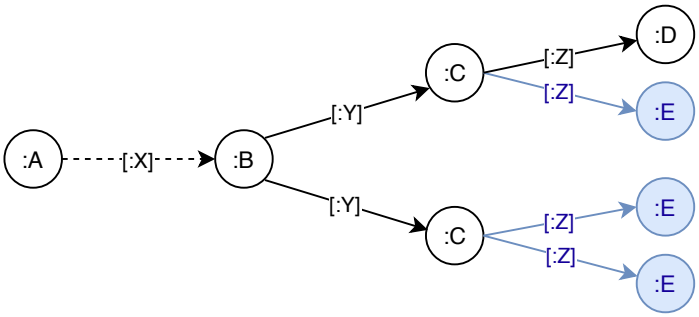


Figure 16: An edge addition with label `[:X]` that creates 1 new indexed path. The traversal-based update method will do extra work, as it runs into nodes with label `(:E)`.

translation). The maintenance time for maintaining after an edge update is thus half the measured value.

Our experiments were performed on a virtual machine with 64 gigabytes of RAM and 8 virtual CPUs. By running these experiments as part of the Neo4j benchmarking suite, we are able to execute the experiments repeatedly in a controlled environment and acquire more accurate results.

7.1.2 *Results*

The results of the experiments on synthetic data are listed in Appendix B.1. Recall that the measured time in milliseconds is the time taken to execute the entire database transaction, of which maintenance is a part. The results thus explain how different translation methods influence the entirety of the transaction, not the cost of translation by itself. Appendix B.1 contains the mean running time in milliseconds for each of the experiment settings, as well as the error rate. For small path index updates, there is little difference between the performance of TBT and SMT. When adding a single path, SMT is slightly outperformed by TBT (2.73% slower), but when adding 100 edges, SMT is 39.75% faster. The experiment that requires TBT to do extra traversals provides a slightly larger advantage to SMT, performing 47.61% faster.

In the experiments that cause TBT to perform extra traversals, unnecessarily traversed paths are not stored in the sub-index used by SMT. For this reason, the memory usage of SMT is significantly smaller.

The experiments with 10,000 and 250,000 paths, however, introduce large differences. Even in the cases where SMT and TBT process the same amounts of paths (no extra traversals), SMT is roughly 2.5 times faster than TBT. When SMT is able to avoid traversals, it outperforms TBT by a factor 47 and 1118 respectively. Given that more edge traversals can be avoided by self-maintaining translation, a higher relative speed-up in maintenance time can be achieved.

Figure 17 contains a *flame graph* for one of the experiments, which clearly the distinct phases in the maintenance process as described in Chapter 5. The flame graph is based on sampling stack traces to determine how much time is spent on each part of the code. In Figure 17, flame *A* corresponds to updating the B+ tree, flame *B* to the path concatenation step, and flame *C* to the translation. For this experiment, translation is a significant part of the updating process (roughly 25%). Writing to the B+ tree however, occupies over 50% of the time. Even though many optimizations can be made to translation, updating the B+ tree is also a time-consuming factor in this experiment. In other scenarios, e.g. the case where TBT does many unnecessary traversals, the translation step will take up a much larger part of the maintenance process, and writing to a B+ tree will be much less expensive. In the future, analyzing flame-graphs in different contexts may provide more fine-grained data on the specific maintenance steps' costs.

Figure 17: A flame graph for the experiment performing SMT on a tree with 10000 paths, where no unnecessary traversals can be skipped.

7.1.3 *Summary*

From our synthetic data experiment, we find that SMT outperforms TBT in almost all scenarios, except for when a very small amount of paths is traversed. This is likely due to the fact that the B+ tree allows for faster reads than the graph store, even when the data has high locality. For larger path counts, given that SMT can avoid unnecessary traversals, the speed-up becomes very visible. In the experiment with 250,000 edges, using SMT results in a speedup of a factor > 1000. Even though this may seem like a large number of paths, these scenarios could be common in real data. Given that a graph has an average degree of 500, a depth-two traversal will already visit 250,000 edges. If a depth three traversal is needed, a graph with an average degree of 63 will already visit over 250,000 edges on average. Self-maintaining translation thus becomes increasingly useful based on the length of

the path indexed, as well as in scenarios where the node degrees are high (for example, in dense clusters). Given that SMT can skip traversals that are unavoidable for TBT, an additional large speed-up can be achieved.

We have also introduced flame graphs as a technique for further fine-grained analysis of maintenance time. Even though a detailed flame graph analysis is out of scope for our current study, they could assist in further experiments, as well as debugging and optimizing the implementation of the maintenance algorithms.

7.2 EXPERIMENTS ON REAL DATA

To evaluate the performance of TBT and SMT on real data graphs, we measure the cost of maintenance on an open RDF dataset. The Geospecies Knowledge Base [30] RDF dataset was used for our experiments. With roughly 1.8 million triples, this graph contains aggregate information of the taxonomy of the animal kingdom and geological data. After conversion of the data to the property graph format, the data set contains 106 node labels and 95 edge labels. Figure 18 contains a high-level schema of the relations between the most important node and edge labels in the dataset.

7.2.1 *Experiment Design*

In order to effectively measure the cost of maintaining indexes for this dataset, we select a number of interesting patterns to index. As indexing all possible patterns is infeasible for the amount of node and edge labels in the dataset, we limit ourselves to a few patterns that could be interesting for a real application:

- Indexing first, second and third-degree topics for species, families, orders, classes, phyla, and kingdoms.
- Indexing the first degree topics for species, families, orders, classes, phyla, and kingdoms that are expected in a given geolocation. (This corresponds to a length two/three path-index)

We also ensure that the required sub-paths for SMT are present in the index. Next, we simulate a query workload by adding and removing edges to/from the graph. By using the Barabási–Albert (BA) model [1], we are able to generate a scale-free network, simulating a real query workload. Given a graph, a new edge is connected to an existing node i with probability p_i:

$$p_i = \frac{k_i}{\sum_{v \in V} k_v}$$

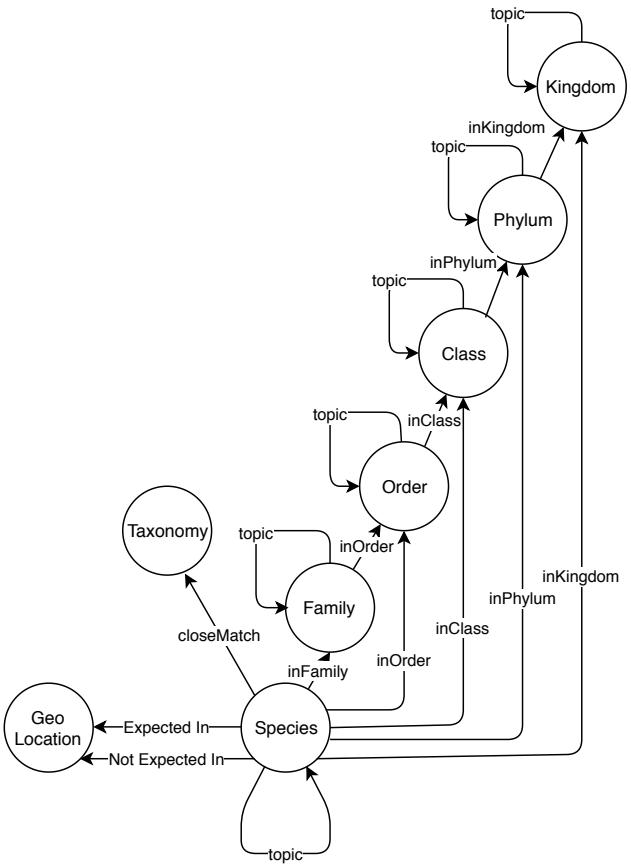


Figure 18: A high-level label schema for the Geospecies Knowledge Base RDF graph.

Here, k_i is the degree of a node i, and V is the set of nodes in the graph. This probability models preferential attachment - the fact that dense nodes are more likely to form relationships than more isolated (less dense) nodes. Using the BA model, we generate a fixed-size workload of 50000 edge additions and 50000 edge deletions and measure the execution time on two copies of the Geospecies Knowledge Base, using TBT or SMT. Our workload consists of `(:X)-[:TOPIC]->(:X)` edges, where X is the label of the node that the edge is attached to.

7.2.2 *Results*

Appendix B.2 contains the results of the experiments of executing the generated query workloads on the Geospecies Knowledge Base graph. By executing the experiments with fifteen repetitions, we collect the mean execution time in milliseconds. On average, using SMT has generated a 2.1 times speed-up compared to TBT.

Even though the experiments indicate that using SMT generates a speed-up, there is no clear evidence that data locality on disk influ-

ences the results. Given that the Geospecies dataset is small enough to fit in main memory, the real impact of random IO operations caused by graph traversals is hard to predict. In Section 8.2, we further discuss the possibility of executing experiments on data graphs that are too large for main memory, thus forcing the occurrence of random IO operations.

8 CONCLUSION

This chapter summarizes the most interesting findings from our study, as well as gives some concluding remarks. Section 8.1 outlines the results of the analyses on path index maintenance and query logs. Section 8.2 elaborates on opportunities relating to the structural analysis of Cypher logs, and concludes with possibilities for future work regarding path index maintenance and the use of path indexing in general.

8.1 SUMMARY

We implemented and evaluated MAGPIE, the first maintainable path indexing method for the industrial graph database. To showcase its applicability to a real graph database, we implemented k-path indexing into the Neo4j indexing architecture, as well as added k-path index maintenance as an integrated part of the Neo4j query processing pipeline.

To investigate the value of path indexing for real-world applications, we performed the first-ever large-scale structural analysis on Cypher query logs. Our analysis of query logs has shown empirical evidence that a path index can aid in query evaluation, as many of our query logs contain (long) frequent patterns that can be indexed. We have also shown that some of these query logs contain frequent updates that can create or delete paths, motivating the need for a maintainable path index.

By an in-depth study of the maintenance process, we have divided path index maintenance into two phases: update translation and writing to index data structures. Our experiment on synthetic data has shown that we are able to decrease maintenance time by a number of magnitudes by *self-maintaining translation*, reading from existing k-path indexes on shorter sub-paths. In the worst case, self-maintaining translation performs only slightly worse than traversal-based translation. Our experiment on real graph data has also shown that SMT can cut down maintenance time significantly if we are using a preferential attachment workload model.

8.2 FUTURE WORK

Both our query log analysis and our study of index maintenance have many interesting possibilities for future work. The following topics are of interesting to look into in relation to the query log analysis:

- Further investigation into the cost of evaluation query patterns. To be able to accurately decide which sub-patterns are worth indexing, we should be able to identify which parts of the query are the most expensive. This is not currently visible in the Neo4j log files, as the logs contain only the total execution time of the query. The value of indexing a sub-pattern naturally does not only depend on the frequency of the pattern, but also the relative speed-up that indexing the pattern produces. To estimate this speed-up, both an estimate for evaluating the pattern with and without indexing is needed, which can be based on observed execution times.

- A natural next step of our research is to look into developing an adaptive path index. Given a certain budget based on the cost of maintenance and the relative speed-up the index can deliver, we select an optimal set of patterns to index. With this, steps can be made to lead up to the development of a workload-driven index advisor for path indexes.

On the topic of path index maintenance, there are many opportunities for future work:

- Indexing label-paths in combination with properties. Given that some of the indexed label-paths may not be very selective, it can be interesting to index properties in combination with paths. This can prevent expensive filtering on a property value after utilizing a path index due to the large number of intermediate results.

- Directly investigate how data locality in the graph store influences the maintenance cost. By running experiments on graphs that do not fit in main memory the true effect of data locality can be better measured. Given that graph traversals could involve many random IO operations, traversal-based translation will likely become increasingly more expensive as the data is scattered over pages that are far away on disk.

- Investigating concurrency. Given that an industrial graph database should be able to handle concurrent querying, we should investigate how the index update translation behaves during concurrent updates. Locking and other synchronization mechanisms could potentially be needed during the update translation procedure.

- Evaluation of SMT and TBT on real query workloads that contain both reads and writes. This can provide a baseline: comparing the usage of a path index with maintenance versus not using a path index in query evaluation. To accomplish this, the query planner should be aware of existing path indexes, which can then be utilized as part of query evaluation. These kinds of experiments can give insights into the problem of adaptive indexing and accurate estimation of the speed-up a path index can produce.

- Hybrid approaches of traversal-based translation and self- maintaining translation. When adding a new edge, there is a possibility of using SMT to gather some of the incoming paths, and TBT to gather other paths. Using hybrid approaches can decrease the number of sub-patterns that need to be indexed, cutting down the storage space used by the indexes. In addition, the extra work required to maintain the shorter path indexes can be eliminated.

APPENDIX A: QUERY LOG DATA

This appendix contains summaries of the analysis on the query log data as described in Chapter 3. Appendix A.1 contains the counts of the types of queries that are present (read-only, write, and custom procedure calls), as well as the number of unique queries per log. For each log, we add a four-letter code denoting the industry the log originated from: (BANK = banking, TELE = telecom, CONS = consultancy, ANAL = analytics, COMP = computer hardware, TRVL = travel).

Appendix A.2 contains counts of query shapes for each of the query logs. In this table, we only consider the read and write queries and not the custom procedure calls. We identify between counts of chain queries, tree queries, forests and query graphs with loops. This summary also contains the number of queries that have a graph size of zero (no patterns), one edge, or variable length patterns.

Appendix A.3 contains a summary of the frequent patterns extracted from the query logs. Again, we only consider read/write queries, not the custom procedure calls. We denote a pattern as 'frequent' if the total occurrence is at least 10% of the query log size. Even though this threshold is arbitrary, appearing so frequently should be a good indicator that the pattern is worth indexing. We distinguish between five lengths of frequent sub-patterns, ranging from a single edge to five edges. Logs D8, D14, D24, D25, D29, D32, D43, D54, D63, and D66 were omitted from the results as they had under one hundred actual Cypher queries.

A.1 TYPES OF QUERIES

name	industry	total	total_unique	total_read	total_write	total_custom
D1	TELE	152635	101	130556	0	22079
D2	TELE	3158	40	3156	0	2
D3	CONS	425811	127	305	11	425506
D4	TELE	122562	180	99816	9	22746
D5	TELE	29726	79	23012	0	6714
D6	TELE	2838	104	2838	0	0
D7	TELE	15410	206	15033	0	377
D8	BANK	36	17	36	7	0
D9	ANAL	15683	7707	15430	0	253
D10	TELE	2804	7	2804	0	0
D11	TELE	102213	127	83250	0	18963
D12	TELE	14861	36	14861	0	0
D13	TELE	32382	269	31409	14	973
D14	TELE	76	3	3	0	73
D15	CONS	918556	916341	917076	916935	1480
D16	TELE	16654	236	16650	0	4
D17	TELE	36698	107	34634	211	2064
D18	TELE	18865	267	12547	10	6318
D19	COMP	567511	32556	543944	454981	247
D20	TELE	19642	236	19642	0	0
D21	BANK	1352	75	1351	21	1
D22	TELE	20040	189	19873	4	167
D23	BANK	4744	140	4695	0	49
D24	TELE	78	9	14	0	64
D25	TELE	99	2	8	0	91
D26	TELE	4406	89	4158	0	248
D27	CONS	1528104	1102571	1247851	1223419	280253
D28	TELE	8324	179	8324	0	0
D29	BANK	26	11	26	0	0
D30	TRVL	11364	3466	3579	3309	7785
D31	TELE	1495	21	716	0	779
D32	TELE	24	3	2	0	22
D33	TELE	21855	326	18706	21	3149
D34	TELE	7454	189	7069	101	385

name	industry	total	total_unique	total_read	total_write	total_custom
D35	TELE	29229	115	29226	0	3
D36	TELE	262	13	187	0	75
D37	BANK	12084	480	12025	94	59
D38	TELE	4476	65	4476	0	0
D39	TELE	9792	270	9234	0	558
D40	TELE	552	15	219	0	333
D41	TELE	955	40	922	0	33
D42	TELE	4415	93	4167	0	248
D43	TELE	98	11	17	0	81
D44	TELE	9622	117	9622	0	0
D45	CONS	149861	146366	146632	146449	3229
D46	TELE	34723	212	33508	12	1215
D47	TELE	615	30	546	0	69
D48	TELE	49567	66	46772	300	2795
D49	TELE	5631	31	5628	0	3
D50	TELE	37286	59	37168	0	118
D51	TELE	1862	68	1529	3	333
D52	TELE	21385	58	21327	0	58
D53	TELE	2075	70	1643	0	432
D54	BANK	46	15	45	0	1
D55	TELE	16336	121	15965	0	371
D56	TELE	1526	39	1450	1	76
D57	TELE	119220	313	84991	8325	34229
D58	TELE	29721	78	23007	0	6714
D59	TELE	57124	148	56924	0	200
D60	TELE	2809	191	2808	0	1
D61	TELE	148623	217	108301	25	40322
D62	TELE	575	38	541	1	34
D63	TELE	75	5	3	0	72
D64	TELE	8597	108	7875	7	722
D65	CONS	374714	131117	313115	197044	61599
D66	TELE	41	5	41	0	0

A.2 QUERY SHAPES

name	total_read	0_edges	1_edge	chain	tree	forest	loops	var_len
D1	130556	123635	511	6921	6921	6921	0	6297
D2	3156	141	0	3015	3015	3015	0	882
D3	304	136	110	144	113	115	53	0
D4	99816	85744	560	14072	14072	14072	0	11961
D5	23012	21809	1183	1203	1203	1203	0	19
D6	2838	332	4	2505	2506	2506	0	1259
D7	15033	13605	43	1428	1428	1428	0	1370
D8	36	1	1	26	29	32	3	0
D9	15430	6	5916	15424	5916	7245	8179	0
D10	2804	2618	0	186	186	186	0	186
D11	83250	77454	738	5796	5796	5796	0	5048
D12	14861	400	0	14461	14461	14461	0	14461
D13	31408	25970	58	5432	5426	5426	12	1435
D14	3	0	0	3	3	3	0	3
D15	917076	114	916839	916890	916920	916920	42	0
D16	16650	11294	190	5356	5356	5356	0	5166
D17	34634	33964	28	670	670	670	0	642
D18	12547	5402	595	7145	7145	7145	0	6550
D19	567254	132133	433822	435028	435056	435060	61	0
D20	19642	1585	2	18057	18057	18057	0	766
D21	1351	0	3	1321	1295	1333	18	0
D22	19873	4896	11895	14977	14977	14977	0	2845
D23	4695	0	22	4651	4559	4695	0	0
D24	14	14	0	0	0	0	0	0
D25	8	0	0	8	8	8	0	8
D26	4158	3408	711	750	750	750	0	39
D27	1247850	107264	1140189	1140472	1140434	1140435	151	0
D28	8324	2652	45	5672	5672	5672	0	5495
D29	26	1	1	19	25	25	0	0
D30	3579	34	510	3545	3333	3333	212	0
D31	716	41	0	675	675	675	0	567
D32	2	0	0	2	2	2	0	2
D33	18706	5124	220	13582	13582	13582	0	13220

name	total_read	0_edges	1_edge	chain	tree	forest	loops	var_len
D34	7069	2506	1552	4563	4563	4563	0	2824
D35	29226	25273	12	3953	3953	3953	0	3940
D36	187	184	0	3	3	3	0	0
D37	12025	2	18	11869	11598	11965	58	0
D38	4476	2414	13	2062	2062	2062	0	2022
D39	9234	8826	36	408	408	408	0	363
D40	219	5	0	214	214	214	0	214
D41	922	474	0	448	448	448	0	325
D42	4167	3417	711	750	750	750	0	39
D43	16	5	2	11	11	11	0	9
D44	9622	461	36	9161	9161	9161	0	9004
D45	146631	279	146286	146314	146292	146294	58	0
D46	33508	5379	18452	28129	28129	28129	0	9176
D47	546	113	0	433	433	433	0	91
D48	46772	46066	24	706	706	706	0	680
D49	5628	3873	1679	1755	1755	1755	0	1
D50	37168	35730	969	1438	1438	1438	0	137
D51	1529	1026	0	503	503	503	0	489
D52	21327	21127	21	200	200	200	0	179
D53	1643	1381	1	262	262	262	0	230
D54	45	1	1	37	39	44	0	0
D55	15965	15199	21	766	766	766	0	733
D56	1450	1243	0	207	207	207	0	200
D57	84991	47554	541	37437	37437	37437	0	36139
D58	23007	21804	1183	1203	1203	1203	0	19
D59	56924	56631	234	293	293	293	0	59
D60	2808	348	10	2460	2460	2460	0	1257
D61	108301	89832	184	18469	18469	18469	0	18200
D62	541	532	6	9	9	9	0	3
D63	3	0	2	3	3	3	0	1
D64	7875	7582	269	293	293	293	0	24
D65	313115	96098	216904	216977	216981	216989	28	0
D66	41	22	0	19	19	19	0	19

A.3 FREQUENT PATTERN ANALYSIS

name	query_total	|E| = 1	|E| = 2	|E| = 3	|E| = 4	|E| = 5
D1	130556	0	0	0	0	0
D2	3156	8	9	19	34	7
D3	304	52	16	0	0	0
D4	99816	4	4	4	4	4
D5	23012	0	0	0	0	0
D6	2838	7	6	16	32	6
D7	15033	0	0	0	0	0
D9	15430	8	12	0	0	0
D10	2804	0	0	0	0	0
D11	83250	0	0	0	0	0
D12	14861	8	8	8	8	8
D13	31408	4	4	0	0	0
D15	917076	8	0	0	0	0
D16	16650	4	4	4	4	4
D17	34634	0	0	0	0	0
D18	12547	8	8	8	8	8
D19	567254	4	0	0	0	0
D20	19642	8	7	14	26	13
D21	1351	1	28	8	8	0
D22	19873	8	4	4	4	4
D23	4695	9	36	12	12	0
D26	4158	2	0	0	0	0
D27	1247850	16	0	0	0	0
D28	8324	12	12	12	12	12
D30	3579	72	112	96	128	128
D31	716	6	4	4	5	4
D33	18706	12	12	12	12	12
D34	7069	6	4	4	4	4
D35	29226	4	4	4	4	4
D36	187	0	0	0	0	0

name	query_total	|E| = 1	|E| = 2	|E| = 3	|E| = 4	|E| = 5
D37	12025	15	28	12	12	0
D38	4476	4	4	4	4	4
D39	9234	0	0	0	0	0
D40	219	8	8	8	8	8
D41	922	10	8	10	11	8
D42	4167	2	0	0	0	0
D44	9622	12	12	12	12	12
D45	146631	8	0	0	0	0
D46	33508	12	8	8	8	8
D47	546	9	16	20	25	16
D48	46772	0	0	0	0	0
D49	5628	2	0	0	0	0
D50	37168	0	0	0	0	0
D51	1529	8	8	8	8	8
D52	21327	0	0	0	0	0
D53	1643	4	4	4	4	4
D55	15965	0	0	0	0	0
D56	1450	4	4	4	4	4
D57	84991	4	4	4	4	4
D58	23007	0	0	0	0	0
D59	56924	0	0	0	0	0
D60	2808	7	4	12	23	4
D61	108301	4	4	4	4	4
D62	541	0	0	0	0	0
D64	7875	0	0	0	0	0
D65	313115	16	0	0	0	0

B

APPENDIX B: EXPERIMENT RESULTS

B.1 SYNTHETIC DATA

The table below contains the results of the experiments on the performance of traversal-based translation (TBT) and self-maintaining translation (SMT). We record the time taken to complete a database transaction with two edge updates and the subsequent index updating. In one of the trees in the layout, an edge is deleted, and in another tree, an edge is added. The measurements thus represent the time in milliseconds required for the complete transaction, which contains *translation* for two disjoint updates. Each of the experiments was performed repeatedly, as such, we record both the mean and error (p=0.999) of the running time in milliseconds. The total size of the index (**memory**) is also reported. Note that for TBT, the size of the index is always 55.3KB (The size of a single empty index in Neo4j).

paths	**extra_t**	**method**	**running time**	**db size**	**index size**	**speed-up**
1	no	TBT	0.234 ± 0.00050	170.5 KB	53.3KB	
1	no	SMT	0.241 ± 0.00051	170.5 KB	+102.3KB	97.27%
1	yes	TBT	0.232 ± 0.00051	170.5 KB	53.3KB	
1	yes	SMT	0.239 ± 0.00055	170.5 KB	+102.3KB	97.12%
100	no	TBT	0.558 ± 0.00134	249,9 kB	53.3KB	
100	no	SMT	0.399 ± 0.00104	249,9 kB	+200.7KB	139.76%
100	yes	TBT	0.352 ± 0.00103	249,9 kB	53.3KB	
100	yes	SMT	0.239 ± 0.00051	249,9 kB	+102.3KB	147.61%
10000	no	TBT	37.449 ± 0.14707	9.9MB	53.3KB	
10000	no	SMT	14.342 ± 0.14810	9.9MB	+19.57MB	261.12%
10000	yes	TBT	10.996 ± 0.03813	9.9MB	53.3KB	
10000	yes	SMT	0.235 ± 0.00046	9.9MB	+102.3KB	4675.15%
250000	no	TBT	939.862 ± 33.21771	223.9MB	53.3KB	
250000	no	SMT	407.528 ± 26.69770	223.9MB	+439.6MB	230.62%
250000	yes	TBT	276.098 ± 7.06853	223.9MB	53.3KB	
250000	yes	SMT	0.247 ± 0.00946	223.9MB	+102.3KB	111808.51%

Table 6: Results of the maintenance experiments on synthetic data.

B.2 REAL DATA

method	**running time**	**db size**	**index size**	**speed-up**
TBT	342917.347 ± 667.45 ms	684.1 MB	65.75 MB	
SMT	163584.567 ± 25508.817 ms	684.1 MB	91.37 MB	209.63%

Table 7: Results of the maintenance experiments on real data. The reported mean and error are over the entire workload (100,000 queries).

BIBLIOGRAPHY

[1] Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks." In: *Reviews of modern physics* 74.1 (2002), p. 47.

[2] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. "An empirical study of real-world SPARQL queries." In: *arXiv preprint arXiv:1103.5043* (2011).

[3] Angela Bonifati, Wim Martens, and Thomas Timm. "An analytical study of large SPARQL query logs." In: *Proceedings of the VLDB Endowment* 11.2 (2017), pp. 149–161.

[4] Adam L. Buchsbaum, Paris C. Kanellakis, and Jeffrey Scott Vitter. "A data structure for arc insertion and regular path finding." In: *Annals of Mathematics and Artificial Intelligence* 3.2-4 (1991), pp. 187–210.

[5] Qun Chen, Andrew Lim, and Kian Win Ong. "D (k)-index: An adaptive structural summary for graph-structured data." In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 134–144.

[6] Edgar F. Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[7] George H.L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. "Efficient regular path query evaluation using path indexes." In: *19th International Conference on Extending Database Technology (EDBT)* (2016).

[8] George HL Fletcher, Dirk Van Gucht, Yuqing Wu, Marc Gyssens, Sofía Brenes, and Jan Paredaens. "A methodology for coupling fragments of XPath with structural indexes for XML documents." In: *Information Systems* 34.7 (2009), pp. 657–670.

[9] George HL Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. "Similarity and bisimilarity notions appropriate for characterizing indistinguishability in fragments of the calculus of relations." In: *Journal of Logic and Computation* 25.3 (2015), pp. 549–580.

[10] George Fletcher and Martin Theobald. "Indexing for Graph Query Evaluation." In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–9. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_212-1. URL: https://doi.org/10.1007/978-3-319-63962-8_212-1.

[11] Roy Goldman and Jennifer Widom. *DataGuides: Enabling query formulation and optimization in semistructured databases*. 1997.

[12] Ashish Gupta, Inderpal Singh Mumick, et al. "Maintenance of materialized views: Problems, techniques, and applications." In: *IEEE Data Eng. Bull.* 18.2 (1995), pp. 3–18.

[13] N. de Jong. *Correlation-Aware Cardinality Estimation for Cypher Queries*. 2018. URL: https://nielsdejong.nl/projects/graph/BGCE.pdf.

[14] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. "Exploiting local similarity for indexing paths in graph-structured data." In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE. 2002, pp. 129–140.

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The case for learned index structures." In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 489–504.

[16] Tova Milo and Dan Suciu. "Index structures for path expressions." In: *International Conference on Database Theory*. Springer. 1999, pp. 277–295.

[17] Knud Möller, Michael Hausenblas, Richard Cyganiak, and Siegfried Handschuh. "Learning from linked open data usage: Patterns & metrics." In: *Web Science Conference* (2010).

[18] Robert Paige and Robert E. Tarjan. "Three partition refinement algorithms." In: *SIAM Journal on Computing* 16.6 (1987), pp. 973–989.

[19] Francois Picalausa and Stijn Vansummeren. "What are real SPARQL queries like?" In: *Proceedings of the International Workshop on Semantic Web Information Management*. ACM. 2011, p. 7.

[20] Jaroslav Pokorný, Michal Valenta, and Jaroslav Ramba. "Graph Patterns Indexes: their Storage and Retrieval." In: *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*. ACM. 2018, pp. 221–225.

[21] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015.

[22] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. "NoSQL databases." In: *Lecture Notes, Stuttgart Media University* 20 (2011).

[23] Mark Sullivan and Michael Olson. "An index implementation supporting fast recovery for the POSTGRES storage system." In: *1992 Eighth International Conference on Data Engineering*. IEEE. 1992, pp. 293–300.

[24] Jonathan M. Sumrall, George H.L. Fletcher, Alexandra Poulovassilis, Johan Svensson, Magnus Vejlstrup, Chris Vest, and Jim Webber. "Investigations on path indexing for graph databases." In: *European Conference on Parallel Processing*. Springer. 2016, pp. 532–544.

[25] Martin Svoboda and Irena Mlỳnková. "Linked data indexing methods: a survey." In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2011, pp. 474–483.

[26] Octavian Udrea, Andrea Pugliese, and VS Subrahmanian. "GRIN: A graph based RDF index." In: *AAAI*. Vol. 1. 2007, pp. 1465–1470.

[27] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. "A comparison of a graph database and a relational database: a data provenance perspective." In: *Proceedings of the 48th annual Southeast regional conference*. ACM. 2010, p. 42.

[28] Xifeng Yan, Philip S. Yu, and Jiawei Han. "Graph indexing: a frequent structure-based approach." In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 335–346.

[29] Shijie Zhang, Meng Hu, and Jiong Yang. "Treepi: A novel graph indexing method." In: *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE. 2007, pp. 966–975.

[30] Peter J. deVries. *GeoSpecies Knowledge Base*. 2013. URL: http://lod.geospecies.org/.