Path indexing for efficient path query processing in graph databases

Sumrall, J.M.

Award date:
2015

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

Path Indexing for Efficient Path Query Processing in Graph Databases

Master's Thesis

Jonathan Maxwell Sumrall

Supervisor:
dr. G.H.L. Fletcher

Examination Committee:
dr. G.H.L. Fletcher
dr. D. Fahland
dr. T. Özçelebi

Eindhoven, July 2015

Wetenschap is de titanische poging van het menselijk intellect zich uit zijn kosmische isolement te verlossen door te begrijpen.

— Willem Frederik Hermans
Nooit Meer Slapen

Dedicated to my parents.

ABSTRACT

The study of path indexing in graph databases has not been studied before in an empirical way. Graph databases have become a popular choice among the different NoSQL databases for improved performance in certain use cases. However, recent work has shown that in situations where graph databases are expected to outperform the SQL solutions, the performance gains do not always materialize. Path indexing is one solution which offers graph databases improved performance in path query evaluation. This thesis presents a comprehensive look at path indexing for graph databases and the performance improvements it provides.

The first goal of the thesis is to present a framework for how a path index can be built. Paths in a graph are represented by the nodes and edges which constitute the path. A path instance is compactly represented in a graph as a K-length vector containing the path identifier followed by the identifiers of the nodes along the path.

A path index is then implemented using a B^+tree to store these keys, sorting the keys lexicographically. This data structure and sort ordering allows for range scans on keys by searching with any prefix of a key. Doing so, all the instances of all paths from the graph can be found by searching for the correct path ID in the index.

The topic of compression is also explored. This thesis presents a technique using ideas from Neumann in [19] to compress keys in the index significantly. This compression scheme out performs the LZ4 compressions scheme and is able to compress an index from 15.99GB to 1.69GB, a compression ratio of 9.5x.

Finally, using two synthetic datasets and one real-world dataset, this work provides an empirical evaluation of our implementation of this path index. All the datasets are of different sizes, and provide a thorough evaluation of our path index. The path index is compared to the graph database Neo4j. The results from these experiments show that the path index provides significant improvements to the query evaluation time in comparison to Neo4j. In a few cases Neo4j is able to cache its results to provide equally fast results, but otherwise the path index can provide speedups anywhere from 2x up to 8000x faster than Neo4j.

The implementation of this path index is provided as open source for further research and development.

ACKNOWLEDGMENTS

My project has been a collaboration between the Web Engineering group in the Department of Mathematics and Computer Science at the Eindhoven University of Technology and with Neo Technology in Malmö, Sweden. My daily supervisor has been dr. George Fletcher.

I would like to thank dr. George Fletcher for proposing this project when I first approached him asking for ideas for a thesis. I would also like to thank him for his continued support over the last 6 months guiding and advising me. I would also like to thank dr. Dirk Fahland and dr. Tanir Özçelebi for serving on my committee.

I am grateful to Neo Technology for assisting in this project and allowing me to visit their engineering HQ in Malmö. In particular I would like to thank Magnus Vejlstrup and Johan Svensson for their assistance in this project. I have learned a lot from them and their feedback and guidance has helped this project immensely.

The Eindhoven University of Technology and by extension the whole of the Netherlands also deserve my thanks for awarding me the ALSP scholarship and allowing me to spend these last two years focusing solely on my studies.

A special thanks goes to my parents, who have been supporting me throughout my entire education and even after telling them of my plan to move to the Netherlands to study. In addition, thank you to friends who have supported me, including Kirsten van der Meulen for giving feedback and proof reading this thesis, Ellen de Weerd for an assortment of help during all of my studies, Sander Kools for moral and caffeinated support, and to all my other friends who deserve many thanks for making my time at the university a truly enjoyable experience. You all have my heartfelt gratitude.

Max Sumrall

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 INTRODUCTION

Many new database systems have been developed in the past decade under the moniker of NoSQL, or "Not only SQL". The features and benefits of these systems are as varied and numerous as there are these new database systems. This class of systems was borne out of the need to cope with the sheer volume of data being produced by modern users and the latest applications. While these systems are all classified as NoSQL, this tagline is insufficient to describe the differences between all of the NoSQL databases. All NoSQL databases can be better defined under one of the following subcategories [16], each with their own attributes:

KEY-VALUE STORE: The simplest and fastest of the NoSQL databases. These systems store indexible keys and values. Given a key, the database can quickly retrieve the value associated with that key.
FEATURES: *High Performance, High Flexibility, Simple Design*

COLUMN STORE: Stores data in columns of a table rather than as a row, and uses this feature to achieve high performance and scalability.
FEATURES: *High Performance, Moderate Flexibility, Simple Design*

DOCUMENT DATABASE: Similar to the Key-Value Store. The values, called documents, are larger and give more structure to the data they contain.
FEATURES: *High Performance, Moderate Flexibility, Simple Design*

GRAPH DATABASE: The most different from the others. Graph databases store data within nodes, with relationships between the data being represented as edges between nodes . These databases are much more complex than the other systems, because of the way that data is modeled, stored, and retrieved. Graph databases are useful when making sense of vast amounts of loosely connected data. The number of interconnections between data may not be known, and therefore defining a schema required by other databases systems is not possible.
FEATURES: *Variable Performance, Highly Flexible, Complex Design*

Compared to the other types of database systems, graph databases provide less performance and have a much more complex design. This might lead one to conclude that choosing another type of database

system is a wiser choice. However the situations where graph databases are the best choice are those in which the other database systems *simply cannot be used to model, store, and retrieve the data you have encountered*, at least not without significant additional work to shoehorn the data to fit the model of the other systems.

As the description above shows, the performance of graph databases is variable. Simple queries like *"Find all users named 'John'"* can be easily modeled in both NoSQL and SQL databases, and will be faster than using a graph database. More difficult queries like *"Find all users who have a friend who like cycling"* are not trivial to model in some other NoSQL or SQL databases, but are trivial in graph databases. This simplicity often translates to higher performance for these queries, as the graph database evaluates the query by performing graph traversals, a natural operation on a graph. The other database systems would require intricate joining of different tables to evaluate the same query. However, the intricacies of modeling these types of queries in non-graph databases does not always mean worse performance, and it can be the case that the graph database does not outperform other systems.

Improving the performance of graph databases in these situations is the goal of this thesis project. This is achieved by identifying a specific class of queries, path queries, and proposing a method for improving query evaluation time for these types of queries. Our proposal is to construct path indexes, borrowing ideas from decades of research on index structures to design a path index which provides significant performance improvements without resorting to complex solutions. First, we look at an example dataset and see how it can be modeled in a graph.

1.2 A SMALL EXAMPLE

It is helpful to begin by first describing a situation where data can be formulated into a graph model. Figure 1 shows a small graph with example data that could be found in a social network setting. In the center of the graph is a user named John. The different people and things related to John are surrounding John's node, with relationships John has to those things represented as directed edges. Examining John's node, it can be seen that he *works for* Apple, is *friends* with Mark, and *lives in* Eindhoven, among other things. It can then be seen that Apple, the place John *works for*, has a person Mark who *consults for* them, and this person is someone John is *friends* with. There are many more types of connections which could be included for John, for Apple, and for all the other things in this graph. These connections between things are already present in social networks or semantic databases. By viewing this data in the form of the graph, it can be seen how certain people and things are related. Queries can then be formulated to

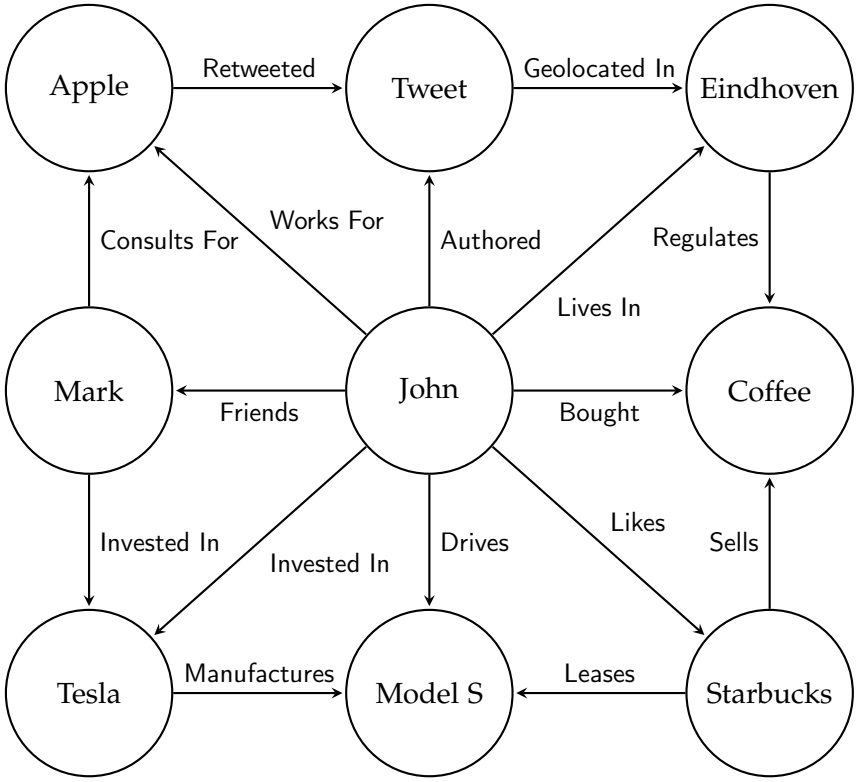


Figure 1: An example graph for a user and their relationships with other people and things.

ask interesting questions for all users in general, such as:

"Which stores does a user like that sells items bought by the user?"

This type of query can be answered by finding users and seeing if they have liked any stores. If so, check if that store sells any items which the user has bought. Such a query can be modeled as a path, shown in Figure 2.

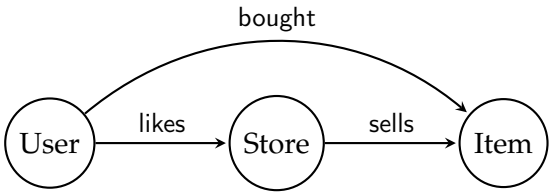


Figure 2: The graph model of a query to find an item *bought* by a user which is *sold* at a store the user *likes*.

1.3 PROBLEM STATEMENT

The main goal of this project is to design and construct a path index and supporting data structures and algorithms for performing path

indexing and path query evaluation on data in a graph database. The second goal is to explore methods for optimizing this index structure to reduce the overall size of the index and the cost of building the index. The third goal is to provide an empirical study of the performance of this implementation of a path index in comparison to a graph database.

To the best of our knowledge, this work is the first to provide a design and implementation of a path index specifically for graph databases as well as an empirical study into the performance of path indexes for graph databases.

1.4 OVERVIEW

The purpose of this chapter is to introduce the topic of path indexing and the efficient evaluation of path queries in graph databases. Chapter 2 gives an overview of the common theory and subjects this study relies upon. Chapter 3 gives an overview of different indexing techniques from the literature and their different merits. Chapter 4 covers the topic of path indexing, design choices and areas of contention when designing a path index. Chapter 5 details the specific path index implementation from this project. Chapter 6 provides a description of the environment used to conduct the empirical study of the performance of the path index implementation in this work. Chapter 7 presents an empirical study of the performance of the path index. Chapter 8 concludes the paper with a discussion of the results from the benchmarking and gives direction for future research.

2

PRELIMINARIES

This chapter begins by introducing the basic concepts and definitions which this work is based on. Section 2.1 introduces the concept of the graph, and the relevant makings of a graph. Section 2.2 introduces paths in the graph context, and Section 2.3 introduces the idea of the path index, the central topic of this work. Section refpropertyGraph-Section introduces a specialized type of graph, the Property Graph. Section 2.5 introduces the memory model used for implementing the systems later described.

2.1 GRAPHS

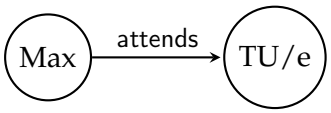


Figure 3: A simple graph with two nodes.

The work in this paper revolves around finite directed edge labeled graphs. A graph is a collection of nodes and edges. Nodes in a graph map closely to nouns in the grammatical sense. Edges are the relationships between nodes. Figure 3 shows a simple graph of two nodes and a single labeled edge between them expressing that the node "Max" has an 'attends' relationship to the node "TU/e". In a directed, edge-labeled graph each edge has a label and a direction. A directed edge means that an edge has a start node and an end node, and they cannot be reversed. For example, the edge labeled 'attends' in Figure 3 from "Max" to "TU/e" implies a relationship from "Max" to "TU/e", but does not imply that there is an 'attends' relationship in the reverse, from "TU/e" to "Max".

A graph is a triple $G = \langle N, E, \mathcal{L} \rangle$ where N is a set of nodes in the graph, $E \subset N \times N$ is a finite set of directed edge relations, and $\ell : N \to \mathcal{L}$ is the set of labels called the graph vocabulary. A *graph vocabulary* is a finite non-empty set $\mathcal{L}$ of *edge labels* from some universe of labels. An *edge relation* is a finite subset of $N \times N$.

2.1.0.1 *A note on notation*

In the remainder of this work, nodes and edges will be discussed often in the text. In such cases, the notation for nodes will be to surround the node with parentheses and for edges to surround them with square brackets. For example, the node Max and the edge at-

tends can be represented as (Max) and [attends], respectively. Relating edges to nodes will be written using dashes (-) and brackets (> or <), to represent undirected and directed edges. For example, the node Max having a directed edge labeled attends to the node TU/e would be represented as (Max)-[attends]->(TU/e). This choice of notation is not without reason— it is the formal notation used in the cypher query language, a graphical query language for the graph database Neo4j. More information about Neo4j is presented in Section 2.6.

2.2 PATHS

The fundamental indexed data in this investigation are paths in a directed graph G. Therefore an edge relation from node a to node b is the simplest path with a length of one. Often pairs of nodes, node a and node b, are referred to as *(source-target)* $\mathtt{paths}$, where a is the source of the path and b is its $\mathtt{target}$.

All paths in the graph with identical nodes and identical edges along the path are said to be identical paths. Otherwise, paths are said to be unique.

The focus in this work is the study of K-length paths. For a natural number K, we define a K-length path, $\mathtt{paths}_K(G)$ to be the set of all *(source-target)* paths such that there is an undirected path of length at most K in G from the source of the path to the target of the path.

2.3 PATH INDEX

A path index is a data structure containing the necessary information to describe all paths in a graph. It is assumed that for all nodes in a graph, there exists some identifying information to specify a single node in the graph. The data model representation of a path is described as an ordered set of node identifiers and the edges between the nodes. The goal of a path index is to identify which paths in a graph match a specified pattern. A pattern S is described as an ordered set of edge labels such that:

$$S = \langle e_1 \dots e_n \rangle, \quad e_i \subset \mathcal{L}, \quad 1 \leqslant i \leqslant n$$

An example of a property graph can be seen in Figure 4 and the paths and patterns obtainable from this graph are shown in Figure 5.

2.4 PROPERTY GRAPH MODEL

The property graph is an extension of the directed edge labeled graph. In a property graph, nodes have an identifier specific to that node within the graph. Edges have a unique identifier within the graph as

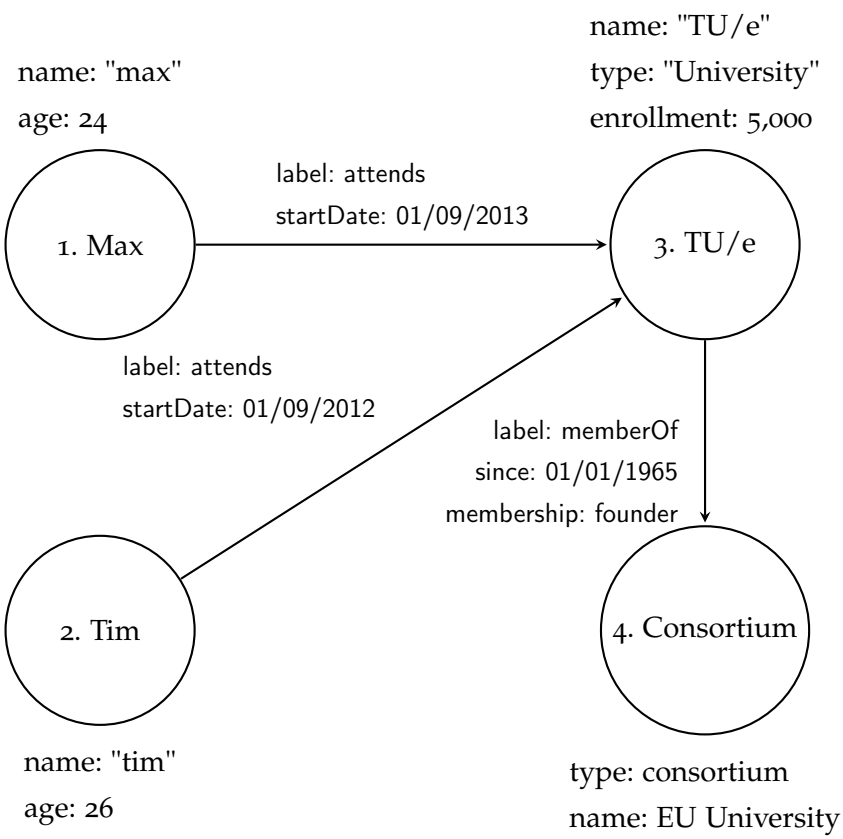


Figure 4: A property graph example.

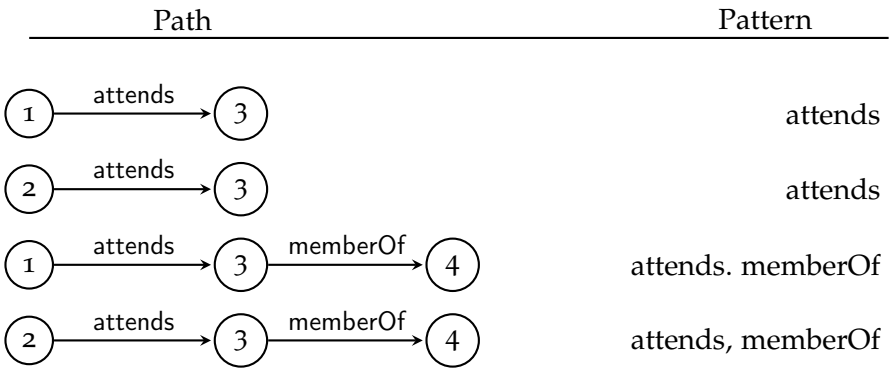


Figure 5: The paths and associated patterns from the graph in Figure 4.

well as a label, a source node, and a target node. Additionally, nodes and edges have a collection of key-value pairs. In Figure 4, node 1 has a collection containing the keys *name* and *age*, and node 4 has the keys *type*, and *name*. Edges can also have collections of key-value pairs, such as the edge from node 1 and node 2, which has the keys*label* and *startDate*. In graph databases which model the property graph, queries on the graph can contain qualifiers for certain key-value pairs, allowing for more selective queries.

2.5 MEMORY MODEL

Data locality, the place where data is stored and the distance it is from the processing unit, is in this work only considered in two forms: internal memory and external memory. Data stored in internal memory is considered to be quickly accessible and random access not incurring extra delays. External memory is considered to be slow, accessed only as a last resort, and with random access incurring extra delays. Internal memory is a limited resource in the computer, and algorithms only adapted to use this memory are therefore limited in size as well. External memory, commonly the spinning hard-disk or a flash-based solid state drive, is essentially unlimited in its storage space. However, the time for retrieving data from these external storage devices is high. Algorithms adapted to use external storage in addition to internal memory are called external memory algorithms.

External memory algorithms are adapted to use external storage space when the internal memory is exhausted. External memory algorithms work by transferring data from external memory into internal memory to do some computations, until the computation is completed. The time complexity of such algorithms is then dominated by the time needed to transfer data from the external storage to internal memory, since this time is often much greater than the time needed to actually do the computation. This time needed to transfer data from the external storage to internal memory is called latency.

Let B be the maximum amount of data able to be transferred from disk to internal memory. B is determined by the physical aspect of a system, such as the hard drive, other hardware, or software. The block size of the system is a contiguous set of data of size B read from external storage in one operation. A transfer from internal memory to external memory is called an IO operation. To improve the speed of external memory algorithms, one technique is to reduce the number of IO operations by operating on data in sets of size B.

2.6 NEO4J

Neo4j is a popular [7] open-source native graph database that implements the property graph model directly. Neo4j offers many features found in traditional relational databases such as being fully transactional. Neo4j also has support for cluster arrangements, offering high availability and scalability. Neo4j also uses its own query language, Cypher. With Cypher and the way that Neo4j stores data internally, modeling and querying data is intuitive. This ease in which data can be modeled and later queries is one of the main benefits of Neo4j. The architecture of Neo4j internally allows for this intuitive data modeling while still providing compelling query evaluation performance. Nodes are stored in flat files with the node ID being doubly used as

its location in the file. Edges in Neo4j are stored internally as doubly-linked lists. Properties on nodes and edges are stored separately.

In this work, queries on Neo4j are implemented in the Cypher query language, and executed through the Java API made available with the distribution of the database.

3 PATH INDEXING TECHNIQUES

Indexing data in database systems is a research field nearly as old as computer science itself. Many ideas have been proposed and implemented in commercial systems for more than three decades now. The most well known and earliest is the Join Index by Patrick Valduriez [23]. However such a data structure is not directly designed to deal with paths and path indexing.

The ability of an index structure to be used for paths is straightforward, however the multitude of indexing structures necessitates taking an overview of the different possible data structures to determine which most suits this problem.

Different fields of database research have produced solutions which fit those needs. It is therefore important to consider the different solutions proposed in the literature for data from relational data stores to solutions proposed for XML[1] data or graph data.

3.1 RELATIONAL

This section presents findings from the literature on indexing solutions proposed for relational databases. Certain examples, such as Section 3.1.2, are foundational works on indexing.

3.1.1 *Hash Index*

The hash index is an efficient implementation of an index structure using a hash function. Hash based indexes are shown to be especially efficient in large main memory systems as shown in [6]. Keys from relation R are hashed to find the relevant tuple of relation S in a constant lookup time. While hash based indexes are efficient in the case of checking for equality of keys in relations R and S, it is not possible to perform greater than or less than operations, as with B-trees, for example.

3.1.2 *Join Indexes*

The Join Index (JI) is a simple data structure with an efficient implementation proposed by [23]. The join index as proposed here is a prejoined relation stored separately from the operand relation. This small size and separate storage achieves improved performance for

1 http://www.w3.org/XML/

complex operations and allows for efficient updates. For two relations R and S, and attributes A and B, where A is an attribute of R and B an attribute of S, each tuple in the relations are uniquely identified by a system generated identifier denoted r_i and s_j respectively. Therefore, the Join Index is defined formally as

$$JI = \{(r_i, s_j) \mid f(\text{tuple } r_i.A, \text{ tuple } s_j.B \text{ is true})\},$$

With f a Boolean function defining the join predicate.

To allow for quickly looking up on r and s values each, the JI can be clustered on r and s respectively. Achieving this is done with two copies of the JI, one for clustering on r and the other clustered on s. If only r or s is going to be used, only one JI may be used instead. In this work, the join index is implemented using a B+ tree.

For path queries, it suffices to maintain multiple joins. Depending on the query, every join may be done using the set of join indexes. If so, all of the join indexes are joined. If it is not possible to perform all of the joins with join indexes, then slower join algorithms need to be used and combined with the possible join indexes. However, this can be inefficient, since for every access path a new join index must be built.

3.1.3 *Jive and Slam Join*

The join index as proposed by [23] is more efficient than other ad-hoc methods such as hash joins and their variations. It is also described as being simple, and therefore implementing it is easy. And it performs well for selective joins. However, this algorithm performs inefficient IO operations, reading in blocks only for a few tuples in some cases, or blocks being read in multiple times during one integration of the algorithm. The Jive and Slam join algorithms by [17] build on simple join indexes by improving the efficiency of the IO operations. Crucially, they only make one pass over the input relation. Join results are stored in a transposed file, a vertically aligned data structure. The algorithms are non-pipelined, as in they materialize views involving joins. By using the join index from [23], the original input relations, and temporary files, the Jive and Slam join algorithms provide a significant improvement due to performing only a single scan of the input relations. The authors of [17] also expand on the method of extending the Jive and Slam join to handle multiple relations. Creating a multidimensional set of partitions, the algorithm can still execute with only one pass over each input relation.

3.1.4 *Bitmap Indexes*

Bitmap indexes, as presented by [20], are efficient implementations of join indexes. In a traditional index table, each index value is associ-

ated with a list of row identifiers which have that value of the index. These list of rows can be represented as a bitmap, where for key value, the bit identifying a particular row is 1 if that row is a match for that index. This is a particularly efficient when the number of key values is small. When the number of attributes is large, the bitmap is likely to be sparse and may not be efficient in space since it requires a large number of zero bit values. In such a case, it is possible to use encoding to reduce the size of the bitmap. Using bitmaps results in large performance improvements, since large tables can be represented in a condensed form. Further, bitmap indexes have enormous computation advantages by using AND and OR to combine predicates, given that most hardware has specialized hardware to compute these operations quickly.

3.1.5 *Main Memory Database Systems*

In Main Memory Database Systems (MMDBS), the most useful subset of the whole database can be kept in main memory. To achieve this, designing a MMDBS focuses on efficient space utilization and efficient processing of database operations. In [21] the DBGraph Storage Model is presented which attempts to do this. Here, the database is represented as a bipartite graph composed of a set of tuple-vertices, a set of value-vertices, and a set of edges connecting these two sets. Then it is only needed to store attribute values once, therefore saving space. Relational tuples can be represented as a set of pointers to data values. The use of pointers is space efficient when large values appear multiple times in the database, since the actual value needs to only be stored once. DBGraph implements primitive operations like which can return for relation R the subset of tuples, ΔR, associated with relation for a given value. This implementation allows for a different approach to indexing. The simple join of R and S in DBGraph is a full scan over the values in the $\mathtt{Domain}_j$ and for each value in this domain, using primitive operations to retrieve the subsets of R and S which are containing tuples in R and S respectively where those tuples are connected by an edge value of the join attribute. Then taking the Cartesian product of ΔR and ΔS gives the join result. This is called the Join1 algorithm.

Since domains are shared across relations, the cardinality of the whole domain can be large compared to the cardinality of R or S. Therefore a faster join, Join2, works by first scanning the smallest operand relation. The join attribute value of each tuple of this relation is retrieved through the primitive operations. Then, with the set of vertices the subset of tuples of the other operand relation with the same value of join can be accessed.

In performance comparisons, the DBGraph Join2 algorithm always performs better than the join algorithm using Inverted Indexes, but

not for the join algorithm using join indexes, as in [23]. However Join2 does perform better in the case where a relation is temporary.

3.2 PATH INDEXING

One way to index graph databases is to index based on features of the graph. Graph features can be many things such as node or edge labels, paths, trees, or subgraphs. Given a path query to execute on a graph database, the path can be examined as a substructure in the graph to find. The aim of feature based indexing is to perform substructure search queries. Substructures which match the given query are candidate results to return. [12]

3.2.1 *DataGuides*

DataGuides[8] are a framework for giving structure for semistructured data. The DataGuide is built by taking a graph of data and returning a smaller graph with all of the nodes and edges of the same type or class only used once, while maintaining the same hierarchy of nodes and edges. Once this DataGuide is built, queries can be carried out on the DataGuide to match the node in the query. DataGuides are also annotated with a B-tree of the values of the nodes, allowing fast lookup of specific nodes by comparing those returned here with the nodes returned from examining the DataGuide. DataGuides can be exponential in the size of the initial graph, requiring a powerset of the nodes of the raw data. Fortunately, [18] shows that for Strong DataGuides on trees will reduce to a 1-index, introduced below, which does not exceed the size of the initial graph.

3.2.2 *T-index*

The T-index, or Template index, is a general index structure over semistructured data[18]. T-indexes are associated with paths in the graph as specified using path templates. T-indexes are computed by finding a bisimilar relation of the graph, which can be done efficiently. Objects in the database are grouped into equivalence classes, which allows for the T-index template path to quickly query for relevant substructures. Templates can be written in more general terms, to help speed up more queries and to save space. Or, space can be traded for more query speed by making more T-indexes on less general templates. Through query rewrites, a query can be evaluated under as many or as relevant T-indexes as possible.

The 1-Index and 2-index, also presented by [18], are both specializations and generalizations of the T-index.

The 1-index is designed to answer queries of the form $\boxed{P}\, x$, where P stands for path expression. We must also introduce the concept of a *refinement*, where an equivalence relation $\approx$ is a refinement if:

$$v \approx u \longrightarrow v \equiv u$$

The 1-index is a rooted, labeled graph where each node represents those nodes in the raw graph of their equivalence class, and has edges representing edges between the nodes of the equivalence classes. For each node in the 1-index, there is an associated list to all nodes in the raw graph that are of the nodes equivalence graph. Therefore, each node in the 1-index is only stored once and the 1-index does not grow larger than the initial graph. A query evaluated over the 1-index is done by first finding a path in the 1-index graph that satisfies the query, and performing a union on all nodes that exist in the raw graph of the same equivalence class of each node in the 1-index of the query path.

3.2.3 *A(K) Index*

The A(K) index is a reduction from the 1-index[14]. Like the 1-index, the A(K)-index partitions the nodes in the data into equivalence classes using their K-bisimilarity. Two nodes are K-bisimilar if they have the same incoming K-paths.

The 1-index and the DataGuide both precisely encode an entire graphs structure, which includes long and complex paths. This results in large index structures. Also, without considering the complexity of the graphs structure, the 1-index may compute separate paths in the graph when, on a local level, two nodes might be more similar than the 1-index accounts for. If the queries to be computed on the graph are also going to be long and complex, then the 1-index is a good choice. However, the authors of [14] build the A(K) index under the assumption that real world queries will not be as complex as the graph structure is, and instead take advantage of local similarity. The A(K) index classifies nodes in the graph based on paths of length K entering that node. This is achieved using K-bisimilarity as defined in [14].

An index node from the K-bisimulation is constructed and inserted into the index graph, and edges are created in the same manner as for the 1-index.

The performance benefits of using the A(K) index manifest in real world tests. In [14] queries were evaluated against data from a large graph of movies, and with $K = 3$, the A(K) index reduced query processing time approximately 47% compared to the 1-index, while also being 63% smaller than the 1-index.

3.2.4 *D(K) Index*

The D(K)-index[3] is another level of generalization on the A(K) index, by changing the value of K for different nodes based on the incoming queries. Objects which usually appear in long path queries have a higher K value than those which appear in short path queries. In this way, the D(K) index is an adaptive structural summary of the general class of graph structured documents. Further, the adaptability is able to be changed on the fly. By this definition, the 1-index and the A(K)-index are special cases of the D(K)-index. The construction of the D(K)-index proceeds similar as for the A(K)-index, with a preprocessing step to first compute the local similarity of the nodes in the graph.

3.3 K-PATHS

The topic of K-paths is the specific topic addressed in this thesis. K-paths are specifically defined in Section 2.2. The following subjects most closely study the topic of K-path indexes as defined in this thesis.

3.3.1 *Paths as Strings*

Index Fabric [5] represents every path in the tree as a string and stores it in a Patricia tree. This work utilizes ideas from [13] about indexing on unbounded length string data.

In comparison with Index Fabric with a commercial DBMS using a B-Tree index, the Index Fabric is faster by an order of magnitude or more.

Index Fabric also builds refined paths, an optimization for highly used paths for single lookup operations on those paths. Since paths usually change slowly, many refined paths can be made.

3.3.2 *GraphGrep*

GraphGrep[22] is a hash based method for finding all occurrences of a query in subgraphs. It allows for variable length paths. The initial construction happens only once, and is done by visiting every node in every subgraph and enumerating all of the paths from that node up to a fixed (small, usually 4) length and storing the path in a hash table with a column for every subgraph where the value is the number of paths in that subgraph. Doing this allows fast pruning of subgraphs which do not contain the query path. Once the subgraphs are identified, they are each scanned to find the matching path.

3.4 CURRENT STATE OF PATH QUERY EVALUATION

There are a number of papers which benchmark different database systems, such as [10] and [24]. Here we present some of their findings.

The work by [24] performs a comparison of the Neo4j graph database against the relational Oracle Database and the system using the Green-Marl Domain Specific Language. The experiment performed was the Dijkstra's shortest path algorithm. In their relational database, they built an index based on a simple relation table with columns for source node, destination node, and weight. For Neo4j, the example implementation solution provided with the database is used. The results show that for this computation, the relational database with an optimized SQL query always performs equally or better than the native graph database Neo4j. Yet the authors concede that in certain query types, Neo4j outperforms the relational solution. Those solutions tend to be ones with a large number of joins caused by a long query path. In these cases, the number of joins needed to be computed cripples the relational database.

In [10], a more comprehensive comparison of query evaluations of different databases, and therefore different indexing structures, is presented. The experiments are also conducted on a synthetic dataset with typical queries found in the real-world. The dataset is the Lehigh University Benchmark, a well known RDF benchmark containing universities, departments, groups, professors, and students in a social graph. The graph databases tested in the paper are Neo4j as well as Sparksee. Neo4j queries are written using the Cypher query language. The authors note that the Neo4j query engine does not perform cost-based query optimization, and performs queries as written. This is an area that can be investigated more closely for finding speed improvements. The results show that even for typical pattern matching operations such as triangle patterns, even unoptimized evaluations in the relational databases perform better than an equivalent evaluation with either of the graph databases in terms of query evaluation time.

4

A PATH INDEX DESIGN

This chapter describes the structures and mechanics of how a path index can be constructed, without going into details about a specific implementation which is done in Chapter 5. Section 4.1 describes the construction of paths from edge labels. Section 4.2 introduces the concept of a path identifier, and how these can be constructed in a deterministic fashion. Section 4.3 then describes how to form index keys based on paths and path identifiers, with index keys being the storage component in an index structure. Section 4.4 then explains how the key for certain paths can be constructed in an alternate order, based on query workloads in order to perform merge joins. Section 4.5 gives a description of how a path index can be built using a B^+tree. Section 4.6 then explains how searching with this index can be done using search keys. Finally, Section 4.7 explains how such an index structure can be maintained during transformations to the original graph.

4.1 PATHS

First, paths are defined with respect to the index structure in this work. Paths are vectors consisting of a set of edge labels. For example, consider an alphabet with edge labels $L_1, ...L_n$. An ordering is assigned to these labels, e.g. alphabetically. Then with the labels in sorted order, values can be assigned to them, values from $1, ...n$. We also consider the inverse of a label. For any label i, the inverse of the label can be defined to be $n + i$. With all labels and their inverse defined, a K-path can be uniquely identified by a K-path vector $(v_1, ...v_K)$ where each v_i is in the range $[1...2n]$.

4.2 PATH IDENTIFIERS

Based on a labeled path's K-vector representation, a unique integer is assigned to the labeled path to identify it.

To illustrate, suppose there is an alphabet consisting of two edge labels: A and B. Let A be lexicographically first. Then, there are the following path representations:

- A = 1
- B = 2
- A^{-1} = 3

- $B^{-1} = 4$.

Now, let K=2. There are then the following possible K-paths:

A	B	A^{-1}	B^{-1}
AA	AB	AA^{-1}	AB^{-1}
BA	BB	BA^{-1}	BB^{-1}
$A^{-1}A$	$A^{-1}B$	$A^{-1}A^{-1}$	$A^{-1}B^{-1}$
$B^{-1}A$	$B^{-1}B$	$B^{-1}A^{-1}$	$B^{-1}B^{-1}$

Table 1: Possible K-paths for alphabet of A, B.

Each of these K-paths can be denoted by their respective K-vectors as shown in Table 2.

$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle 3 \rangle$	$\langle 4 \rangle$
$\langle 1, 1 \rangle$	$\langle 1, 2 \rangle$	$\langle 1, 3 \rangle$	$\langle 1, 4 \rangle$
$\langle 2, 1 \rangle$	$\langle 2, 2 \rangle$	$\langle 2, 3 \rangle$	$\langle 2, 4 \rangle$
$\langle 3, 1 \rangle$	$\langle 3, 2 \rangle$	$\langle 3, 3 \rangle$	$\langle 3, 4 \rangle$
$\langle 4, 1 \rangle$	$\langle 4, 2 \rangle$	$\langle 4, 3 \rangle$	$\langle 4, 4 \rangle$

Table 2: K-vector transformation of K-paths in Table 1.

And finally, these K-vectors can be mapped and referenced by the value representing their ordering, namely: $\langle 1 \rangle = 1$, $\langle 2 \rangle = 2$, $\langle 3 \rangle = 3$, $\langle 4 \rangle = 4$, $\langle 1, 1 \rangle = 5$, $\langle 1, 2 \rangle = 6$, and so on.

4.3 KEYS

The index only stores keys, which are represented as atomic values. The objective of the path index data structure is to maintain a collection of instances of different paths from the graph. These unique instances of specific paths need to be constructed in a standard fashion following a set scheme such that specific elements of the path can be identified, that different paths can be compared to each other, and that these paths can be serialized. Such a formalized version of a path is called a key, to be used in the same sense as in other database models with key-value pairs.

A key must contain a minimum set of data to accurately describe a specific path in the graph. It must contain either:

1. the unique identifiers of the nodes along the path, and the labels of the edges between the nodes
2. the unique identifiers of the edges between the nodes in the path.

For example, Figure 5 shows the paths and patterns from the small graph from Figure 4. Using these paths, we can construct keys using the first method, with unique identifiers for the nodes along the path and the labels of the edges along the path as shown below in Figure 6.

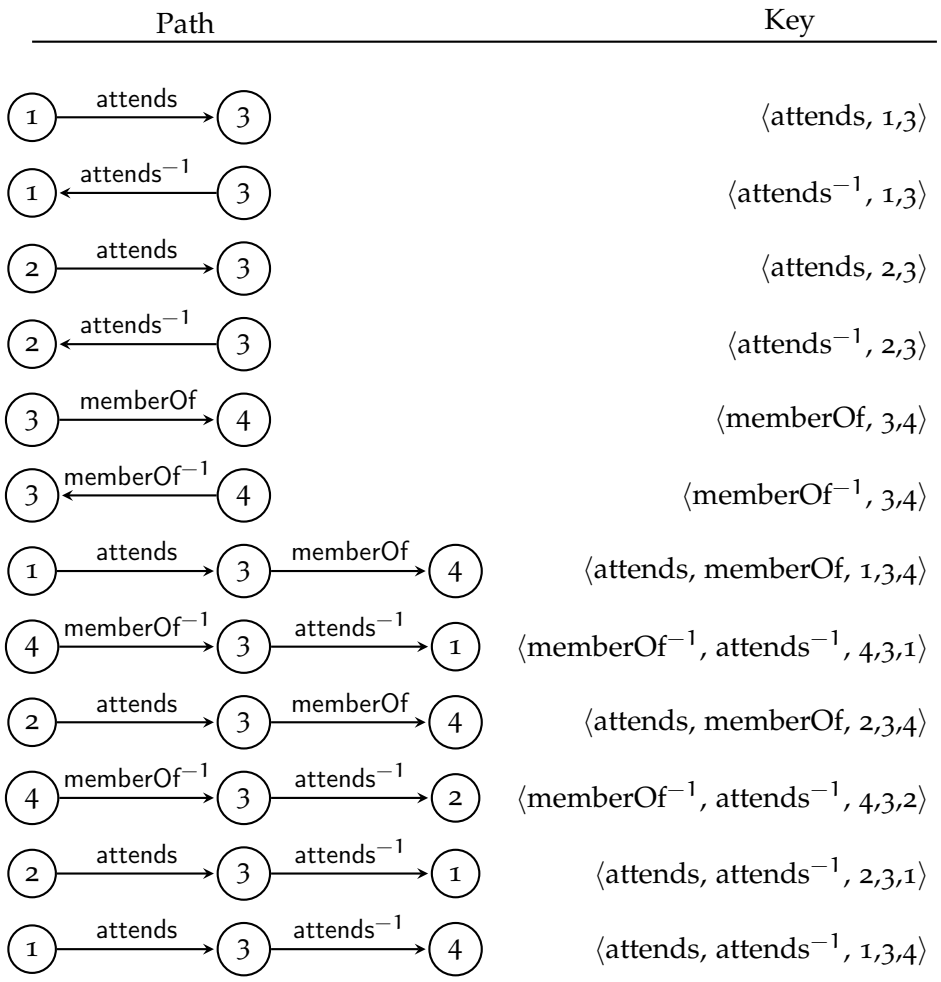


Figure 6: Keys as built using the nodes and edges along paths in Figure 5.

Indeed, as the size of the graph increases, the number of possible paths increases greatly. Further, many of the paths are simply different permutations of the same path. For a simple path, (Max)-[attends]-(TU/e), there is the inverse of it, (TU/e)-[attends^{-1}]-(Max).

4.4 PATH SIGNATURES

Up to now, paths, patterns, and keys make the assumption of defining their internal order based on the logical order of the edge labels along the path. From this, you can enumerate many different orderings of a path and its inverse to perform a particular query on the index. However, if this path index is part of a a larger query evaluation solution,

Result Set A ⟨ person, job ⟩	Result Set B ⟨ location, person, company ⟩
Alex, Electrician	Alabama, David, ATT
Bob, Developer	Colorado, Eugene, KPN
Charlie, Carpenter	Connecticut, Charlie, NYT
David, Builder	Delaware, Alex, ASML
Eugene, Accountant	Georgia, Bob, NBC

Table 3: Two different result sets which cannot be merge joined.

Default Signature ⟨ location, person, company ⟩	Modified Signature ⟨ person, location, company ⟩
Alabama, David, ATT	Alex, Delaware, ASML
Colorado, Eugene, KPN	Bob, Georgia, NBC
Connecticut, Charlie, NYT	Charlie, Connecticut, NYT
Delaware, Alex, ASML	David, Alabama, ATT
Georgia, Bob, NBC	Eugene, Colorado, KPN

Table 4: Result set B sorted on the left with the original, default, signature compared to the same result set but with a different sort ordering due to a modified path signature on the right. This allows for a merge join with result set A from Table 3.

the results from a query on the path index could require a different ordering of the results from a path index to be able to achieve a merge join between different result sets. In these cases, it would be beneficial to specify for a particular pattern and path identifier a special ordering to use for the values in the key.

For example, consider a query A searching for the pattern ⟨ person, job ⟩ and a query B searching for the pattern ⟨ location, person, company ⟩, with the results of these two queries shown in Table 3. However, if these queries were required to be joined such that they could return the people which appear in both result sets, a merge join would not be possible since the results from the second query are first returned sorted by the location, and only then by the person. To do a merge join between these two queries, a special signature can be specified for the second query such that that pattern is stored with the person node at the first item, and therefore the sort order will match that of the first query. The sort order effects of using such a path signature can be seen in Table 4, and the merged result set for these two queries can be seen in Table 5.

⟨ person, job ⟩	⟨ person, location, company ⟩
Alex, electrician	Alex, Delaware, ASML
Bob, developer	Bob, Georgia, NBC
Charlie, carpenter	Connecticut, Charlie, NYT
David, builder	David, Alabama, ATT
Eugene, accountant	Eugene, Colorado, KPN

Table 5: Merge join of result set A and result set B after using a modified signature for result set B.

Now result set B from Table 3 can be merged with result set A by using the modified signature specified in Table 4, and the results of which can be seen in Table 3.

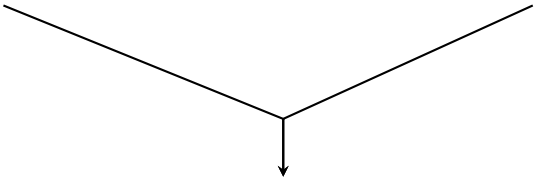
4.5 B+TREE DESIGN

As a storage mechanism, the B+tree is ideal for a path index [4]. The requirements are the ability to store and retrieve keys in as efficient of a way as possible for large sets of keys which may exceed the amount of internal memory in the system. In this regard, the characteristics of a B+tree are ideal. B+trees are suited to situations where the set of data to be searched is much larger than internal memory. This is because the B+tree has a large fanout and traversing to the leaf nodes of the tree can be done in only a few traversals. Specifically, locating a key stored in a b-order B+tree with n records can be done in $O(log_b n)$ operations. Further, B+tree support range queries naturally, given that the leaf nodes in the tree form a linked list. This is necessary since the keys stored in the tree will be full enumerations of paths, while searches will be done only for the path. Then, searches can be done for the first record in the tree matching our pattern, and by traversing along the linked list of leaf nodes we find all the matching keys. Searching is explored in more detail in Section 4.6.

This design of a B+tree differs from the classical description. In the context of path indexing, we only have the notion of fully specified keys rather than key/value pairs where the value can be some data or a pointer to a records file. With this in mind, only keys are stored in the B+tree, and the "value" is being able to find all of the keys matching a particular path id value in sorted order. The significance of the sorted order is being able to specify not only which paths to search for, but also specifying additional prefixes of full keys and directly finding a smaller subset of keys to return. The notion of prefix searching is explained in more detail in Section 4.6 below.

Set A	
⟨Person,	Job⟩
Alex,	Electrician
Bob,	Developer
Charlie,	Carpenter
David,	Builder
Eugene,	Accountant

Set B		
⟨Person,	Location,	Company⟩
Alex,	Deleware,	ASML
Bob,	Georgia,	NBC
Charlie,	Connecticut,	NYT
David,	Alabama,	ATT
Eugene,	Colorado,	KPN



Set A ⋈ Set B

⟨Person,	Job,	Location	Company⟩
Alex,	Deleware,	Deleware,	ASML
Bob,	Developer,	Georgia,	NBC
Charlie,	Carpenter,	Connecticut,	NYT
David,	Builder,	Alabama,	ATT
Eugene,	Accountant,	Colorado,	KPN

Figure 7: Sorted Merge Join of Set A and Set B where B is sorted with the modified signature.

4.6 SEARCHING

The primary goal of a path index is to retrieve fully specified keys from the index which represent paths in the graph. The canonical search on the path index would be in the form of "Retrieve all keys matching the path (x)-[edgeA]-(y)-[edgeB]-(z)". In this search, the path [edgeA, edgeB] would be transformed into a path identifier as specified in Section 4.2, and a search operation would be initiated on the index to search for all keys with the matching path identifier. For example, consider searching for all keys with the path identifier "B" in the tree in Figure 8. Starting from the root, the left most leaf node

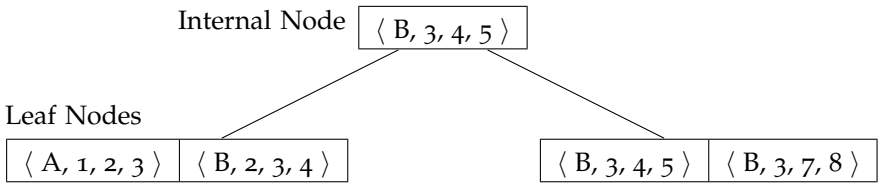


Figure 8: Small B$^+$tree storing four keys in the leaf nodes.

is traversed to since the key in the root node had the same value in the first position. Then, from the first node, we find the second key matching our path identifier, and continue searching into the second leaf node and finding the remaining two matching keys.

Another view to take on searching is that instead of merely defining searching as specifying a path identifier to find, we can define a search in the tree to match the classical definition for a B$^+$tree in which you search using a fully specified key. However searching for a specific key is not the desired search operation. Finding a specific key would require to know the full key upfront. Searching for a specific path identifier is a search for all keys where the first element of the keys are identical to the path identifier, and the remaining elements of the key are allowed to be any values. With this view, and by denoting wild card values with an asterisk (*), a search for the path identifier "B" in the graph in Figure 8 would be to search for the key:

$$\langle \mathrm{B}, *, *, * \rangle$$

A subsequent feature of this search key definition is that it becomes possible to search not only for keys based on the path identifier, but also on any prefix of the desired keys. In our example B$^+$tree, a search for keys with the path identifier "B" and the first node in the path having the identifier of 3 would be more efficient by specifying additional criteria in the search key. Without doing so, and only specifying the path identifier "B", the search would traverse to the left leaf node and inspect all keys until finding those which have the correct first node identifier. By giving the more details search key prefix, $\langle \mathrm{B}, 3, *, * \rangle$, a search traversal would end in the right leaf node, and the desired keys would be found sooner.

4.7 UPDATES

An index structure must first be able to improve the performance of query evaluations for the database they are assisting. In addition to that, updates to the database also requires the index to update itself to reflect the current state of the database. If any index is unable to do this, then it is less helpful in situations where changes to the database happen often. In situations where the database does not change often, and queries are mostly read operations, then such an index may still

be helpful. The subject of updates to the index and the graph at large are explored in the section below.

4.7.1 *A difference of perspective*

Performing updates to a B$^+$tree are relatively inexpensive. Locating a key or leaf node requires $O(\log_b n)$ operations, and inserting a key and removing a key also require $O(\log_b n)$ operations.

While the deletion of a specific key from the index is straight forward, it is not straight forward on how to translate changes in the graph to changes in the index. This translations requires the identification of which keys to add or remove from the index. Graph changes are assumed to be either node insertions or deletions, or edge insertions or deletions. In both cases, the added or deleted entity will need to be examined and all paths which intersect with that entity need to be enumerated. For node or edge insertions, this can only be done by performing a localized search in the graph. Such a search would be a typical breath or depth first search of depth k, and inserting each path found by this search into the index. Enumerating the different paths that a newly inserted node or edge creates cannot be done without doing such a search in the graph.

In the reverse case where a node or edge is removed from the graph, the same operation can be performed as in the insertion case, making sure to do the local search before the graph actually completes the removal of the entity from the graph. Such a search would enumerate all the paths the entity intersects with, and each of those paths can be submitted to the index for deletion.

Considering the nature of what it means to remove a node or edge from the graph and subsequently the index, some alternative solutions exist which can handle node or edge deletion without needing to reference the graph. When a node or edge is deleted, the information which is needed to update the index is all of the paths which that deleted entity belonged to. For example, in the example index in Figure 8, if node with identifier 4 is deleted, the keys $\langle$ B, 2, 3, 4 $\rangle$ and $\langle$ B, 3, 4, 5 $\rangle$ must be deleted. Since the sort order of the keys in the index is optimized for searching first by path identifier, and subsequent node identifiers in the order they appear in the path (unless there is a special path signature specified), then it is not possible to find paths in the index only by the identifier of the deleted node, 4. However, if $k + 2$ duplicates of the index were stored, each with a different sort ordering, it would be possible to search in each index with the first element of the search key set to the identifier of the deleted node. After performing all $k + 2$ searches, all of the keys which contain that node identifier will be found, and the index (including all the duplicated indexes of different sort orderings) can be updated to remove those found keys. This allows for removing keys from the index without

referencing the graph. The cost of this solution is a $k + 2$ multiple of both the space and time requirements of having one index and performing a single delete on that one index.

5 INDEX IMPLEMENTATION

With the rise of many NoSQL class databases, graph databases have become more popular alternative database architectures due to their data modeling properties. Further, certain query workloads cause relational databases to perform an excessive number of joins when disparate data tables are necessary to evaluate the query. In these cases, graph databases can use their alternative data model to their advantage and evaluate queries many times faster. A number of graph databases have emerged, including Neo4j, Orient DB, Titan ArangoDB, Giraph, and Sparksee. Of these, this work focuses on the Neo4j database. Neo4j is the best choice as it implements the graph model through to the storage layer. Other graph database systems rely on other database implementations, making a path index implementation difficult. Further, Neo4j is currently the most popular graph database publicly available [7].

In this chapter we describe the path index developed for the Neo4j graph database. This path index provides improved path query evaluation compared to evaluating identical queries in standalone Neo4j, shown by the results in Chapter 7.

In Section 5.1 the representation of nodes and edges in Neo4j and their path index duals are described. Section 5.2 describes how paths are mapped concisely to path identifiers. Section 5.3 details how searches are executed in the path index and describes the search cursor which searches resolve to. Section 5.4 gives an explanation of how the index structure is built, and the differences in this implementation from the standard B^+tree. Using the index can only be done with a large enough dataset, which necessarily leads to Section 5.5 which describes how an index can be efficiently initialized. Finally, in Section 5.6 the compression schemes our index uses are discussed.

This path index implementation is open source work, being made available for further research and development. [1]

5.1 NODES AND RELATIONSHIPS

Neo4j implements the property graph model all the way to the storage layer. Nodes and Relationships exist as unique data entities on the disk. Nodes, node properties, relationships, and relationship properties are each stored in separate files. Nodes and relationships are assigned a unique identifier when they are created. Given a node identifier, Neo4j can calculate an offset to use for directly finding the entry

1 https://github.com/jsumrall/Path-Index

for that node in the node file, which resolves to pointers on where to find the relationships and properties for that particular node. This architecture is also used for relationships, and pointers to nodes connected to relationships can be found by reading the relationships file at the offset calculated by the relationship identifier. By knowing the identifier of a node or relationship, you can reference all the information in Neo4j about that node or relationship. For building a path index, referencing only the node or relationship identifier is sufficient for evaluating path queries. Neo4j assigns unique identifier values to nodes and relationships, therefore reusing these values in the index is the straightforward solution.

5.2 PATH IDENTIFIER RESOLUTION

Given two paths with identical edge labels, the path index must produce an identical path identifier. For all paths in the graph where the edge labels are not identical, our path index must produce a unique path identifier. The implementation of building path identifiers is conceptually similar to the procedure described in Section 4.2. The goals of this implementation are to be fast in determining a path identifier, being simple to construct the path identifier in the case of new edge labels.

When parsing a path, the edges are examined in the order they appear in the path. Therefore, two paths with identical edge labels but in different orders will produce a different path identifier. By examining the edges in the order they appear in the path, the direction of the edges can be understood. All paths have a start node, and by examining edges beginning from the start node, the direction of the edge can be determined with respect to the subsequent nodes in the path. An edge is said to be *forward* if the ending node on the edge appears later in the path than the starting node on the edge. Otherwise, the edge is said to be *backwards*. Figure 9 shows a path where the first edge in the path is backwards, and the remaining edge is forward.

When building a path identifier where certain edges in the path are backwards with respect to the path, such a path identifier should be unique from a path with identical edges where those edges are is in the opposite direction. For example, in Figure 9, if the edge "drinks" was reversed, the path identifier should not be the same as if it was still backward. To fulfill this requirement, the edge labels of edges determined to be backward are reversed before being concatenated with the labels of the remaining edge labels. Doing so results in a different hash value for paths where edge directions are reversed. Figure 10 shows two paths, and the concatenated strings from each.

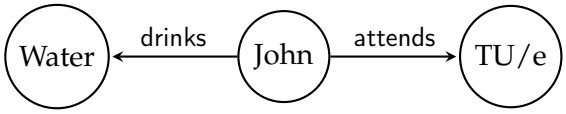
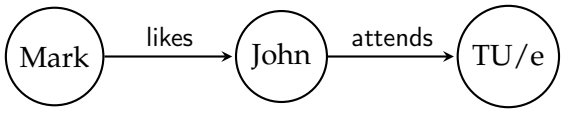
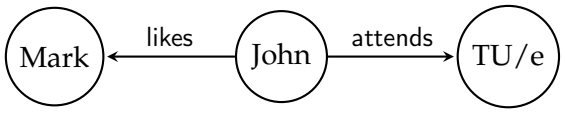


Figure 9: An example path with the first edge with label "drinks" being *backward*, and the second edge with the label "attends" being *forward*.



$$\texttt{PathIdentifier}(\text{(likes, forward), (attends, forward)}) = \texttt{hash}(\text{likes attends}) \rightarrow 123$$



$$\texttt{PathIdentifier}(\text{(likes, backwards), (attends, forward)}) = \texttt{hash}(\text{sekil attends}) \rightarrow 456$$

Figure 10: An example path with the first edge with label "drinks" being *backward*, and the second edge with the label "attends" being *forward*.

5.2.1 *Mapping Dictionary*

While paths are the underlying item stored in the index, only the identity of the path is necessary for the workloads defined in this paper. The primary concern of the index is to answer queries for a specific path without doing any deconstruction of the indexed path. The consequence of this is the lack of need of storing the edges which make up the path in an addressable way, rather, only the full path needs to be addressable or identifiable. Edges along a path are identified by labels represented as string literals. As these string literals can be varying length, combined with the K number of string literals for each edge along the path, it is inefficient to store the full set of edge labels which make up each path within the path index. Instead, the concatenation of the string literals of the edge labels of the path are replaced by ids using a *mapping dictionary*. This has the effect of reducing the size of paths in the index, by only storing the ids of the paths. Further, this simplifies query processing, as operating on the index can be done more quickly when comparisons are done between path ids rather than string literals.

The cost of this is the need for the mapping dictionary. When querying the index, the concatenation of the edge labels must be constructed and used to determine the path id to use with the index. This

can be implemented using numerous methods, and in this work the mapping dictionary is constructed using a hash map. In the data sets tested in 7, the number of paths is small and the implementation of the mapping dictionary is not a concern.

5.3 SEARCH CURSOR

This implementation of a path index provides support for searching using any prefix of a key stored in the index. In the general case, this is a search only using the path identifier. However, the search infrastructure does not differentiate between a search using a path identifier and a search with a path identifier and additional node identifiers. To achieve this, the comparison utility for evaluating the difference between two keys is modified to handle the case of a key with unspecific suffixes. In the insertion operation in the index, key order is maintained by comparing two keys and determining and order between them. In a traditional search operation in a canonical B^+tree with a fully specified search key, the tree is traversed by comparing the search key to the keys in the internal nodes. This implementation of the B^+tree lets the search key contain unspecified elements. When comparing the search key containing unspecified elements to a key in the tree, the unspecified elements are considered to be lexicographically before the specified element they are compared to. Doing so, a search key with only the path identifier specified will result in a tree traversal which leads to the first element containing that path identifier. However, since the first element of the search key is specified, the traversal will lead to the leaf node containing the first element with that same path identifier and not any other location.

5.3.1 *Edge Case Consideration*

In certain traversals, there is an edge case where an extra page access is required to ensure the actual first key of a particular path identifier is found. Figure 11 demonstrates a certain situation where this additional page access is necessary. Consider the situation where there is a search for the path identifier "B" in the tree in Figure 11. From the internal node, in both trees, we would traverse from to the left most leaf node since our search key has unspecified elements when compared to the full specified key $\langle$ B, 3, 4, 5 $\rangle$. In the top most tree in the figure, this is the correct traversal to make. However in the lower tree, this traversal does not find any keys in the left most leaf node with the path identifier "B", and has to traverse to the following leaf node to find the first result.

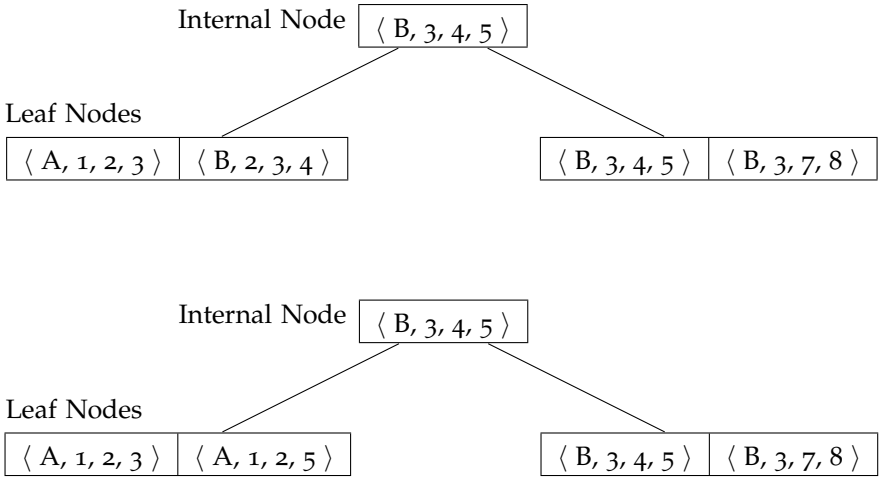


Figure 11: B+trees with different keys in the first leaf, demonstrating the need for an additional page access in these situations.

5.4 B+TREE ARCHITECTURE

5.4.1 *Page Design*

In the design of the tree, nodes are either internal nodes with references to other nodes lower in the tree and keys which direct traversals to those lower nodes based on a search key, or nodes are leaves which only contain keys. However all nodes contain a header with the essential information needed to interact with the node. The headers contain the references needed to find both the proceeding and following sibling of the node. This is necessary for the linked-list structure with the leaf nodes form, and allows for searching along the leave nodes for a range of values. The header also includes information about the number of keys in the node, and whether the node is a leaf or internal node. By including this information we can more efficiently pack more information into the nodes by following a strict schema for both internal and leaf nodes and not requiring delimiter values between items within the nodes.

Header	Child	Child	Child	Key	Key
25 B	8 B	8 B	8 B	$(K+2)*8$ B	$(K+2)*8$ B

(a) Internal Node

Header	Key	Key	Key	Key
25 B	$(K+2)*8$ B	$(K+2)*8$ B	$(K+2)*8$ B	$(K+2)*8$ B

(b) Leaf Node

Figure 12: Layout of the internal and leaf pages of the B+tree.

Figure 12 details the structure of both the internal and leaf nodes. The internal nodes contains the 25 byte header, followed by the references to the children nodes, followed by the keys which sort the children nodes. The leaf node contains the 25 byte header, followed by the keys. Since the header contains information about the number of keys in the nodes, it is possible in the internal node and the leaf node to directly navigate to specific keys in the node by calculating an offset value based on the size of the keys and the ordered position of the desired key.

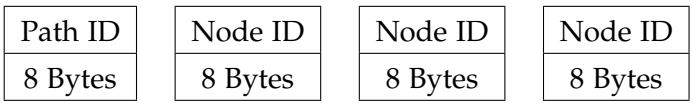
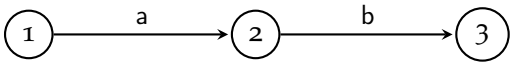


Figure 13: Structure of an uncompressed key for a path with two edges.

5.5 FAST INDEX INITIALIZATION

(a) Two separate paths of length one.



(b) Path of length two constructed from the paths above.

Figure 14: Two paths of length one, where a path of length two can be constructed by concatenating these edges together on the common node labeled "2".

For the path index to be used it must first be constructed by identifying and storing all of the paths which are desired to be indexed. In this work, the full index is constructed containing all possible paths up to length K. The method used to enumerate all the paths of length K is scalable to handle any size of K. However, in this implementation, the largest supported K for value for this operation is 3.

In modestly small graphs, with only a few million nodes and edges, the number of paths up to length 3 can still be quite large. The chosen method for enumerating all possible paths can have a large effect on the time required to complete this initialization operation. One method for enumerating all possible paths is to perform a query on the database for a path of the desired length with no bindings on nodes and edges along the path. This method has a number of problems. The first is that the database is not optimized for this type of query, and enumerating through the entire result set is a slow oper-

ation. However this disadvantage is small compared to the problem or the sort order of the paths. When querying the database for all K-length paths, the results will not be in the correct sort order for storing in the index. This necessitates sorting the keys such that bulk-loading of the B^+tree can be done.

An alternative, and the implementation used in this work, is to iteratively build up the index in a bootstrapping fashion. This is done by building larger K paths using smaller paths which have already been indexed. The process begins with the smallest paths of length one. These paths cannot be built by the index and must be retrieved by the database. Fortunately this database offers a fast operation for enumerating all edges in the graph, which is exactly what the paths of length one are (the set of all edges). We enumerate all of the paths of length one, and perform an external memory merge sort on these paths, bulk loading them into the B^+tree index. The external sorting implementation and bulk index construction is discussed in more detail in Section 5.5.1. We also store the inverse of these paths of length one by reversing the edge and storing that value in the K1 index.

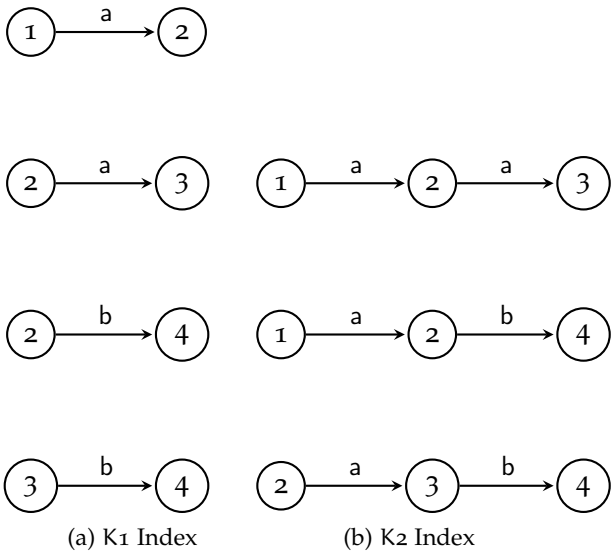
With the K1 index constructed, the K2 index can be constructed by doing a concatenation of all paths of length one where the paths have a common node between them at the appropriate ends of the path. An small example of this is illustrated in Figure 14, where there are two paths of length one with a common node between them at the correct ends. The top edge has its ending node with ID 2, and the bottom edge has its starting node with ID 2. These two edges can then be merged in the K2 index to form the path with edge labels a, b and nodes 1, 2, and 3. Since the inverse of the K1 edges are also stored in the index, the paths of length two with an inverse edge in the first, second, or in both positions will also be enumerated. The inverse edges are not shown in Figure 14.

Building the index in this way is not only faster than using the database to enumerate all the paths, but since we store the information about all nodes along the path, the concatenated paths are also constructed in sorted order. Beginning with the K1 index and building the K2 Index, the edges from the K1 index are examined in sorted order, and merging attempts are made with other paths also by enumerating through all of the paths, in sorted order.

It is important to iterate over the smaller indexes in the correct order to preserve the sort ordering with the larger index being constructed. The iteration order is shown in Figure 16.

5.5.1 *Bulk Index Construction*

Many of todays datasets already exceed the amount of main memory in an individual computer. As datasets and memory sizes increase, datasets will likely continue to outpace the growth of available main



(a) K1 Index (b) K2 Index

Figure 15: Building the K2 index using the K1 index, while remaining sorted.

```
for Each PathA in the K1 Index do
    for Each PathB in the K1 Index do
        for Each keyA of PathA do
            for Each keyB of PathB where starting node of keyB
            matches ending node of keyA do
                Insert merged path in K2 Index;
            end
        end
    end
end
```

Figure 16: Algorithm for the iteration order for building the K2 index from the K1 index and maintaining sorted order.

memory. While this work focuses on the performance of a path index, it is crucial that constructing the index be possible in a reasonable amount of time. This is both for the usability of the index in general but also and for the feasibility of constructing the indexes on the datasets in Chapter 6.

Inserting a full dataset into a B$^+$tree one-by-one is a known suboptimal solution [9]. This causes the B$^+$tree to sort the input, and the time spent doing a tree traversal on each insert and splitting when necessary can be very slow compared to the alternative of bulk-loading the index. To bulk load the index, the input must first be sorted. To achieve this, an external merge sort algorithm is implemented to handle the sorting of the paths in the cases where the number of paths is large enough to not fit within main memory. The algorithm is a standard implementation of the external memory merge sort [15], where the comparison of the sorted values uses our lexicographical order-

ing of the different values within the keys. The merging algorithm has a tunable parameter, known as the *fan in*. This value determines how many sorted sets are merged together in each pass. The *fan in* of the merging process is chosen by experimenting with different values and choosing the value resulting in the fastest time in our benchmarks. These benchmarks are shown in Table 6, and the best value in the tests with one million and ten million keys is a fan in value of four.

Compression of the keys is not applied during this phase of building the index, as the compression scheme is only able to produce any significant compression when the data is in sorted order. More information about the compression scheme is provided in Section 5.6. Without compression, the set of keys can require a significant amount of space. However, by iterating over the index of the smaller path lengths and building the larger indexes in sorted order, this sorting only needs to be done when initially building the K1 index.

After sorting the keys, the output of the sorting operation are the keys sorted and stored in the blocks of memory which are used as the leaves of the B$^+$tree. The upper leaves of the B$^+$tree above the leaf level are then built. This is done iteratively, building levels of internal nodes of the B$^+$tree until a level is reached where only one internal node is needed. This node then represents the root of the tree.

	Sort Time	
Fan In	1,000,000 Keys	10,000,000 Keys
2	3.0 seconds	3.1 seconds
4	3.0 seconds	3.0 seconds
8	3.4 seconds	3.3 seconds
16	3.4 seconds	3.5 seconds
32	3.7 seconds	3.8 seconds
64	3.8 seconds	3.9 seconds

Table 6: Sorting time with different values for the *fan in* of the external memory merge sorting algorithm.

5.6 COMPRESSION TECHNIQUES

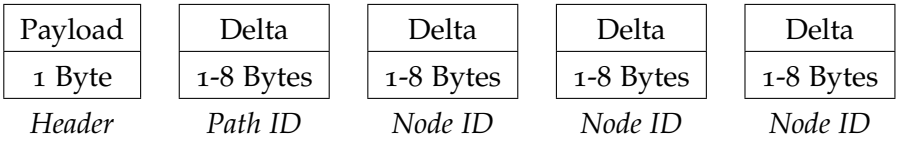


Figure 17: Structure of a compressed key for a path with two edges.

The constituent elements of the keys are all standard 8 byte long values, where the first value is always the path id and the following $K + 1$ elements are node ids. These elements can be termed $value_1$, $value_2$, $value_3$, $value_4$, ... Within the index, keys are sorted lexicographically by ($value_1$, $value_2$, $value_3$, $value_4$). This ordering causes neighboring keys to be similar. Most keys have the same values in $value_1$ and $value_2$ in particular, since many neighboring keys have the same path ids and the same starting node id along the path. This is similar to the situation in [19] on efficiently storing RDF triples, and allows for a similar compression scheme. This technique borrows ideas from inverted lists in text retrieval systems. The compression method involves not storing the full key. Instead, the changes between keys is what is stored. This results in a high compression, as the changes between keys is much smaller than the full key itself.

For each value in the key, the delta to obtain this key from the previous value is calculated. Once each delta is obtained, the minimum number of bytes necessary to store the *largest* delta for this key is found. Each delta is then truncated to only that minimum number of bytes. A header byte contains the value representing the number of bytes for each delta. Each delta requires between 1 byte and 8 bytes, depending on the size of the delta. This index does not store duplicate keys, so there is never the case where all delta values can require zero bytes.

Often, the prefix between keys can be identical to the previous key, while the final value in the key can require a large delta. In the compression scheme above, we would allocate the number of bytes to store the large delta, but the delta for the first few values would be zero. The maximum number of bytes needed to store deltas is 8 (when the delta requires the full 8 bytes), and the minimum number of bytes is 1. These seven different values can be stored in 3 bits, leaving 5 additional bits unused. To compress even more, the first 5 bits in the header can be used to signal when the corresponding value has a delta of zero, essentially forming a gap in the series of deltas stored for this key. By signaling this in the header, we can avoid writing the delta for that value altogether, and only write the values which has a non-zero delta. In this implementation, we only consider the first two deltas for header bit encoding. We call these bits *gap* bits, as they indicate a small gap in key where the delta is zero and not written.

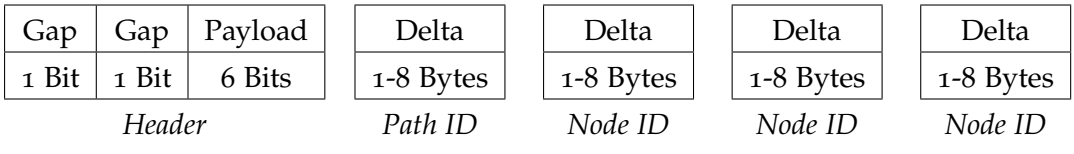


Figure 18: Structure of a compressed key with gap bits for a path with two edges.

In [19], the comparison between bit-level versus byte-level compression has been made. The conclusion the authors draw is the excessive CPU cost of doing bit-level compression is not worth the size savings compared to doing only byte-level compression. In many situations, the byte-level compression even results in better compression sizes due to the nature of the values stored in the keys— namely, the typically small and increasing difference between values.

Compression is only applied to individual leaf pages, never across pages. Compressing larger portions would produce a smaller overall index, but at a cost of usability. By only compressing individual pages, the design of the index does not change from a normal B^+tree. We can still traverse to any leaf node and immediately begin reading keys. If larger portions of the index were compressed together, then those additional portions would need to be fetched and decompressed before beginning to read keys.

Compression is also not applied to pages representing internal nodes in the B^+tree. Internal nodes in the tree account for a much smaller share of the total number of pages in the tree, as most pages are leaves. Further, there is the assumption that internal pages will be accessed often during traversals, and the additional decompression time on these pages does not justify the possible space savings compressing the pages would yield.

LUBM Dataset - Comparison of Compression Size

Index	Uncompressed	LZ4	Path Index
K1	0.16 GB	0.053 GB	0.02 GB
K2	15.99 GB	3.67 GB	1.69 GB

Table 7: Benchmark of compression techniques comparing the compressed data size.

Using this compression technique results in significantly reduced index sizes. This compression method works better than using generalized compression algorithms in terms of speed and implementation clarity. The LZ4[2] compression library is used to compare the compression scheme described above. LZ4 is a lossless compression algorithm related to the LZ77 type compression schemes. LZ4 is fast, but it does not compress as well as other compression algorithms such as gzip or bzip2. A comparison of the size of resulting indexes from each compression technique is shown in Table 7, and a comparison of the speed of the compression techniques is shown in Table 8. The comparison is done by inserting sequentially increasing keys into the index and measuring throughput time and final index size.

2 https://github.com/jpountz/lz4-java

LUBM Dataset - Comparison of Compression Time			
Index	Uncompressed	LZ4	Path Index
K1	< 1 Minute	4 Minutes	< 1 Minute
K2	28 Minutes	266 Minutes	27 Minutes

Table 8: Benchmark of compression techniques comparing compression speed, with times rounded to the nearest minute.

The comparison shows that the implementation done in this work beats the LZ4 algorithm is terms of speed and compression size. For the K2 index, the LZ4 compression algorithms runs in 226 minutes and compresses the index to 3.67 gigabytes, compared to the path index which runs in 27 minutes and compresses the index to 1.69 gigabytes. The implementation of the pages in the B^+tree do not allow both the LZ4 algorithm and the compression scheme used in this work to perform at their fastest speeds, but this comparison does show that relative to each other, our compressions scheme is better in both aspects.

6 EXPERIMENT SETUP

To evaluate our path indexing implementation we have conducted a number of experiments. In this chapter the goals of the experiments are discussed, the setup of the experiments is described, and the datasets used in the experiments are presented. A discussion of the obtained results follows in Chapter 7.

6.1 OBJECTIVE

The purpose of conducting these experiments is to answer the following questions:

1. How effective are path indexes for paths up to k-length at answering path queries of the same length?
2. How effective are path indexes for paths not up to k-length at answering path queries of a greater length by performing merge joins on sub paths?
3. How well can path indexes be built for different datasets of different graph density?
4. How expensive is the full path index initialization up to length k?
5. How expensive is path index initialization based on query workloads?

6.1.1 *Datasets*

Experiments are run on three different datasets. Each dataset comes from a different source and are of different sizes. Two datasets, the Lehigh University Benchmark (LUBM) dataset [11] and the Linked Data Benchmark Council (LDBC) dataset [2] are synthetic datasets, while the Advogato dataset [1] is a real-world dataset. It may be the case that certain properties of these datasets and their different sizes expose differences in effectiveness of path indexing.

6.1.2 *Queries*

The queries used here are taken directly or are derived from the queries supplied with the dataset. For the LDBC dataset, Cypher

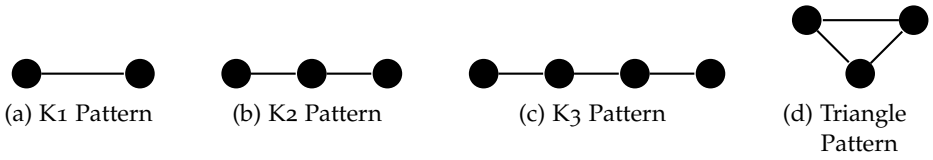


Figure 19: Different query patterns.

queries are supplied by Neo4j. However, these queries originally include filtering on certain node properties. In this work, these filters are excluded and the queries report all matching paths. For the LUBM dataset queries, the original queries supplied with the data generator are used as Cypher equivalents as used in [10]. For the Advogato dataset, we provide our own queries. The number of possible paths in the Advogato dataset is small since the number of edge labels is small. Therefore the consequence of choosing queries is assumed to be small.

Figure 19 depicts the different shapes the path queries take. In this work we focus on queries which can be represented as a path. In some cases, the end points of the path are the same node, leading to the triangle query pattern. We can also perform queries with a property filter on one or more nodes or edges along the path. Since these properties are not stored in the path index, it is necessary to check the property values of nodes or edges which are being filtered on by making requests to the Neo4j database. The aim of these experiments is to evaluate the performance solely of the path index, and therefore queries with node property filters are not focused on in these experiments. There is however one query in the LUBM experiments dataset with a node property filter, and it is included only for the sake of having the complete set of queries tested. In the LDBC queries however, these filters are removed since they are originally part of all of the queries. The discrepancy of including the filter in one query in the LUBM dataset and not including it in all the LDBC queries is not expected to influence the results of the experiments since the comparison will be done on a query-by-query basis with Neo4j. By not considering these filters, the experiments better highlight the performance of the path index in comparison to Neo4j. Considering these filters is a suggestion for future work, in Section 8.2.

6.2 SETUP

In this section details of the experiments are presented including what is measured, how that is measured, and in what environment the experiments are conducted.

6.2.1 *Measurement*

The measured data in these experiments is time. When comparing the path index to the database, the comparison is made by the difference in time to retrieve the same result sets. Shorter times indicate a faster and more efficient result set retrieval. Only the time needed to retrieve the results is compared. The time needed to open and close the database or index is ignored. For the database, we do not record the time needed to open and close a transaction event. For the index, we do not record the time needed to determine the path ID of the path to be searched. The total number of path IDs is small, and therefore in the experiments the path IDs are hardcoded into the query. Elapsed time is measured internally using Java's System.nanoTime() function, which according to the Java documentation provides the value of the most precise available system timer.

In the experiments on the full indexes, each query was executed 5 times per run, with 6 runs conducted. Between each run, the systems cache was flushed using the command *sync && purge*. In OSX this is the equivalent to doing *sync && drop_caches* on a unix system. On each run, the first execution is considered a "cold" run, with empty caches, and the following runs are considered "warm" runs, where the system cache is likely to provide better evaluation times. These results are kept separate. Once all the results are collected, we exclude 20% of the data points from the top and bottom tails of the data set. This removes the outliers from the dataset. We then report the mean computed from the remaining values.

6.2.2 *System Specifications*

All experiments are tested on a Apple Macbook Pro (MC721LL/A) with an Intel 2.0 GHz Core i7 (I7-2635QM), 8 GB of main memory, and a Samsung 850 EVO solid state disk drive capable of random read performance of 98,000 IOPS and random write performance of 90,000 IOPS. The operating system is OSX Yosemite, version 10.10.3 build 14D136.

All experiments were conducted using the latest version of Neo4j available at the time. The tested version is Neo4j 2.3.0-M01.

6.3 DATASETS

The experiments are conducted on three different datasets of different sizes and from different domains. These datasets are described in the following sections.

```
Original Query 7:
MATCH (x)-[:memberOf]->(z),
(x)-[:undergraduateDegreeFrom]->(y)<-[:subOrganizationOf]-(z)
RETURN ID(x), ID(y), ID(z)

Rewritten Query 7:
MATCH (x)-[:undergraduateDegreeFrom]->(y)<-[:subOrganizationOf]-(z)<-[:memberOf]-(x)
RETURN ID(x), ID(y), ID(z)
```

Figure 20: Rewriting of query 7 from [10] to be a single path used in the experiments in this paper.

6.3.1 *Lehigh University Benchmark*

The Lehigh University Benchmark (LUBM) is a data generation tool developed to perform evaluations on semantic web repositories in a systematic and controlled way [11]. The dataset is well-known and widely used for testing RDF systems. The data generated the LUBM tool models the university scenario. The dataset contains universities, departments, students, teachers, research groups, and publications. In these experiments, the data is generated with a scale factor of 50, meaning the dataset contains 50 universities from which to generate data from. The generated dataset contains approximately 6.8 million unique triples. This work follows the same data preparation steps taken by [10] to use this dataset with Neo4j, except the dataset is not enriched with inferred facts derived from the ontology rules. For example, nodes of type *Associate Professor* do not also get the more general label *Professor*, which is inferred in the dataset. LUBM is provided with 14 different queries, and in this paper only the 9 queries used by [10] are considered. Certain queries in the LUBM dataset are of variable length, meaning certain edges can be repeated a certain number of times and the path can be a range of lengths. In this paper, these variable queries are considered to be queries of different paths, and not studied. Instead these queries are rewritten to specify the exact path to match. In all cases, these variable number of edges are specified to only have one occurrence of the edge. The queries are also rewritten such that they can be represented as a single path, and unspecified edge directions are given a direction. For example, Query 7 as written in [10] is shown in Figure 20, and its rewriting in this paper is shown below it. The *memberOf* edge from node x to node z is appended to the second line, removing the need for two paths and instead forming a single path. The queries used are shown in Section A.1.

The number of nodes and edges in Lehigh University Benchmark is show in Table 12.

LENGTH	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
K1	X	X	X							
K2				X	X	X				
K3							X	X	X	X

Table 9: Length of each query for the LUBM dataset, where length is the number of edges in the query pattern.

6.3.2 *Social Network Benchmark*

The Social Network Benchmark is the first workload released by the Linked Data Benchmark Council, an independent authority responsible for specifying benchmarks [2]. The Social Network Benchmark models a social network similar to that of *Facebook*. The datasets contains persons and a friendship network that connects people together. The majority of the data is represented as messages posted by users. The queries for this dataset are also provided by the LDBC. Specifically in this paper we use the SNB-Interactive workload. This workload contains complex read only queries which have many filters and specifications on node and edge properties. In these situations, the requirements on properties are removed, and only the underlying path query is used for these experiments. After identifying the underlying path query, similar queries are removed from the test set. The final set of queries used is shown in Section A.3.

The number of nodes and edges in LDBC - Social Network Benchmark is shown in Table 12.

LENGTH	Q1	Q2	Q4	Q5	Q7	Q8	Q10	Q11
K1								
K2	X	X		X	X			
K3			X			X	X	X

Table 10: Length of each query for the LDBC-SNB dataset.

6.3.3 *Advogato*

The Advogato dataset is a trust network from the Advogato online community discussion board for developers of free software [1]. Advogato uses a trust metric to determine a single global trust value for each user. This value is computed by the *ratings* users give to each other. These ratings can be three possible values: *apprentice*, *journeyer*, and *master*. Advogato uses this trust to allow users to access certain administrative controls for the message board. The graph con-

structed from this dataset contains nodes which represent the users, and edges between nodes are the ratings users give to each other. Using a real dataset compared to the synthetic datasets is important in determining if our path index is applicable in these different situations. The Advogato dataset does not have specified queries. Therefore the queries were formulated to test the path index on paths of the greatest length indexed, where the path forms a complete loop. The queries used can be seen in Section A.2.

The number of nodes and edges in Advogato dataset is show in Table 12.

LENGTH	Q1	Q2	Q3	Q4	Q5
K1					
K2					
K3	X	X	X	X	X

Table 11: Length of each query for the Advogato dataset.

6.4 RESULT SET SIZES

Each dataset contains different sizes of result sets for each query. The Advogato dataset is the smallest, followed by the LDBC-SNB dataset. The table below shows the number of nodes and edges contained in each dataset, followed by the number of results found for each query of the datasets.

SET	LUBM	LDBC-SNB	ADVOGATO
Total Nodes	1,083,848	212,317	6,539
Total Edges	6,659,576	2,194,716	98,262
Q1	519,842	31,754	19,555
Q2	1	844,159	42,852
Q3	35,973	-	14,767
Q4	1,433,737	760,383	1,844
Q5	7,726,641	5,506,078	1,044
Q6	519,842	-	-
Q7	130	34,485	-
Q8	13,639	45,865	-
Q9	14,861	-	-
Q10	999	845,075	-
Q11	-	69,729	-

Table 12: Number of nodes and edges in each dataset and the size of the result set for each query of each dataset.

7 EXPERIMENTAL VERIFICATION

In this chapter the results of the conducted experiments are discussed. Additional details about the setup of the experiments can be found in Chapter 6. Section 7.1 shows the baseline speed of the index on performing insertion, deletion, and read operations. Section 7.2 shows the sizes of the constructed indexes. Section 7.3 gives the results of the experiments conducted using the full K-path index. Section 7.4 presents the result of the workload based experiments, with indexes constructed based on the query workload. Section 7.5 concludes with a discussion about the results found.

There are two types of experiments conducted. The first experiments use the full set of K1, K2, and K3 paths from the datasets. Using these full indexes, the queries for each dataset are evaluated using the indexes as well as with Neo4j. The second set of experiments are termed "Workload driven" where the indexes only contain the K1 paths from the datasets, and additional paths of length K2 and K3 are built at query evaluation time using the K1 index and performing joins to build up the larger paths. These workload driven experiments are designed to highlight the additional cost of selectively building the path index while providing improved query performance on subsequent queries of the same path.

7.1 BENCHMARK

	NUMBER OF KEYS		
Performance	1,000,000	10,000,000	100,000,000
Insertion (Micros/op)	8.5	6.8	6.8
Read (Micros/op)	8.6	5.1	5.2
Delete (Micros/op)	14.1	17.0	17.2

Table 13: Benchmark of the performance of the path index on basic I/O operations.

The first tests on the path index are the speed in which basic index operations can be completed. This index implements searching, insertion, and deletion of paths. These three operations are run on successively larger sizes of data to find the average time each requires. In this experiment, the data is generated randomly. Paths inserted into

the index are of length 3, therefore requiring four values to be stored in the keys being inserted. The results are shown in Table 13. The data shows that at 100 million keys, the index provides an insertion time of 6.8 microseconds per operation, a search time of 5.2 microseconds per operation, and 17.2 microseconds per operation.

7.2 INDEX CONSTRUCTION

For the first experiments with the full index of K1, K2, and K3 paths, the index building time is not reflected in the query evaluation time. The construction of these indexes is done in a separate operation. The time needed to build the indexes and the size of the indexes for all of the datasets are shown in Table 14.

	LUBM	
	SIZE(GB)	TIME(MIN)
K1	0.02	0.20
K2	1.69	27.28
K3	41.58	178.26
WORKLOAD	0.1	4.3

	LDBC	
	SIZE(GB)	TIME(MIN)
K1	0.0003	0.006
K2	3.94	21
K3	220.1	501
WORKLOAD	0.038	2.2

	ADVOGATO	
	SIZE(GB)	TIME(MIN)
K1	0.0003	0.002
K2	0.29	0.11
K3	4 3.6	5.1
WORKLOAD	0.0083	0.3

Table 14: Size and build time for K 1, 2 and 3 indexes on different datasets.

The K1 indexes for all datasets are small— the largest being 20 megabytes for the LUBM dataset. These indexes are also fast to build, the largest also being from the LUBM dataset and requiring 12 sec-

onds. However, the size and building time rapidly grows as longer paths are indexed. For the full K2 index, the LDBC-SNB dataset takes 3.94 gigabytes and 21 minutes to build. However, the K2 index for the LUBM index is smaller at 1.69 gigabytes. This rapid growth becomes even more problematic for the full K3 index, requiring 220.1 gigabytes for the LDBC-SNB dataset. This is again larger than the corresponding K3 index for the LUBM dataset, which requires 41.58 gigabytes. The Advogato dataset is smaller for all index sizes, with the K3 index requiring 3.6 gigabytes.

In addition to the indexes for the full K1, K2, and K3 paths, Table 14 also provides the size and building time of the indexes produced by the workload based experiments from section 7.4. These indexes contain the full K1 index as well as the paths and subpaths needed to evaluate the queries specified for each dataset. To build the K3 paths, three different paths from the K1 index must be joined together. When joining two paths, the sorted property of the paths allows for a fast merge join operation to be used. However, with three paths, there will be one join out of the three that cannot be done directly via a merge join. This is because the results from performing a join on two of the paths will not be in sorted order and able to be merge joined with the third path. There are two ways to perform a join on these results:

HASH JOIN: After joining two of the paths, a hash index must be built on the smaller of either the merged paths or the remaining third path using the ID of the node to be used to join the paths. After merging the results, insert the new paths into the index.

SORT MERGE JOIN: After joining two of the paths, sort the intermediate results, and then perform a join with the third path. After joining the results, insert the new paths into the index.

In this implementation, a modification of the second option is used. The intermediate results are sorted by inserting them into the index. Then, a new join operation is performed on these sorted paths which are now in the index, and they are joined with the third path. The new paths are also inserted into the index. Inserting the intermediate results into the index is a more expensive operation when considered in isolation, however it is cheaper for subsequent queries which can use intermediate merge operations when joining with another third path. Otherwise, this intermediate result would have to be recalculated. The time needed to do these operations is included in the time reported for each workload index in Table 14. We see that the largest workload index is on the LUBM dataset with a size of 10 megabytes.

7.3 FULL K-PATH INDEXES

The first experiments are on the full K-path indexes. The query evaluation time is affected by the caching done by the operating system

as well as by the Neo4j software. To accurately study and compare the query times of the path index, the experiments on the full K-path indexes are done with results with cold caches kept separate from results with warm caches. The term "cold cache" here means that, to the best of our efforts, the caches of the system do not contain any prefetched material or results which would speed up the performance of the query evaluation. The term "warm caches" then means that the queries have already been executed once, and these are the results from subsequent evaluations of the queries. As can be seen in the tables below, warm caches have a large impact on the query time for Neo4j.

The results show the time to the first result and the time to the last result. For both Neo4j and the path index, the time to the first result is measured as the time from immediately before Neo4j's or the path index's *find* operation is executed, and the time immediately after the first result is found. The time to the last results is measured as the time immediately before Neo4j's or the path index's *find* operation is executed, until the time immediately after the last result is found. Therefore the time spent iterating over the whole result set can be found by subtracting the time for the last results and the time for the first result.

For example, in Table 15, for Query 7 and Neo4j, the time to the first result is 2198 milliseconds after executing Query 7 in Neo4j. The time until the last result for Neo4j is 7610 milliseconds. Therefore it took 5412 milliseconds to find and return all of the results. This is in comparison to the K3 path index (the third column), where we see that the time to the first result is 23 milliseconds and the time to the last result is 91 milliseconds, requiring 68 milliseconds to find all of the results.

In addition to querying for a K path in the index, queries are also conducted by using only $K-1$ paths in the index. Again looking at Query 7 in Table 15, we see under the column labeled "Index K2(ms)" the time needed to evaluate Query 7 using the K2 and K1 subpaths of the query and joining the results. This tells us how long the query evaluation time would be if the index only had the smaller subpaths and not the full K3 path. The column "Index K1(ms)" for Query 7 is blank, as there is no data point here. These experiments only show the times needed to perform a single join to evaluate a given query. Evaluating Query 7 using only the K1 paths is possible, but would require joining two paths first, and doing a sort merge join with the third path or performing a hash join with the third path. In this implementation of the path index, hash joins are not implemented, and the sort merge join is explored further in the workload based query experiments.

7.3.1 *LUBM Dataset*

For the LUBM Dataset queries in the Cold Start setting in Table 15, we see that when using the best available index for each query the path index is able to provide a significantly better query evaluation time. For finding the first result, the path index provides at least a 77x speedup compared to the Neo4j. For finding the last result, the path index is able to provide at least a 5x speedup. In the best case, the path index provides a 1114x speedup to arrive at the first result in Query 3.

The Best Available index being that which contains the exact path, without requiring joins.

When the cache is warm, we see a large improvement in query evaluation time for both Neo4j and the path index. For all queries, the path index is able to return the first result in at least 3 milliseconds. Neo4j is also able to return the first result in single digit times for Queries 7, 8, 9, and 10. These queries have the smallest result set size, which likely accounts for this large improvement. For these queries, the path index does not provide a large speedup for finding the first result. However the path index still provides a 2x speedup for finding the last result in the worst case. In the best case, the path index is able to find the first result within 0.05 milliseconds, an 8260x speedup.

7.3.2 *LDBC-SNB Dataset*

The LDBC-SNB dataset experiments provide similar results from the LUBM experiments. The path index is able to provide a significant speedup to all queries, especially in the cold start. The best result is for Query 1, where the path index is able to return the first result in 6 milliseconds, a 507x speedup. In Query 7, the path index returns the first result in 17 milliseconds for the lowest speedup of 15x. With respect to the time needed to return the last result, the path index achieves its best result with a 175x speedup in Query 7, and its worst result in Query 10 with a speedup of 8x.

Similar again to the results in the LUBM Warm experiments, the results for the LDBC Warm experiments still achieve a significant speedup compared to Neo4j. The smallest speedup in finding the first result occurs in Query 7, with a speedup of 35x. The largest speedup in finding the first result occurs in Query 5, with a speedup of 224x. For finding the last result, the smallest speedup occurs in Query 4 with a speedup of 9x, and the largest speedup occurs in Query 7 with a speedup of 92x.

7.3.3 *Advogato Dataset*

The Advogato dataset is the smallest of the three used in this paper. The results from these experiments show that this smaller dataset still achieves improvements in the cold start situation, although the

LUBM - Cold Cache

		Neo4j (ms)	Index K3(ms)	Index K2(ms)	Index K1(ms)	Speedup
Query 1	First Result	5333	-	-	6	888x
	Last Result	7305	-	-	91	80x
Query 2	First Result	4667	-	-	6	777x
	Last Result	4668	-	-	9	518x
Query 3	First Result	5574	-	-	5	1114x
	Last Result	6724	-	-	20	336x
Query 4	First Result	5976	-	25	213	239x
	Last Result	9414	-	375	10833	25x
Query 5	First Result	5644	-	23	107	245x
	Last Result	19739	-	1521	4796	12x
Query 6	First Result	5665	-	25	108	226x
	Last Result	7780	-	172	2963	45x
Query 7	First Result	2198	23	26	-	95x
	Last Result	7610	91	261	-	83x
Query 8	First Result	1866	24	24	-	77x
	Last Result	11275	2130	573	-	5x
Query 9	First Result	1813	22	25	-	82x
	Last Result	6637	40	252	-	165x
Query 10	First Result	1805	22	22	-	82x
	Last Result	6635	26	59	-	255x
Average	First Result	4054	22	24	5	382x
	Last Result	8778	571	689	40	152x

Table 15: Query Evaluation time to retrieve the first result and the last result in Neo4j and in the Path Index with cold caches.

LUBM - Warm Cache

		Neo4j (ms)	Index K3(ms)	Index K2(ms)	Index K1(ms)	Speedup
Query 1	First Result	480	-	-	0.19	2526x
	Last Result	2080	-	-	37	56x
Query 2	First Result	2014	-	-	1	2014 x
	Last Result	2014	-	-	1	2014x
Query 3	First Result	413	-	-	0.05	8260x
	Last Result	1352	-	-	4	338x
Query 4	First Result	774	-	0.8	173	967x
	Last Result	3741	-	112	10932	33x
Query 5	First Result	457	-	2	45	228x
	Last Result	13303	-	1439	4645	9x
Query 6	First Result	437	-	2	47	218x
	Last Result	2225	-	107	2831	20x
Query 7	First Result	8	2	2.4	-	4x
	Last Result	2221	32	179	-	69x
Query 8	First Result	1	1	2	-	1x
	Last Result	5319	1992	493	-	2x
Query 9	First Result	1	2	2	-	0.5x
	Last Result	1378	8	179	-	172x
Query 10	First Result	1	3	2	-	0.3x
	Last Result	1392	4	16	-	348x
Average	First Result	458	2	1	<1	1444x
	Last Result	3502	509	552	14	306x

Table 16: Query Evaluation time to retrieve the first result and the last result in Neo4j and in the Path Index with warm caches.

LDBC-SNB - Cold Cache

		Neo4j (ms)	Index K3(ms)	Index K2(ms)	Index K1(ms)	Speedup
Query 1	First Result	3044	-	6	25	507x
	Last Result	3374	-	24	274	140x
Query 2	First Result	2975	-	5	24	595x
	Last Result	4721	-	137	380	34x
Query 4	First Result	2893	26	25	-	111x
	Last Result	5004	556	441	-	9x
Query 5	First Result	2934	-	6	26	489x
	Last Result	12343	-	705	766	17x
Query 7	First Result	261	-	17	37	15x
	Last Result	3153	-	18	715	175x
Query 8	First Result	2746	105	45	-	26x
	Last Result	3551	141	936	-	25x
Query 10	First Result	2960	183	26	-	16x
	Last Result	5326	600	483	-	8x
Query 11	First Result	2838	21	22	-	135x
	Last Result	3297	65	419	-	50x
Average	First Result	2581	83	8	-	236x
	Last Result	5095	340	221	-	57x

Table 17: Query Evaluation time to retrieve the first result and the last result in Neo4j and in the Path Index with cold caches.

LDBC-SNB - Warm Cache

		Neo4j (ms)	Index K3(ms)	Index K2(ms)	Index K1(ms)	Speedup
Query 1	First Result	208	-	<1	3	208x
	Last Result	292	-	5	181	58x
Query 2	First Result	217	-	<1	3	217x
	Last Result	1573	-	69	288	22x
Query 4	First Result	238	2	3	-	119x
	Last Result	1892	202	363	-	9x
Query 5	First Result	224	-	<1	3	224x
	Last Result	8855	-	455	649	19x
Query 7	First Result	106	-	3	7	35x
	Last Result	371	-	4	627	92x
Query 8	First Result	99	2	8	-	49x
	Last Result	470	17	836	-	27x
Query 10	First Result	235	2	3	-	117x
	Last Result	2210	219	385	-	10x
Query 11	First Result	211	2	3	-	105x
	Last Result	391	23	344	-	17x
Average	First Result	192	2	1	-	134x
	Last Result	2006	115	133	-	31x

Table 18: Query Evaluation time to retrieve the first result and the last result in Neo4j and in the Path Index with warm caches.

Advogato - Cold Cache

		Neo4j (ms)	Index K3(ms)	Index K2(ms)	Index K1(ms)	Speedup
Query 1	First Result	2097	17	6	-	123x
	Last Result	3359	359	504	-	9x
Query 2	First Result	2295	23	7	-	99x
	Last Result	7058	2093	1360	-	3x
Query 3	First Result	2636	22	6	-	119x
	Last Result	5376	931	730	-	5x
Query 4	First Result	2559	31	7	-	82x
	Last Result	4791	564	511	-	8x
Query 5	First Result	2315	23	6	-	100x
	Last Result	3578	441	668	-	8x
Average	First Result	2380	23	-	-	104x
	Last Result	4832	877	-	-	6.6x

Table 19: Query Evaluation time to retrieve the first result and the last result in Neo4j and in the Path Index with cold caches.

speedup in the warm start times are modest. This is not unusual, considering that Neo4j is able to achieve very good results by extensive use of caching.

For the cold stat situation, the path index is able to achieve its largest and smallest speed up for finding the first result on queries 1 and 4 with speedups of 123x and 82x, respectively. For finding the last result, the path index achieves its largest and smallest speedups on queries 1 and 2 with speedups of 9x and 3x.

7.4 WORKLOAD DRIVEN INDEXES

As Table 14 shows, the full K3 index is much too large to be used in production, and requires a significant amount of time to build. It would be desirable to have small indexes which can be built quickly, but are still able to provide the same significant improvements to query evaluation time. To that end, these experiments are conducted to determine how well indexes can be built which only contain the requisite data to evaluate the queries specified for each experiment. The workload indexes are built at runtime, where the necessary K1 paths are joined to form the paths in the queries, or joined a third time to form the paths of length 3. For each dataset, there are two tables showing information about the experiment in relation to that dataset. The first table details the building of the index. The times for finding and merging the results are reported separately from the time needed

Advogato - Warm Cache

		Neo4j (ms)	Index K3(ms)	Index K2(ms)	Index K1(ms)	Speedup
Query 1	First Result	3	3	< 1	-	1X
	Last Result	501	165	224	-	3X
Query 2	First Result	2	2	< 1	-	1X
	Last Result	3178	1880	1054	-	1.7X
Query 3	First Result	2	3	<1	-	0.6X
	Last Result	1341	790	456	-	1.7X
Query 4	First Result	2	3	< 1	-	0.6X
	Last Result	812	332	211	-	2X
Query 5	First Result	7	2	< 1	-	3X
	Last Result	442	127	206	-	3X
Average	First Result	3	2	-	-	1X
	Last Result	1254	658	-	-	2X

Table 20: Query Evaluation time to retrieve the first result and the last result in Neo4j and in the Path Index with warm caches.

to insert those results into the index, with the total time also shown in a different column of the tables. Finally the query evaluation time is shown for each query after inserting those paths into the index. The second table compared the query time to find the final result in the result set in Neo4j and in the path index after having inserted that path in the path index. This second table also shows the time total to build those paths for the index, and the speedup the path index provides compared to Neo4j after the paths have been inserted into the path index.

7.4.1 *LUBM Dataset*

The LUBM dataset shows that building the paths for certain queries can take a significant amount of time. For example, Query 5 from Table 22 takes the longest time at 129499 milliseconds, or 2.1 minutes. This is because this query has the largest result set size out of all of queries of all datasets. However, in Table 23 the speedup of Query 5 in the path index is 17x in comparison to Neo4j. Query 7A provides even more promising results, where building the index requires only 769 milliseconds, and subsequent queries in the path index can be evaluated in less than 1 millisecond. Query 7A can be built so quickly because it uses the same intermediary results from Query 6, as well as having the smallest result set size in this dataset.

LUBM

	Query Plan
Query 4	takesCourse $\bowtie$ teacherOf^{-1}
Query 5	memberOf $\bowtie$ subOrganizationOf^{-1}
Query 6	memberOf $\bowtie$ subOrganizationOf
Query 7A	undergraduateDegreeFrom $\bowtie$ Query 6^{-1}
Query 7B	P_{7B} = subOrganizationOf^{-1} $\bowtie$ memberOf^{-1}
	undergraduateDegreeFrom $\bowtie$ P_{7B}
Query 8A	hasAdvisor $\bowtie$ Query4^{-1}
Query 8B	P_{8B} = teacherOf $\bowtie$ takesCourse^{-1}
	hasAdvisor $\bowtie$ P_{8B}
Query 9	P_9 = worksFor $\bowtie$ subOrganizationOf^{-1}
	headOf^{-1} $\bowtie$ P_9
Query 10	P_{10} = worksFor $\bowtie$ subOrganizationOf
	headOf^{-1} $\bowtie$ P_{10}

Table 21: Workload experiment with paths constructed from the K1 index with joined results inserted into the index.

LUBM

	Result Set Size	Search Time (ms)	Insertion Time (ms)	Total (ms)	Subsequent Search (ms)
Query 4	1433737	10317	19972	30289	119
Query 5	7726641	5001	124497	129499	775
Query 6	519842	2685	8427	11113	39
Query 7A	130	768	1	769	<1
Query 7B	519842	253	7888	8141	38
	130	7689	2	7691	<1
Query 8A	13639	736	100	836	2
Query 8B	1433737	787	126	913	2
	13639	109	1680	1790	2
Query 9	534980	235	8427	8662	37
	14861	13	132	145	2
Query 10	35973	67	739	806	2
	999	8	8	16	<1

Table 22: Workload experiment on the LUBM dataset with paths constructed from the K1 index with joined results inserted into the index.

LUBM - Time to Last Result

	Neo4j Query Warm (ms)	Path Index - Building (ms)	Path Index - Query (ms)	Speedup
Query 4	3741	30289	119	31x
Query 5	13303	129499	775	17x
Query 6	2225	11113	39	57x
Query 7A	2221	769	<1	2221x
Query 7B	2221	15832	<1	2221 x
Query 8A	5319	836	2	2659x
Query 8B	5319	2703	2	2659x
Query 9	1378	8807	2	689x
Query 10	1392	822	<1	1392x
Average	4124	22296	104	1327x

Table 23: A comparison of the query evaluation time of Neo4j and the Path Index on the LUBM dataset using the workload based approach for building the index.

7.4.2 *LDBC-SNB Dataset*

The results from the workload experiments on the LDBC-SNB dataset show that building the index of the paths of the queries can be done quickly and provide significant speedups in subsequent queries. In the worst case, it can be seen in Table 26 that Query 5 requires 82252 milliseconds to build and insert the paths into the index, with subsequent queries being completed in 436 milliseconds. This cost of building the index is large compared to the 8855 milliseconds Neo4j requires to evaluate the query, however the 20x speedup achieve in subsequent queries quickly makes this cost worthwhile.

7.4.3 *Advogato Dataset*

The Advogato dataset, being the smallest dataset tested, provides the best results in the workload experiments. The time needed to build the index for a query is 6242 milliseconds in the worst case on Query 2, and subsequent queries in the path index provide a 454x speedup compared to Neo4j.

7.5 DISCUSSION

We have seen from the results of our experiments how the full K3 path index greatly reduces the query evaluation time for all path queries compared to the time needed to evaluate the queries in Neo4j. In all experiments and all queries, the time to the first result in the result

LDBC-SNB

	Query Plan
Query 1	KNOWS $\bowtie$ PERSON_IS_LOCATED_IN
Query 2	KNOWS $\bowtie$ POST_HAS_CREATOR^{-1}
Query 4	Query 2 $\bowtie$ POST_HAS_TAG
Query 5	KNOWS $\bowtie$ HAS_MEMBER^{-1}
Query 7	POST_HAS_CREATOR^{-1} $\bowtie$ LIKES_POST^{-1}
Query 8	P_8 = REPLY_OF_POST^{-1} $\bowtie$ COMMENT_HAS_CREATOR
	POST_HAS_CREATOR^{-1} $\bowtie$ P_8
Query 10	KNOWS $\bowtie$ Query 1
Query 11	P_{11} = WORKS_AT $\bowtie$ ORGANIZATON_LOCATED_IN
	KNOWS $\bowtie$ P_{11}

Table 24: Query Plan for the workload experiments for the LDBC Dataset.

LDBC-SNB

	Result Set Size	Search Time (ms)	Insertion Time (ms)	Total (ms)	Subsequent Search (ms)
Query 1	31754	250	422	672	5
Query 2	844159	317	10447	10764	74
Query 4	760382	4958	5856	10815	76
Query 5	5506078	948	81303	82252	436
Query 7	34485	545	515	1060	2
Query 8	45865	264	681	945	3
	45865	503	334	837	4
Query 10	845075	215	6095	6311	394
Query 11	6125	37	89	126	<1
	69729	245	501	746	6

Table 25: Workload experiment on the LDBC-SNB dataset with paths constructed from the K1 index with joined results inserted into the index.

LDBC-SNB - Time to Last Result

	Neo4j Query Warm (ms)	Path Index - Building (ms)	Path Index - Query (ms)	Speedup
Query 1	292	672	5	58x
Query 2	1573	10764	74	21x
Query 4	1892	10815	76	24x
Query 5	8855	82252	436	20x
Query 7	371	1060	2	185x
Query 8	470	1782	4	117x
Query 10	2210	6311	394	5x
Query 11	391	872	6	65x
Average	2006	14361	124	61x

Table 26: A comparison of the query evaluation time of Neo4j and the Path Index on the LDBC-SNB dataset using the workload based approach for building the index.

Advogato

	Query Plan
Query 1	P_1 = apprentice ⋈ apprentice
	apprentice ⋈ P_1
Query 2	P_2 = journeyer ⋈ journeyer
	journeyer ⋈ P_2
Query 3	P_3 = master ⋈ master
	master ⋈ P_3
Query 4	P_4 = journeyer ⋈ master
	apprentice ⋈ P_4
Query 5	P_5 = apprentice ⋈ master
	apprentice ⋈ P_5

Table 27: Workload experiment with paths constructed from the K1 index with joined results inserted into the index.

Advogato

	Result Set Size	Search Time (ms)	Insertion Time (ms)	Total (ms)	Subsequent Search (ms)
Query 1	76290	100	890	990	12
	19555	105	157	263	3
Query 2	412935	214	4975	5190	41
	42852	699	352	1052	7
Query 3	218621	151	2632	2784	19
	14767	319	119	439	1
Query 4	218272	193	2739	2932	17
	1844	137	14	152	<1
Query 5	61128	95	790	885	7
	1044	118	10	129	<1

Table 28: Workload experiment on the Advogato dataset with paths constructed from the K1 index with joined results inserted into the index.

Advogato - Time to Last Result

	Neo4j Query Warm (ms)	Path Index - Building (ms)	Path Index - Query (ms)	Speedup
Query 1	501	1253	3	167x
Query 2	3178	6242	7	454x
Query 3	1341	3223	1	1341x
Query 4	812	3084	<1	812x
Query 5	442	1014	<1	442x
Average	1254	2963	2	643x

Table 29: A comparison of the query evaluation time of Neo4j and the Path Index on the Advogato dataset using the workload based approach for building the index.

set is faster using the index. Iterating through the results also shows that in the path index all the results in the result set can be found faster compared to iterating through the results in Neo4j. Even as the size of the index grows to sizes many times larger than the original graph, the time needed to search in the index to find the results can be done faster than Neo4j.

When comparing the evaluation times with cold caches, the index greatly outperforms Neo4j. For example in the LUBM dataset, the index is able to reach the first results in all queries in less than 30 milliseconds, compared to the minimum of 1800 milliseconds needed for the time to the first result in Neo4j. With warm caches, Neo4j is able to return results with much better times, reducing the time to the first result in the LUBM dataset to 1 millisecond in best case, however, the index is able to achieve sub-millisecond execution time in those cases. Further, even in the cases where Neo4j is able to find the first result is approximately the same time as the index, the index is much faster at iterating through the result set.

The results also show that performing a search on the index for path of length K by merging the results from the K-1 and K-2 paths, the time to the first result still outperforms Neo4j, and in cases where the result set size is small, outperforms the actual K index. The cases where joining on the smaller results having better execution times than directly using the K index is unexpected, but can be attributed to the smaller indexes being able to better fit in memory compared to the larger index. Further, the differences in execution time in real terms is small.

The size of the path indexes was expected to be large, however our results show that building these indexes by including all paths up to K3, the size of the index becomes a limiting factor to the usability of the index. While the index sizes may be large, the evaluation time for paths using the index remains small, highlighting the performance of the index. The size of the indexes is a measure of the number of K length paths. This is effected by the number of nodes and edges in the graph, and the density of the graph. Tightly connected graphs have more paths than less dense graphs, and this explains the increase in the index size between the LUBM dataset and the LDBC dataset. While the LDBC dataset is smaller in terms of nodes and edges compared to the LUBM dataset, the density is much higher.

Although the full indexes are large, the workload experiments show more promise by being much smaller than the original index. The initial loading of the index on the first evaluation of the query requires building the path from the K1 index or from previously seen smaller paths. In most cases, this requires more time than Neo4j to evaluate the query. However, subsequent queries on these paths are able to be evaluated orders of magnitude faster than Neo4j.

8 CONCLUSION

The goal of this thesis has been to develop the necessary systems to improve path query performance in the graph database Neo4j. The design of the system produced can be applied to graph databases in general, and is not limited to one specific system. The first step towards this goal has been the design of a path index. This design has chosen simplicity where possible, drawing on decades of research from the literature on indexing in relational databases. The main theoretical contributions are found in Chapter 4, where the design of the path index is presented. Chapter 5 presents the implementation of this path index design. An empirical evaluation of this implementation is first outlined in Chapter 6 with the results presented in Chapter 7. This thesis ends with a conclusion in Section 8.1, and an outline for future work in Section 8.2.

8.1 OVERVIEW

Evaluating queries efficiently is the driving force behind the construction and adoption of many of the new NoSQL databases. With ever increasing amounts of data being generated, these new systems provide a way to store, query, and analyze data quickly. Graph databases in particular offer new features for storing and querying data in a way that was not possible without significant work in traditional relational databases. The flexibility that graph databases provide occasionally results in poor query evaluation performance. The goal of this work has been to improve the performance of path queries by using a simple indexing solution commonly found in other database systems.

This thesis has focused on one fundamental type of query, the path query, and its performance in the graph database Neo4j. As discussed in Section 3.4 and identified the work by [10], path query evaluation can perform poorly in Neo4j. This thesis presents a new and simple solution to improve path query performance. The solution developed here uses a simple B^+tree index to store paths from the graph in such a way that they can be quickly found again at query evaluation time. This implementation is accompanied with tools to load the index with the paths from the graph in an efficient way. The complete codebase with the path index implementation is available as open source [1] for further research and development

1 https://github.com/jsumrall/Path-Index

An empirical study using three different datasets has been conducted on this path index implementation to identify what performance improvements path indexing provides a graph database. These experiments show that in every query for each dataset, path indexing provides a significant improvement in path query evaluation time. In a limited number of cases, the graph database alone performs as well as this path index. However in an overwhelming majority of the queries, the path index is able to provide at least a 2x speedup in query time. In the best cases, the path index is able to provide a 8000x speedup to the graph database.

Experiments have also been conducted where the index is initially built only with the K1 paths, and as queries are encountered the results for the query are constructed by performing joins on the K1 subpaths until the full path is available. Those subpaths are also stored in the index, able to be used for future queries with identical subpaths. The results from these experiments show that the additional time to build the query results is relatively large, but the subsequent query times on the index again provide significant speedups compared to the graph database. Further, these workload based indexes are multiple orders of magnitude smaller than the full K1, K2, and K3 indexes.

We also identify situations where data in the path index can be compressed. By compression only at the page level, the path index is still able to have a significant size reduction. By only compressing at the page level, traversals in the index can be done without decompressing more than the pages necessary to perform any operation. The compression applied in this index works by only storing the deltas between values, which are usually small due to the keys in the graph being sorted. For the K2 index on the LUBM dataset, this compression scheme reduces the index size from 15.99GB to 1.69GB, a 9.5x reduction.

8.2 FUTURE WORK

This work performed experiments on three varied datasets, which establishes the performance benefits path indexing provide. Future work can include more datasets, using queries or various path lengths beyond K3.

The query workload based experiments show the most promise in terms of index size, index construction time, and query performance. Building these types of path indexes is the natural progression of this work. Additional experiments should be conducted to identify how to best build the index based on encountered queries. Ideas for this include examining query logs and building indexes based on frequent queries. The index can also be built automatically by developing a system to deconstruct query paths and identifying the subpaths in the query, and inserting those subpaths into the index.

The queries tested in this work were simple path queries. Additional work can be done with more complex path queries, with conditional filtering on certain properties of the nodes along the path. The original queries for the LDBC-SNB dataset include these types of conditions. Including those qualifiers to the over the path index is an important topic for further study.

BIBLIOGRAPHY

[1] Advogato network dataset – {KONECT}, October 2014.

[2] Renzo Angles, Ioan Toma, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, and Venelin Kotsev. The linked data benchmark council. *ACM SIGMOD Record*, 43(1):27–31, May 2014.

[3] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*, page 134, San Diego, California, USA, June 2003. ACM Press.

[4] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[5] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th International Conference on Very Large Data Bases - VLDB '01*, pages 341–350, Roma, Italy, September 2001. Morgan Kaufmann Publishers Inc.

[6] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, volume 14, page 1, Boston, Massachusetts, USA, June 1984. ACM Press.

[7] DB Engines. DB-Engines Ranking, Graph Databases, 2015.

[8] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB '97 Proceedings of the 23rd International Conference on Very Large Data Bases - VLDB '97*, pages 436–445, Athens, Greece, August 1997. Morgan Kaufmann Publishers Inc.

[9] Sven Groppe. Data Management and Query Processing in Semantic Web Databases. Chapter 3. Springer, Berlin, Heidelberg, January 2011.

[10] Andrey Gubichev and Manuel Then. Graph Pattern Matching. In *Proceedings of Workshop on GRAph Data management Experiences and Systems - GRADES'14*, pages 1–7, Snowbird, Utah, USA, June 2014. ACM Press.

[11] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, October 2005.

[12] Xifeng Yan Han and Jiawei. *Managing and Mining Graph Data: Chapter 5, Graph Indexing*, volume 40 of *Advances in Database Systems*. Springer US, Boston, MA, 2010.

[13] H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. *ACM SIGMOD Record*, 29(2):403–414, June 2000.

[14] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings 18th International Conference on Data Engineering*, pages 129–140, San Jose, California, USA, 2002. IEEE Comput. Soc.

[15] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Chapter 5*. Addison Wesley Longman Publishing Co., Inc., January 1998.

[16] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, February 2010.

[17] Zhe Li and Kenneth A. Ross. Fast joins using join indices. *The VLDB Journal The International Journal on Very Large Data Bases*, 8(1):1–24, April 1999.

[18] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295, Jerusalem, Israel, January 1999. Springer-Verlag.

[19] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, September 2009.

[20] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, September 1995.

[21] Philippe Pucheral, Jean-Marc Thévenin, and Patrick Valduriez. Efficient Main Memory Data Management Using the DBGraph Storage Model. *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 683–695, August 1990.

[22] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '02*, page 39, Madison, Wisconsin, June 2002. ACM Press.

[23] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[24] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis. In *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*, pages 1–6, New York, New York, USA, June 2013. ACM Press.

APPENDIX A

A.1 LUBM CYPHER QUERIES

Query 1:
```
MATCH (x)-[:memberOf]->(y)
RETURN ID(x), ID(y)
```

Query 2:
```
MATCH (x)-[:memberOf]->(y)
WHERE x.URI = "http://www.Department0.University0.edu/UndergraduateStudent207"
RETURN ID(x), ID(y)
```

Query 3:
```
MATCH (x)-[:worksFor]->(y)
RETURN ID(x), ID(y)
```

Query 4:
```
MATCH (x)-[:takesCourse]->(y)<-[:teacherOf]-(z)
RETURN ID(x), ID(y), ID(z)
```

Query 5:
```
MATCH (x)-[:memberOf]->(y)<-[:subOrganizationOf]-(z)
RETURN ID(x), ID(y), ID(z)
```

Query 6:
```
MATCH (x)-[:memberOf]->(y)-[:subOgranizationOf]->(z)
RETURN ID(x), ID(y), ID(z)
```

Query 7:
```
MATCH (x)-[:undergraduateDegreeFrom]->(y)<-[:subOrganizationOf]-(z)<-[:memberOf]-(x)
RETURN ID(x), ID(y), ID(z)
```

Query 8:
```
MATCH (x)-[:hasAdvisor]->(y)-[:teacherOf]->(z)<-[:takesCourse]-(x)
RETURN ID(x), ID(y), ID(z)
```

Query 9:
```
MATCH (x)-[:hasAdvisor]->(y)-[:teacherOf]->(z)<-[:takesCourse]-(x)
RETURN ID(x), ID(y), ID(z)
```

Query 10:

```
MATCH (x)-[:hasAdvisor]->(y)-[:teacherOf]->(z)<-[:takesCourse]-(x)
RETURN ID(x), ID(y), ID(z)
```

A.2 ADVOGATO CYPHER QUERIES

```
Query 1:
MATCH (x)-[:apprentice]->(y)-[:apprentice]->(z)-[:apprentice]->(x)
RETURN ID(x), ID(y), ID(z)

Query 2:
MATCH (x)-[:journeyer]->(y)-[:journeyer]->(z)-[:journeyer]->(x)
RETURN ID(x), ID(y), ID(z)

Query 3:
MATCH (x)-[:master]->(y)-[:master]->(z)-[:master]->(x)
RETURN ID(x), ID(y), ID(z)

Query 4:
MATCH (x)-[:apprentice]->(y)-[:journeyer]->(z)-[:master]->(x)
RETURN ID(x), ID(y), ID(z)

Query 5:
MATCH (x)-[:apprentice]->(y)-[:apprentice]->(z)-[:master]->(x)
RETURN ID(x), ID(y), ID(z)
```

A.3 LDBC CYPHER QUERIES

```
Query 1:
MATCH (x)-[:KNOWS]->(y)-[:PERSON_IS_LOCATED_IN]->(z)
RETURN ID(x), ID(y), ID(z)

Query 2:
MATCH (x)-[:KNOWS]->(y)<-[:POST_HAS_CREATOR]-(z)
RETURN ID(x), ID(y), ID(z)

Query 4:
MATCH (x)-[:KNOWS]->(y)<-[:POST_HAS_CREATOR]-(z)-[:POST_HAS_TAG]->(w)
RETURN ID(x), ID(y), ID(z), ID(w)

Query 5:
MATCH (x)-[:KNOWS]->(y)<-[:HAS_MEMBER]-(z)
RETURN ID(x), ID(y), ID(z)

Query 7:
MATCH (x)<-[:POST_HAS_CREATOR]-(y)<-[:LIKES_POST]-(z)
RETURN ID(x), ID(y), ID(z)
```

```
Query 8:
MATCH (x)<-[:POST_HAS_CREATOR]-(y)<-[:REPLY_OF_POST]-(z)-[:COMMENT_HAS_CREATOR]->(w)
RETURN ID(x), ID(y), ID(z), ID(w)

Query 10:
MATCH (x)-[:KNOWS]->(y)-[:KNOWS]->(z)-[:PERSON_IS_LOCATED_IN]->(w)
RETURN ID(x), ID(y), ID(z), ID(w)

Query 11:
MATCH (x)-[:KNOWS]->(y)-[:WORKS_AT]->(z)-[:ORGANIZATION_LOCATED_IN]->(w)
RETURN ID(x), ID(y), ID(z), ID(w)
```

APPENDIX B

Provided here is a brief overview of the Path Index implemented in this thesis.

B.1 OVERVIEW

All of the code is contained within a single Java package available on GitHub at: https://github.com/jsumrall/Path-Index

B.1.1 *Inserting Data*

```
DiskCache disk = DiskCache.temporaryDiskCache("pathIndex.db", false);
IndexTree index = new IndexTree(4, disk);
long[] key = new long[]{123,456,678,999};
index.insert(key);
```

B.1.2 *Querying*

```
DiskCache disk = DiskCache.temporaryDiskCache("pathIndex.db", false);
IndexTree index = new IndexTree(4, disk);
long[] key = new long[]{123,456,678,999};
index.insert(key);
long[] search_key = new long[]{123};
long[] search_result;
SearchCursor cursor = tree.find(key);
try(PageProxyCursor cursor = tree.disk.getCursor(result.pageID,
                    PagedFile.PF_SHARED_LOCK)) {
        while(result.hashNext(lock))
                search_result = cursor.next(lock));
}
```

B.1.3 *Removing Data*

```
DiskCache disk = DiskCache.temporaryDiskCache("pathIndex.db", false);
IndexTree index = new IndexTree(4, disk);
long[] key = new long[]{123,456,678,999};
index.insert(key);
index.remove(key);
```