EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Database Group

# Path Indexing in the Cypher Query Pipeline

*Master's Thesis*

Jochem Kuijpers

*Academic Supervisors:*
dr. George Fletcher
dr. Nikolay Yakovets

*Industry Supervisor:*
Tobias Lindaaker
(Neo4j, Sweden)

*Examination Committee:*
dr. George Fletcher
Tobias Lindaaker
dr. Nikolay Yakovets
dr. Nicola Zannone

In collaboration with neo4j

January, 2020
Eindhoven, The Netherlands

Jochem Kuijpers: *Path Indexing in the Cypher Query Pipeline*

# Path Indexing in the Cypher Query Pipeline

Jochem Kuijpers
j.j.l.kuijpers@student.tue.nl
tue@jochemkuijpers.nl

## ABSTRACT

In this study, we investigate how a recently developed index, a so-called path index, can be integrated into the Cypher query pipeline of the industrial Neo4j graph database. We investigate the practical aspect of implementation; we look at how to maintain these indexes and we perform an experimental analysis using our prototype implementation to identify characteristics of use-cases where these indexes are beneficial to query evaluation and index maintenance performance. Due to their exponential worst case size requirement and associated scan times, we conclude that path indexes are most effective when used on selective patterns that allows the query planner to avoid high intermediate state cardinality and thus speed up query performance.

## 1 INTRODUCTION

The demand for efficient data retrieval of connected data is ever-increasing [1]. A graph database facilitates pattern queries on highly connected data by executing the query directly on a graph data-structure. When expensive patterns exceed acceptable query performance, one can generally pre-compute some selection on the data and store it separately from the main data set. This is called an index.

Over the last decades, the increasing connectedness of data and the demand of accurate analysis has called for efficient retrieval of connected data in large datasets. A graph database is a type of database that specializes in storing and retrieving highly connected data. Graph algorithms assist the database in executing data queries.

A common operation in graph databases is pattern query evaluation [1]. This operation searches for sub-graphs in the data with a structure that is constrained by the query. This is typically done by a breath-first search in some way or another. A typical query pipeline includes a query planner that attempts to find a good query plan, as simply running a breath-first search can be exceedingly expensive on large data sets.

A structural index on paths has been introduced [11, 12] in prior work. It has been shown that this index can be effective in speeding up query evaluation [8] and can be maintained when the underlying data graph is updated [2]. In this thesis we document the practical integration of such an index into the Cypher query pipeline of the Neo4j graph database management system. With this implementation we explore use-cases where such path indexes are expected to improve performance, and analyse whether or not this effect is achieved using benchmarks.

*Contributions.* In this study, we investigate how a recently developed index, a so-called Path Index, can be integrated into the Cypher query pipeline of the industrial Neo4j graph database. We investigate the practical aspect of implementation; we look at how to maintain these indexes and we perform an experimental analysis using our prototype to identify characteristics of use-cases where these indexes are beneficial to query evaluation and index maintenance performance.

## 2 PREREQUISITES

This section contains relevant concepts and required knowledge used in this report. Section 2.1 discusses the Neo4j graph database management system, including data layout, the Cypher query language and a high-level overview of the query pipeline. Section 2.3 gives an overview of the Path Index.

## 2.1 Neo4j Graph DBMS

As a part of this study, we base our prototype on the Neo4j 3.5 code base [6]. While we build our prototype with access to the enterprise code base, no enterprise features were used in this study. This section provides an overview of the important aspects of this code base.

### 2.1.1 Property Graph

Neo4j uses the *Property Graph* meta-model for its data storage. A property graph is a graph with the following properties; Every node can have an arbitrary number of labels. Every relationship is directed between two nodes and has exactly one type. The graph may contain multiple relationships of the same type between the same nodes, i.e. it is a multi-graph. Every node and relationship can have an arbitrary number of associated key-value pairs.

### 2.1.2 Data Layout

The efficiency of data retrieval depends on the structure in which the data is stored. Therefore, we will briefly discuss the memory representation of the property graph. A visualization of the memory layout is given in Figure 1.

The data records in this figure each have their own store. Every store is a sequential block of memory that is mapped to a file on

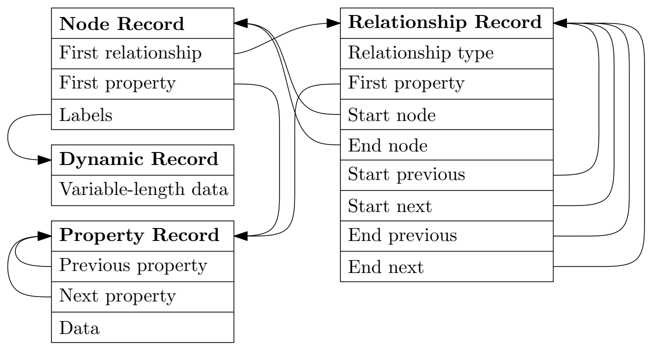

**Figure 1: The memory layout of the data records that make up a property graph.**

disk. The database abstracts transactional modifications to these stores for us, so we will not discuss that aspect.

Node records contain a pointer to a linked list of relationships. Each relationship therefore is an element in both the linked list of the start and end nodes. Further, node records contain a pointer to a linked list of the node properties.

Relationship records function as the doubly-linked list elements for the lists of relationships for the start and end nodes. The start and end nodes are the two nodes connected by the directed relationship. The type is stored in the record since there can be only one. Relationship properties are also stored in a linked list.

Additionally, a node whose number of relationships exceeds a threshold, will not link to a *relationship record*, but instead its relationships are divided into groups, one for each relationship type. The relationship pointer of a node then points to the start of a similarly linked list of *relationship group records*, each of which contains a pointer to the first *relationship record* of the type of that group. This allows faster iteration on relationships with a given type.

### 2.1.3 The Cypher Query Language

Cypher is a declarative graph query language that is loosely based on SQL, as described by Francis et al. [5]. It contains familiar SQL keywords such as `WHERE` that function more or less the same by allowing users to apply predicates to filter the results of the query. However, in Cypher, the primary way to retrieve data is using the `MATCH`-clause. Such a clause contains one or more pattern expressions. A pattern expression is an alternating sequence of nodes and relationships, starting and ending with a node. Nodes are expressed using parentheses while relationships are expressed as *ASCII-art* arrows. Query variables are declared by their inclusion in one or more pattern expressions and can be used in other clauses.

As an example: `(n)` represents a single node `n` and `(o)-[r]->(p)` describes a directed relationship `r` from node `o` to node `p`. A pattern expression can also express label or type constraints, such as the following expression.

`(alice:Person)-[likes:Likes]->(bob:Person)`. This expression describes a node `alice` that has a label `Person` and is connected to another node bob with the same label, by a relationship `likes` of type `Likes`.

Finally, a Cypher query always ends with a `RETURN`-clause, which lists the values that are projected out from the query to produce the query result. Multiple queries can be chained by using the `WITH`-clause in place of a `RETURN` in order to project out values and use those as arguments for the following query. An example of an Cypher query is given in Figure 2.

### 2.1.4 Cypher Query Pipeline

Every query that is executed in the database is enclosed in a transaction. Marking the transaction successful and closing it applies all changes made to the data, thus the entire query pipeline works within a transactional context. If a transaction is not marked successful, the changed made will not be applied to the underlying data store. Everything in the query pipeline is executed on the same thread that the transaction was created on. To simplify the calling API, a transaction is therefore thread-bound.

```
// one connected pattern
MATCH (a:A)-[r:R]->(b)
MATCH (b)-->(a)
MATCH (b)-->(c)
WHERE a.prop = b.prop

// projection boundary
WITH a, r

// another small pattern
MATCH (s)-->(t)
WHERE s.prop = r.prop

// final projection
RETURN a, r, s, t;
```

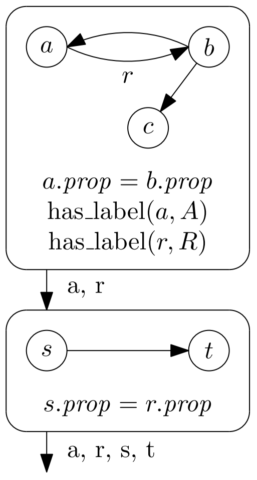

**Figure 2: A cypher query (left) and a visualization of the two query graphs that represent the query parts between the projection boundaries (right).**

Figure 3 shows a high-level overview of the query pipeline architecture. Upon submission, the query is parsed and handed off to the planner. The query plan is a tree of special-purpose operators that either represent a data source (a so-called leaf operator), or a transformation of one or more results from data sources, the result of which is another data source in the tree. The root of the tree is the operator that projects out the query result.

The planner uses a cost estimator which in turn uses statistics from the graph store to estimate the cardinality of sub-trees of a query plan. The cardinality, along with a cost-per-row for each operator, creates a cost per plan. This cost is used to select the cheapest plan which is expected to have the best performance.

Once a query plan is created for the query, the symbolic operators are replaced by runtime-specific operators that either read directly from a data store or implement the transformations represented by the symbolic operators. The data stores that the leaf operators pull

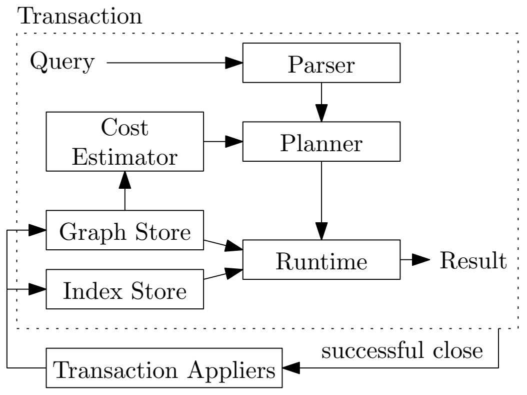

**Figure 3: An overview of the query pipeline architecture.**

from can either be the graph data directly or an index structure. The results are pulled from the executable plan using an iterator interface. This interface is then used by the user to produce as many or as few results as they need.

When a transaction is closed and it marked as successful, any modified transaction state is transformed into a series of commands to be applied to the graph store. These commands are sent to transaction appliers that maintain the data stores and statistics. For indexes specifically, these commands are transformed into a series of index entries that are to be added or removed from the indexes.

## 2.2 Overview of the Cypher Planner

The Cypher query string is parsed into an *abstract syntax tree* (AST). This AST is semantically annotated (e.g. with data types for variables). The AST query is then split into parts by `WITH` or `RETURN` clauses or boundaries. `WITH` boundaries act as projections after which another sub-query follows that uses the projected variables as arguments. A `RETURN` boundary simply projects out the query results at the end of the query.

The query parts in-between these boundaries consist of `MATCH` and `WHERE` clauses. These clauses together describe a query graph. For each of the query parts, a query graph object is therefore constructed. Further, this query graph is split into connected components. Each connected component describes a query pattern that is connected by zero or more relationships such that all nodes in the pattern are reachable from the other nodes in the pattern when following relationships in either direction.

An example of a complex query and its resulting query graphs and projections can be found in Figure 2. This query cannot be rewritten to reduce the number of query graphs since the patterns are not connected. Thus the patterns will each be planned separately. The query graphs will be planned in top-to-bottom order and then combined into a query plan for the entire query.

### 2.2.1 Leaf Operators

The first step in creating a plan for a query graph is to look at all available leaf operators that can solve some portion of it. A leaf operator is a query operator that does not depend on the output of other query operators and typically get their results from disk.

Leaf operators either produce a node or a relationship connecting two nodes. Since the remaining part of the planner reasons about a query graph in terms of its relationships, some leaf operators may need to be extended to solve for an adjacent relationship. For instance, an *AllNodesScan* operator may be extended by an *ExpandAll* operator that traverses all relationships of a given type from the node produced by the node scan operator. These optionally extended leaf operators are called leaf plans.

The leaf plans are grouped into bins based on the relationship of the query graph that they solve, as some plans may solve the same relationship. These are then cost-compared and, for each group, the best plan is kept.

### 2.2.2 Iterative Dynamic Programming Solver

After the leaf plans are in place, a dynamic programming solver based on Iterative Dynamic Programming [7] combines the leaf plans into bigger query plans. The resulting plans that solve the same part of the query graph are cost-compared and only the best is kept for the next dynamic programming iteration.

The resulting plans are created by pattern matching functions called *solver steps*. These provide ways to combine existing plans to create a plan that solves exactly one relationship more than the plans generated in the previous generation. For instance, two smaller plans that both solve the same node $n$, can be joined together by a *NodeHashJoin* on this node $n$. An alternative solution can be to take the best plan that is missing one relationship and extending the plan with *ExpandAll* or *ExpandInto*. These are the two solver steps that exist in the code base.

### 2.2.3 Cypher Query Operators

The following list provides an overview of some of the most used query operators available in Neo4j 3.5. The list is by no means exhaustive and for many of these operators, slight variations exist. However, it should give a good overview of the capabilities of the planner.

(1) AllNodeScan – scans all nodes in the database.
(2) NodeByLabelScan – utilizes the built-in index on node labels.
(3) NodeByIdSeek – find nodes by their internal ID.
(4) NodeByIndexScan – scans all nodes present in a user-defined index.
(5) NodeByIndexSeek – find nodes using a user-defined index.
(6) Expand(All) – Traverse a relationship to find new nodes connected to previously found nodes.
(7) Expand(Into) – Find a new relationship between previously found nodes.
(8) NodeHashJoin – Combines two independent results based on an overlapping set of nodes.
(9) CartesianProduct – Combines two independent results without any overlapping components.

## 2.3 Prior Work on the Path Index

Querying graph databases using indexes is a complex field of study. In 2018, *Querying Graphs* by Bonifati et al. [1] provides a detailed survey of existing techniques in Chapter 6.

The *k-path index* was introduced by Sumrall, Peters and Fletcher et al. [4, 9, 11, 12] in 2015 and showed that significant query performance improvements are possible by using this new type of index that indexes all, or a subset of, paths of length up to $k$, where $k$ is the count of relationships in the indexed paths. Peters built the $k$-path index by concatenating edges in the graph and storing the resulting paths in a relational database. The resulting table was indexed and used to speed up regular path queries.

Sumrall describes the $k$-path index as directly implemented using a $B^+$-tree and demonstrates the potential for answering simple path queries. In his work, Sumrall also describes how one could index paths with specific patterns rather than all paths of a given length $k$. A path pattern describes the node labels and relationship types that a path must contain for it to be added to the index.

In 2016, Persson [8] utilized the path index concept to speed up dense network data retrieval by indexing all paths of length 1. This index was called a *Shortcut Index*. Persson intentionally limited his work to indexes with a single relationship to avoid expensive

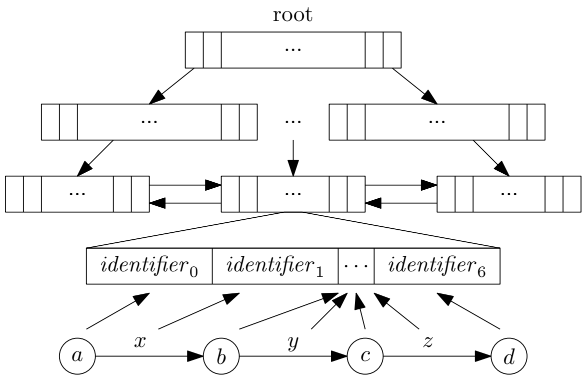

**Figure 4: The B$^+$-tree layout of the path index. Every entry is a list of numeric identifiers (of 8 bytes each). This example stores a 7-component pattern.**

maintenance computations on graph updates, which could not be afforded in the described use-case.

In 2019, De Jong [2] demonstrated how indexes on path patterns of greater length can be more efficiently maintained by maintaining sub-patterns as separate indexes. This allows for a significant speed-up of index maintenance, at the cost of increased storage requirements for the path indexes on sub-patterns.

Our work continues on the prior work mentioned here. Our path index implementation was inherited from the work done by De Jong, although we have made modifications. This index only indexes paths with specific patterns as opposed to all paths with a given length. From now on, we shall refer to this type of index as a *path index*. The next section will briefly describe the implementation we used.

### 2.3.1 Path Index Data Structure

The primary path index data structure is a B$^+$-tree. The specific B$^+$-tree implementation that we use is provided by Neo4j 3.5. Every indexed pattern is stored in its own separate tree, such that we need not store any pattern information in the B$^+$-tree itself. The only information that we store in the index are the identifiers of the nodes and relationships that we wish to index.

Figure 4 shows the different components of the B$^+$-tree. The keys stored in the internal nodes are also lists of identifiers. The B$^+$-tree entries are sorted lexicographically. The figure shows a pattern containing 7 components and a key storing the 7 corresponding identifiers. Each identifier is 8 bytes long. Thus for a path pattern of a given length $k$, the entries will be $8(2k + 1)$ bytes in size.

The B$^+$-tree allows fast prefix searches on the identifiers of paths, as well as sequential scans in the sorted entries of the B$^+$-tree, although it does not provide us with a way to compress the redundant information easily. The size requirements are $O(kn)$ where $n$ is the number of paths stored in the index and $k$ the length of the pattern. In the worst case, $k$ can be exponential in $n$, thus the size requirement can be $O(e^n)$.

## 3 PROBLEM STATEMENT

In this case-study, we integrate the path index into the Neo4j Cypher query pipeline and we perform an experimental evaluation. We wish to identify actionable properties of use-cases in which the benefits of path indexes outweigh their storage size and cost of maintenance. To do this, we aim to answer the following questions.

(1) What are the benefits of indexing longer path patterns?
(2) Can we maintain an arbitrary set of path indexes efficiently?
(3) In what ways can we use path indexes during query evaluation?
(4) How can we balance the size of the index with its performance gain?

In summary, we find that there are indeed use-cases where path indexes are able to significantly speed up our query workloads. These use-cases tend to be query workloads with selective path patterns on correlated data.

One aspect that is responsible for this result, is that the cost estimator of the database is based on an assumption of independent selectivity. Correlated data violates this assumption, thus the resulting plans tend to be sub-optimal. Path Indexes are easier to plan when large parts of the query pattern matches the indexed pattern, thus generally result into faster plans.

Though, another aspect is that path indexes enable the query planner to avoid high-cardinality intermediate states if there is an index on a selective or correlated structure. We found that even with a manually optimized plan, path indexes can still yield significant performance increases.

## 4 PATH INDEX IMPLEMENTATION

In this section we will discuss the implementation related to integrating the path index into the database code-base. We will start with an overview of all components that required modifications in Section 4.1. Section 4.1.1 describes query-based path index maintenance. Then Section 4.1.2 provides information on path index initialization.

The changes made to the planner and runtime are discussed in more detail in Section 5.

## 4.1 Modified Pipeline Components

The implementation of our path index into the query pipeline required modifications in several components. Figure 5 shows an overview of the components, where a shaded background means the component required some modification. The dashed block "Subquery" represents a new component.

The cost estimator is extended to estimate a cost for path index operators. We re-used the existing cardinality estimator due to scoping constraints which assumes that all filtering and combining operations behave according to the global statistics of the data.

The path index implementation was partly inherited from prior work done by De Jong, though changes were made to facilitate partially materialized indexes. Some more changes needed to be made to support the baseline extension discussed in Section 5.

### 4.1.1 Query-based Path Index Maintenance

De Jong [2] observed that, as a consequence of the policy in the Neo4j database to never allow the deletion of a connected node, we only ever need to look for relationship updates in the graph in order to update the path index.

De Jong describes two methods for translating the graph updates into path index updates;

(1) Traversal-based translation; starting from the updated relationship, traverse the indexed pattern on the data graph and make note of all the paths encountered. If the relationship was added, then add all these paths to the index. Otherwise, remove all these paths from the index.
(2) Self-maintaining translation; maintain a path index for all the sub-patterns that can be created from the index pattern. Some of the sub-patterns need to be reversed in order to do prefix-scans on these indexes. Then, when handling an updated relationship, simply do a prefix search on the largest index that contains the changed relationship in both directions, and combine the two resulting sets created by these actions. Then add or remove these paths from the index.

The first method is a naive graph traversal. The second method requires all sub-patterns to be indexed. We introduce the following maintenance method, which uses the query pipeline itself to find the most effective way to search for updated paths:

(3) Query-based translation; query the index pattern with an additional predicate that the modified relationship must be part of the resulting paths. This query then returns all paths through the updated relationship. We can then add or remove these paths from the index.

The last approach is far more flexible in terms of which pattern indexes are allowed to exist, compared to self-maintaining translation. However, it requires executing new queries while processing transactions. This broke some assumptions about the way transactions are handled in Neo4j, namely one transaction per execution thread. As a result, our prototype does not support concurrent updates. Another issue was that we needed to by-pass the query cache, otherwise we had no control over which indexes would be used in the maintenance queries.

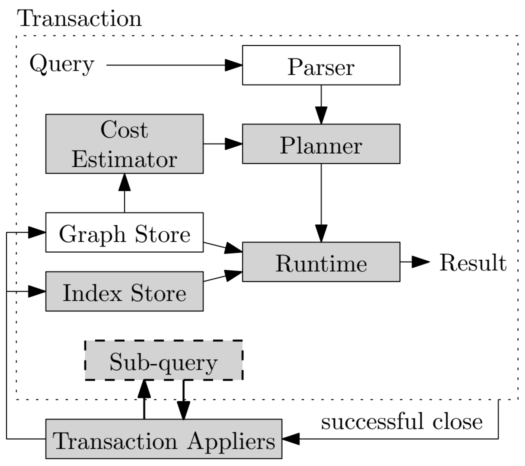

**Figure 5: An overview of the query pipeline architecture where shaded boxes indicate modifications to existing components and the dashed box indicates a new component.**

When a relationship is modified, added or removed in the graph, this could mean entries need to be added or removed from the path indexes. To find out which paths have been changed, a query is executed that contains the pattern of the index and an additional constraint that the relationship in the pattern must match the updated relationship from the transaction. This is described in Algorithm 1.

**Algorithm 1** Path Index Maintenance

```
    Input Modified relationship $r$ in graph $G$.
 1: $b$ := the label of the start node of $r$
 2: $e$ := the label of the end node of $r$
 3: $t$ := the type of $r$
 4: $I$ := a list of path indexes with patterns that contain
    ...(:b)-[:t]->(:e)...
 5: Sort $I$ by pattern length, ascending.
 6: $T_{old}$ := the committed transaction state.
 7: Reset the transaction state.
 8: if $r$ is a removed relationship then
 9:     for all $index$ in $I$ do
10:         $P$ := the pattern of $index$ containing relationship $r$
11:         $R$ := QUERY($P$, $G$)
12:         Remove all entries $R$ from $index$.
13: Process all other transaction appliers for $r$.
14: if $r$ is an added relationship then
15:     for all $index$ in $I$ do
16:         $P$ := the pattern of $index$ containing relationship $r$
17:         $R$ := QUERY($P$ but avoid using $index$, $G$)
18:         Add all entries $R$ to $index$.
19: Set the transaction state to $T_{old}$.
```

There are some important things to consider. First of all, Neo4j 3.5[1] binds a transaction along with its transaction state to the thread that has opened it. That means that, while we are applying the transaction, it is not trivial to start a new transaction for the maintenance query since we are using the same thread. The default Neo4j behaviour is to return the already active transaction when creating a new transaction, which has a modified transaction state that modifies the query results we receive during maintenance. As a work-around, we store the old transaction state, reset the transaction state for the maintenance queries and then after processing the update, restore the transaction state such that other Neo4j operations are not affected.

Further, index processing ordering does matter. Since we are halfway through applying a transaction, some of the data might not be up-to-date while other data will have been updated to reflect the applied transaction already. By sorting the indexes small-to-large, we are sure that any maintenance query plan that uses such an index will include the data to add or remove. Further, by querying for removal before any other data is updated and querying for addition after all other data is updated, we ensure that regardless of the selected query plan, the right query results will be returned. Note that modified relationships can be modelled as a removal followed by an addition, thus this does not need to be a special case.

[1]Transactions no longer bind to threads in Neo4j 4.0

In a similar style, maintenance can be applied to node label updates. Node additions and removals do not need to be processed, as those are only allowed for nodes with no attached relationships, making a possible match with a path index impossible.

#### 4.1.2 Path Index Initialisation

A small but important aspect is being able to create indexes on existing data: index initialization. This is done by querying the pattern on the existing data graph and adding the result set to the new index in a single transaction. Other indexes that have already been initialized may be used at this point. Index initialization thus follows the simple procedure described in Algorithm 2.

Sumrall [11] proposed constructing a B$^+$-tree directly from a sorted list of query results in order to speed up the B$^+$-tree construction, though this was not practical to achieve in our implementation since the B$^+$-tree memory layout is abstracted in the code base. As index initialization was not the primary focus of this study, we used our more naive approach, which increases the one-time construction cost of any path index.

**Algorithm 2** Index initialization

```
    **Input** index pattern $P$, data graph $G$
    **Output** initialized index $I$
1:  $I$ := a new path index
2:  $Result\_Iterator$ := Query($P$, $G$)
3:  **while** $Result$ := $Result\_Iterator.next$ **do**
4:      Add $Result$ to $I$
```

## 5 PATH INDEX QUERY PLANNING

In this section, we will only discuss *read-only* queries, as *writing* or *updating* queries do not affect how path indexes are planned. Path index maintenance for graph updates has already been discussed in the previous section.

### 5.1 Path Index Query Operators

Recall that there are two components in the query planner that create symbolic query plans. Firstly there is a leaf planner, which produces leaf plans. These are query plans that simply read from a data source and after some optional filtering, emit the result. Secondly we have solver steps. These are invoked by the dynamic programming solver described in Section 2.2.2.

Using this loose definition of a leaf plan, reading from a path index could be seen as a leaf plan. However, when initializing the dynamic programming solver, it is assumed that all leaf plans solve exactly one relationship. This assumption is a key part of how the solver functions; all plans in the current iteration are expected to solve an identical number of relationships.

Therefore, we were forced to write two separate planners:

(1) A leaf planner for path indexes, that plans all path index operators that provide exactly one relationship.
(2) A solver step planner, that provides plans using path index operators with more than a single relationship in the final result.

Both of these planners are able to plan the same operators. The only difference is that the PathIndexPrefixSeek operator is not planned by the leaf planner since this is never a leaf plan.

#### 5.1.1 PathIndexScan

This operator is the most simple way to use a path index in a query plan. It is offered as a possible plan whenever the indexed pattern matches the requested query pattern exactly. Since the planner builds query plans iteratively, any query pattern that contains the indexed pattern will at some point be offered to use the PathIndexScan operator. If the operator is then chosen as the best plan for this sub-pattern, it may be used in larger plans as a leaf node in the query plan tree.

This query operator simply scans the entire path index from start to finish. Because of how the path index is structured, this also provides the property that the first stored value in the path index is in ascending order.

The cost for the PathIndexScan is derived as follows. First the $c$ is estimated using the cardinality estimator. Then the number of stored identifiers in the pattern, $n$, is computed. The final cost of this operator is then computed to be: $cost = c(1 + 0.1n)$. This formula is a heuristic based on a small number of benchmarks. Special debug parameters were added to reduce the cost function and to provide more control over the selected plan for these new operators.

#### 5.1.2 PathIndexFilteredScan

Whenever PathIndexScan is applicable and there are predicates left to filter on, a plan containing PathIndexFilteredScan is created as well. The data structure of the index, the B$^+$-tree, provides the capability of finding the first occurrence of a prefix in logarithmic time. Therefore it is easy to quickly skip over entire ranges of the index that cannot satisfy a particular constraint.

For instance, in a length-$k$ path $p$, the condition that node $n$ and $m$ node cannot be equal, allows us to skip over all paths where $p_n = p_m$ by searching for the prefix $\langle p_0, \ldots, p_n, \ldots, p_{m-1}, (p_m+1)\rangle$ and continuing the index scan from there as soon as the condition is violated. Of course, the smaller the prefix, the larger the part of the index sub-tree that can be skipped over.

Using $c$ for the cardinality estimation of the indexed pattern including the added predicates, the cost heuristic of this operator is computed as $cost = c(1.05 + 0.1n)$. For somewhat selective predicates, this should always be a cheaper option than the PathIndexScan operator as it is expected to be faster in these cases. This should be reflected in the cardinality estimation.

#### 5.1.3 PathIndexPrefixSeek

Whenever the indexed pattern is found as a subset of the requested query pattern, and some relationships remain unsolved, an already computed plan can be extended with this operator if its pattern overlaps with the beginning of the indexed pattern. The operator will first take in all results from the child plan, compute the relevant prefix for each result and group all results by this prefix.

Then for each prefix, it will seek the index for this prefix and scan all paths. Then combine all the returned paths with all the results in the prefix group. This will continue until all prefix groups have been processed.

The cost for this operator is slightly more complex as it will depend on the cost of the child plan. The child plan cost will be denoted with $cost_{child}$ and the child cardinality with $c_{child}$. The cardinality for the new plan, including the PathIndexPrefixSeek operator, is computed as follows. $fraction$ equals the number of symbols in the prefix divided by the number of symbols in the child plan. $m = c_{child} \cdot fraction$. $cost = 2 \cdot cost_{child} + 10m + \frac{c}{m}$. The value of $m$ is an approximation of the number of unique prefixes. This value is not necessarily accurate, although this cost function has been shown to work well in some of our experiments.

#### 5.1.4 PathIndexClosure

This operator would produce the closure of a pattern. However, this proved impossible to query in the current version of the Cypher query language. It is currently not possible to create a Kleene-star closure over an arbitrary pattern expression. To limit the scope of this project, we decided to let this pass and focus on the things that can be queried using Cypher only.

## 6 EXPERIMENTAL SETUP

The goal of our experiments is to answer the research questions in Section 3. During this study, we noticed how path indexes were typically useful in correlated data sets with low cardinality patterns. To explore this hypothesis, we perform benchmarks on synthetic data sets where we can control the correlations. Further, we show how index maintenance is affected by the presence of our chosen indexes and we show the index initialisation time and storage requirements.

Later we perform benchmarks on real-world data sets and attempt to verify our hypothesis that finding selective patterns on correlated data is a good heuristic for path index use-cases.

### 6.1 Baseline Planner Extension

The current planning model used by Neo4j is node-centric. There exist a number of ways to selectively scan or seek nodes by their node labels and indexes exist on node properties, but there are almost no ways to selectively scan or seek based on relationship types.

Since our path index implementation will be able to selectively scan results based on relationship types as part of the indexed pattern, we believe a fair comparison should extend the baseline planner to add this capability as well.

Thus we utilize our path index implementation and provide an index that contains exactly one relationship type as its pattern, without node labels. These can then be queried using the RelationshipByTypeScan operator that we introduce. As the name suggests, this operator will scan all relationships given a certain type. It is introduced as a leaf operator with the same per-row cost as NodeByLabelScan.

### 6.2 Hardware and Software

Our experiments were performed on a benchmark server containing four Intel Xeon E5-4610 v2 CPUs running at 2.30GHz, 500GB of DDR3 RAM at 1600MHz and for our experiments, we used its 260GB NVMe SSD as data storage. The benchmark machine runs Ubuntu 16.04.3 LTS and the Oracle Java (TM) SE Runtime Environment (version 1.8.0-151). Our prototype was forked from the Neo4j 3.5 code base.

### 6.3 Testing Methodology

Our experiments were run with a pre-allocated heap of 100GB. Each experiment ran until running time converges, which indicates that hot code paths were optimized. Then we ran the experiment five times, triggering a garbage collection cycle between each run. We then discarded the highest and lowest running time and averaged the middle three. For data set size measurement, we summed the total file size on disk of all database files, except for transaction logs. Indexes were measured and reported separately from the rest of the database where applicable.

For testing the behaviour of cold experiments, we kept the program running but closed and re-opened the database within the program. This causes the database to flush its memory cache without losing the optimized code paths. This is done for every run of the cold experiments.

### 6.4 Data sets

We use four data sets in our experiments. Two data sets are synthetically generated while the other two are real-world data sets.

The first synthetic data set is used to demonstrate the behavior of path indexes on structurally correlated data. It was generated by connecting 25 000 labeled paths. The resulting data set has 125 000 nodes and 12 600 000 relationships, while the number of occurrences of the labeled paths has not increased.

The second synthetic data set is generated to demonstrate the behavior of path indexes on data with no structural correlations. This data set is a scale-free graph that was created using a preferential attachment model. Each new node was attached with 20 new relationships to the graph, where the target node was probabilistically determined based on the degree of the target node. The final graph has 250 000 nodes and 5 000 000 relationships. Node labels and relationship types were randomly and uniformly assigned.

The third data set is our first real-world data set. This is the YAGO data set [10]. It contains 77 139 979 nodes and 99 218 253 relationships. The last and second real-world dataset is the GeoSpecies data set [3], which contains 225 093 nodes and 1 542 463 relationships. Both real-world data sets were originally distributed as RDF graphs, and have been imported into the property graph format.

## 7 EVALUATION

This section is divided into several parts. Each part discusses one experiment, which is explained and motivated. Then the results are presented alongside a discussion of the results.

### 7.1 Benchmark on Correlated Data

Path indexes allow us to avoid large intermediate results while querying larger patterns. We wish to see how our path index performs compared to a query plan provided by the baseline planner. In this experiment, we present an optimistic scenario where the data set and query workload are chosen such that the path index is very beneficial to our performance. More specifically, relationships in our graph are strategically added to our graph to create a very selective pattern.

| Result | Baseline | Full Index | Speed-up |
|---|---|---|---|
| First result, cached | 36 568.60 ms | 2.97 ms | ≈ 12 310× |
| Last result, cached | 51 485.67 ms | 103.63 ms | ≈ 497× |
| First result, cold | 36 421.72 ms | 102.36 ms | ≈ 356× |
| Last result, cold | 49 326.42 ms | 202.71 ms | ≈ 243× |

**Table 1: Query evaluation comparison of baseline versus path index plans. The values reported are the time it took for the query execution to produce the first or last result. Both memory-cached and *cold* scenarios were tested.**

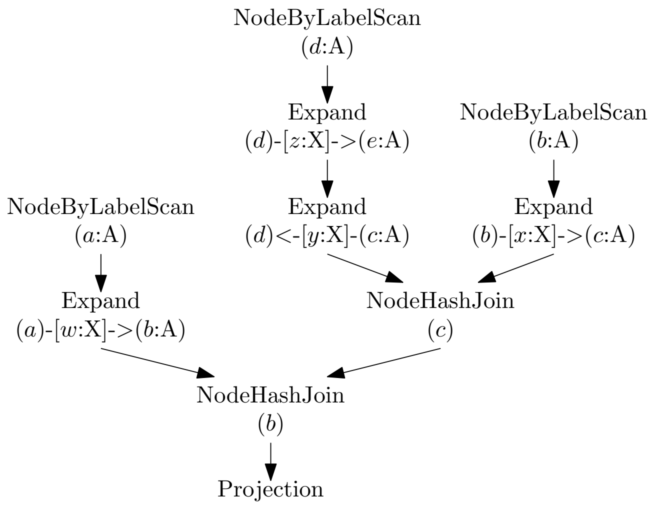

**Figure 6: A simplified illustration of the baseline query plan. Some additional filter steps that filter on node labels or remove duplicate relationships[2]are not depicted.**

### 7.1.1 Baseline Versus Full Index

These experiments execute the following Cypher query. This path pattern appears only 25 000 times in our data set due to structural correlations in the graph.

```
MATCH (a:A)-[w:X]->(b:A)-[x:X]->(c:A)-[y:Y]->
      (d:B)-[z:X]->(e:A)
RETURN *;
```

We create a path index on the same pattern: `(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)`. This pattern contains a string of 4 typed relationships connecting 5 nodes with a specified label, for a total of 9 symbols.

We run four benchmarks. A baseline query plan and a query plan that uses this path index are used. We further looked at the scenario where the index and graph data is present in the system memory (*cached*) and the scenario that this data only resides on disk (*cold*) and first has to be brought into memory.

Table 1 shows the timings for the four scenarios. The presented times represent the time between submitting the query and the first or last result to be received from the result iterator, respectively.

The cardinality of the query result is 25 000. However, the baseline query plan, presented in Figure 6, has a maximum intermediate cardinality of 6 299 500 at result of the Expand (*a*)-[*w*:X]->(*b*:A) operator. While this is not a perfect metric for query plan efficiency, the maximum intermediate cardinality appears to correlate quite well with query performance as it gives a lower bound on the amount of computation required to reach the final result set.

The path index also has a cardinality of 25 000. From this, it is clear how the path index plan is able to achieve a much faster time as it only reads 25 000 entries from the index which are directly added to the result set. This index requires 3.91 MiB of disk space, as can be seen in Table 2, where *Full* represents the index used here.

### 7.1.2 Full Index Versus Indexed Sub-patterns

Database users typically cannot create an index for every query that they want to execute. On regularly updated data, a lot of time would be spent on maintaining these indexes. Further, every index stores redundant data, which requires additional disk space. Therefore, users might want to consider patterns that are still beneficial to their query performance while these smaller indexed patterns can be applied to a larger number of queries.

Further, since path indexes store occurrences of that pattern in the graph, the cardinality of the index should not be too high. A high cardinality has two disadvantageous effects on performance. Firstly, scanning from a larger index means a more data has to be moved, which takes more time, and secondly, the point of a path index is to avoid large computations required to reach the final result. Indexing exactly the patterns in a query that have high intermediate cardinality typically means that the most intense computations are not skipped, which thus defeats the purpose of using a path index.

To demonstrate these properties, we will use the same query and dataset from the previous experiment. However, we create indexes on smaller parts from the query pattern and measure the performance of their query plans. Note that the cost-based planner in our implementation might avoid using the index if another plan was assigned a lower cost. We therefore forced the planner to use the best available plan that uses the specified index.

For this query, we identified 8 indexable patterns that could assist in the evaluation of this query. Table 2 lists the full indexed pattern, as well as these sub-patterns $Sub_1 \ldots Sub_8$. The index cardinality describes the number of entries in the index. Size on disk represents the amount of storage that the Neo4j $B^+$-tree implementation uses. The total data size represents only the actual data stored in the $B^+$-tree. The initialization time is the amount of time it took to construct this index using the baseline planner. The *Graph* row lists the size of the graph data on disk for comparison.

We execute the same query as listed in the previous section using the indexed sub-patterns only. We force the planner to pick a plan that contains an operator that uses this index. In reality, if the plan with the sub-pattern index performs significantly worse, the planner would have likely picked a different plan based on cost estimation. The results of this experiment are listed in Table 3 and visualized in Figure 7.

The left column lists the name of the index used. The first group of columns lists the result of a memory-cached index while the second group of columns lists the result where the index was evicted

[2]The default query semantics in Cypher do not allow a relationship to be matched twice in the same `MATCH` pattern, unless specified otherwise.

| Name | Indexed pattern | Cardinality | Size on disk | Total data size | Initialization |
|---|---|---|---|---|---|
| *Graph* | - | – | 413.97 MiB | – | – |
| *Full* | `(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)` | 25 000 | 3.92 MiB | 1.72 MiB | 1 120 ms |
| $Sub_1$ | `(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)` | 25 000 | 3.17 MiB | 1.33 MiB | 3 862 ms |
| $Sub_2$ | `(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)` | 25 000 | 3.17 MiB | 1.33 MiB | 918 ms |
| $Sub_3$ | `(:A)-[:X]->(:A)-[:X]->(:A)` | 12 524 000 | 970.56 MiB | 477.75 MiB | 14 248 ms |
| $Sub_4$ | `(:A)-[:X]->(:A)-[:Y]->(:B)` | 25 000 | 2.39 MiB | 0.95 MiB | 923 ms |
| $Sub_5$ | `(:A)-[:Y]->(:B)-[:X]->(:A)` | 6 274 500 | 471.59 MiB | 239.35 MiB | 7 083 ms |
| $Sub_6$ | `(:A)-[:X]->(:A)` | 6 299 500 | 364.95 MiB | 144.18 MiB | 4 004 ms |
| $Sub_7$ | `(:A)-[:Y]->(:B)` | 6 274 500 | 250.27 MiB | 143.61 MiB | 6 102 ms |
| $Sub_8$ | `(:B)-[:X]->(:A)` | 25 000 | 1.55 MiB | 0.57 MiB | 23 ms |

**Table 2: The available indexes in the query experiment on correlated data, with their respective cardinality, storage size, data size and initialization time. Data size reflects the actual size of the data in the index. *Graph* shows the graph size.**

| Name | Memory-cached | | | Cold | | | Max. intermediate |
|---|---|---|---|---|---|---|---|
| | First result | Last result | Speed-up | First result | Last result | Speed-up | state cardinality |
| *Baseline* | 36 568.60 ms | 51 485.67 ms | - | 36 421.72 ms | 49 326.42 ms | - | 6 299 500 |
| *Full* | 2.97 ms | 103.63 ms | ≈ 497× | 102.36 ms | 202.71 ms | ≈ 243× | 25 000 |
| $Sub_1$ | 6.84 ms | 210.28 ms | ≈ 244× | 136.03 ms | 341.63 ms | ≈ 144× | 25 000 |
| $Sub_2$ | 12 419.65 ms | 12 606.06 ms | ≈ 4.1× | 12 981.33 ms | 13 176.95 ms | ≈ 3.7× | 6 299 500 |
| $Sub_3$ | 14 620.86 ms | 48 780.42 ms | ≈ 1.1× | 15 032.85 ms | 53 018.79 ms | ≈ 0.9× | 12 524 000 |
| $Sub_4$ | 12 517.03 ms | 12 794.13 ms | ≈ 4.0× | 13 022.80 ms | 13 303.54 ms | ≈ 3.7× | 6 299 500 |
| $Sub_5$ | 31 263.34 ms | 42 028.41 ms | ≈ 1.2× | 32 672.25 ms | 43 342.15 ms | ≈ 1.1× | 6 299 500 |
| $Sub_6$ | 12 887.62 ms | 22 154.25 ms | ≈ 2.3× | 13 796.22 ms | 22 542.65 ms | ≈ 2.2× | 6 299 500 |
| $Sub_7$ | 13 307.46 ms | 26 324.40 ms | ≈ 2.0× | 13 997.20 ms | 27 736.36 ms | ≈ 1.8× | 6 299 500 |
| $Sub_8$ | 32 799.38 ms | 43 815.78 ms | ≈ 1.2× | 35 333.48 ms | 47 130.84 ms | ≈ 1.0× | 6 299 500 |

**Table 3: The results of the query experiment on correlated data. The left-most column holds the names of the indexes. The middle sections show the query performance in a memory-cached and cold scenario and the right-most column shows the maximum intermediate state cardinality during query execution.**

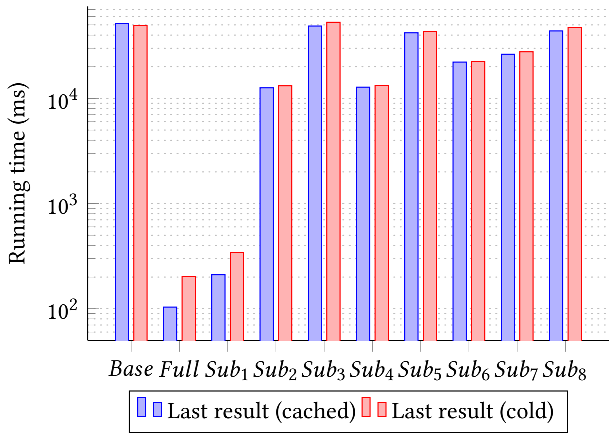

**Figure 7: The benchmark results from Table 3 on correlated data.**

from main memory and had to be streamed from disk. The final column shows our previously discussed metric for performance analysis, namely the maximum cardinality of the intermediate state while executing this query. For the middle two column groups, the time is listed for both the first result returned from the result iterator, as well as the last result. The speed-up factor is computed over the time for the last result only, in comparison with the baseline.

This experiment shows that the plans associated to the *Full* and $Sub_1$ indexes provide almost identical performance increases, though the sub-pattern index requires additional graph traversal to fully answer the question. Both plans are effective in reducing the cardinality of the intermediate state, thus reducing the computations required to answer the query. The other plans achieve this to a much lesser degree, where the $Sub_3$ plan is actually slower than the *Baseline* plan for the cold version of this experiment. Further, we can see how the maximum intermediate state cardinality correlates with query performance.

| Name | Relationship addition |  | Relationship deletion |  | Average speed-up |
|---|---|---|---|---|---|
|  | Full index | Sub index | Full index | Sub index |  |
| None | 0.436 ms | – | 0.836 ms | – | – |
| $Sub_1$ | 0.301 ms | 0.266 ms | 0.824 ms | 0.570 ms | ≈ 0.65× |
| $Sub_2$ | 0.368 ms | 0.333 ms | 0.855 ms | 0.586 ms | ≈ 0.59× |
| $Sub_3$ | 0.288 ms | – | 0.675 ms | – | ≈ 1.32× |
| $Sub_4$ | 0.277 ms | 0.242 ms | 0.648 ms | 0.714 ms | ≈ 0.68× |
| $Sub_5$ | 7 885.829 ms | 0.536 ms | 7 919.113 ms | 1.025 ms | ≈ 0.00× |
| $Sub_6$ | 0.184 ms | – | 0.489 ms | – | ≈ 1.89× |
| $Sub_7$ | 7 110.481 ms | 0.561 ms | 7 142.197 ms | 0.734 ms | ≈ 0.00× |
| $Sub_8$ | 0.219 ms | – | 0.443 ms | – | ≈ 1.92× |

**Table 4: Results of the maintenance experiment on correlated data. The rows show the amount of time required to update the index, given the presence of a sub-pattern index named in the left-most column.**

#### 7.1.3 Maintenance Using Indexed Sub-pattern

Not only can these sub-pattern indexes provide performance benefits during query execution. As De Jong [2] showed, sub-patterns can also be used to speed up the maintenance of the full index. Where the *self-maintaining translation* introduced by De Jong exhaustively provides all sub-patterns such that no data has to be read from the graph, our approach simply defers this decision to the query planner, as maintenance is performed using a specific query on the indexed pattern. This allows us to pick an arbitrary set of path indexes, which may also be used to speed up maintenance when applicable.

In this experiment, we look at the performance benefits provided for maintenance as we provide one of the sub-pattern indexes from Table 2 alongside the *Full* index. The graph is updated to remove one of the Y-labeled relationships in a transaction, after which this same relationship is added again in a new transaction. Table 4 shows the results of this experiment. The first row contains the maintenance performance of just the *Full* index and the subsequent rows contain the performance of using a maintenance plan that includes the sub-pattern index. Further, the sub-pattern index itself may also need to be maintained, thus these measurements are also included. For this experiment, the query planner is forced to use a plan that uses the sub-pattern index. However, in case the *Full* index maintenance is slower using the sub-index, we may assume that the planner would not use the sub-index given the choice. The average speed-up reported is the factor of performance increase of both maintenance operations for the removal and addition of a relationship.

Interestingly, both $Sub_1$ and $Sub_4$, the indexes that provided the most performance increase during query execution, do not speed up the maintenance operations of changes to this specific relationship. $Sub_3$ provides a moderate performance increase for maintenance computations, while it was the worst performing index in the previous query execution experiment.

### 7.2 Benchmarks on Independent Data

We believe our path indexes are most beneficial in situations where structural correlations occur, that is, where a type of relationship is more or less likely to occur on a given node, based on which other types or relationships are connected to its neighborhood.

| Result | Baseline | Full Index | Speed-up |
|---|---|---|---|
| First result, cached | 1 345.49 ms | 3.10 ms | ≈ 434× |
| Last result, cached | 5 814.97 ms | 2 901.87 ms | ≈ 2.0× |
| First result, cold | 1 934.50 ms | 100.69 ms | ≈ 19.2× |
| Last result, cold | 6 107.95 ms | 3 821.86 ms | ≈ 1.6× |

**Table 5: Results of the independent data benchmark with the baseline query plan versus the plan with the fully indexed pattern.**

To confirm our hypothesis, we have generated a large data set that does not contain these structural correlations. Our graph contains 250 000 nodes and 5 000 000 relationships. It is incrementally generated by adding nodes one by one from a starting set of 20 nodes. All nodes are connected by 20 new relationships to the rest graph, though the target nodes of these relationships are chosen with a preferential attachment model where the probability of connecting to a node corresponds to the degree of that node relative to the total degree of all nodes in the graph. We assign labels A, B, . . . , E randomly and uniformly to the nodes, and label the relationships V, W, . . . , Z randomly and uniformly as well.

Now we perform the same experiments as in the previous section on correlated data. However, the query has been adapted to fit this data set, as follows.

```
MATCH (a:A)-[v:V]->(b:B)-[w:W]->(c:C)-[x:X]->
      (d:D)-[y:Y]->(e:E)
RETURN *;
```

#### 7.2.1 Baseline versus Full Index

Similar to the previous experiment, the baseline planner is compared to a plan utilizing the index of the full query pattern under both a memory-cached and cold scenario. The results are shown in Table 5. We can see that the performance increase is vastly smaller than in our previous experiments on correlated data. Though, the index still provides a faster query plan in both scenarios. Table 6

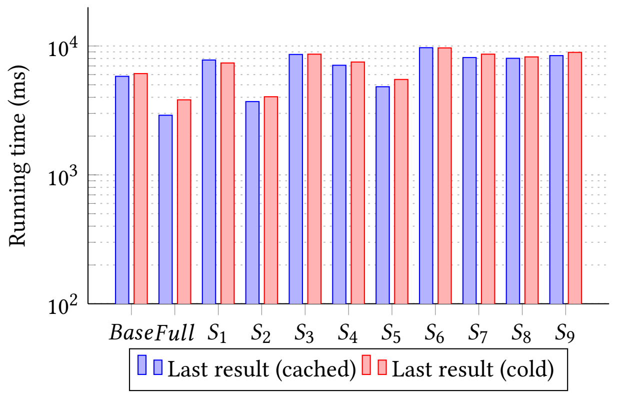

**Figure 8: The benchmark results from Table 7 on independent data.**

shows the size of the indexed pattern as well as its cardinality. This graph is similarly dense to the correlated data set, but the pattern is considerably less selective. This is reflected in the increased cardinality of the *Full* index and its larger relative size to the graph size.

### 7.2.2 Full index versus indexed sub-patterns

We repeat the sub-patterns experiments on this data set. Table 6 shows the cardinality of the sub-pattern indexes, which are similarly to the *Full* index, considerably larger than in the previous experiment. In this experiment, we force the planner to use a plan that uses the sub-pattern index to answer the query. The results are reported in Table 7 and visualized in Figure 8.

We show that the performance benefits of using a path index cannot be replicated on this data set. At best, a speed-up factor of 2.0× is achieved with the *Full* index. The cardinality of the result set is a lot higher and the maximum intermediate cardinality is relatively close to the cardinality of the result set. This shows that in order to answer this query, it is almost impossible to skip over the high cardinality computations using a path index.

### 7.2.3 Maintenance using indexed sub-pattern

We observe that similar speed improvements can be achieved using strategic sub-pattern indexes on the full index in Table 8. Disk space requirements for these indexes is larger than in the correlated data set, so it may not be as beneficial to use sub-pattern indexes in this scenario.

## 7.3 Using Path Indexes with YAGO

We found that correlated patterns in graph data are good use-cases for path indexes, since that provides an opportunity to index a selective pattern that skips over some large intermediate state compared to the baseline plan.

In this experiment, we show that this indeed does work in real-world scenarios. We use the YAGO [10] data set. Further, we find that the path index initialization time shows us that a more optimal baseline plan should exist.

To find correlated patterns, we looked at a query workload of several thousand length-5 patterns; path patterns that contain 5

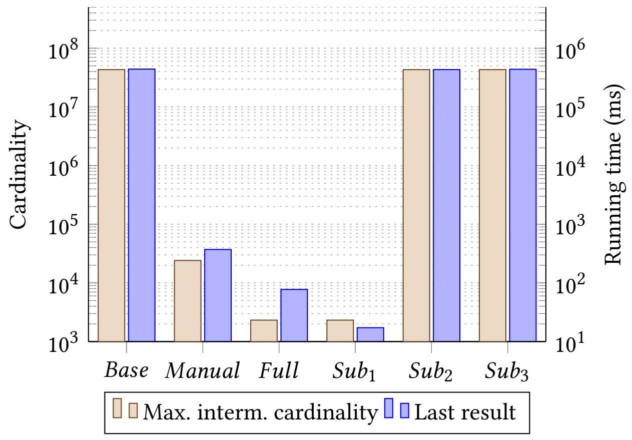

**Figure 9: The benchmark results from Table 10 on the YAGO data set, illustrating the strong correlation between the maximum intermediate cardinality and running time.**

relationship types. Since the Neo4j planner uses an independence model for cardinality estimation, comparing the cardinality estimation of these patterns to cardinality of the result set allows us to compute a misprediction factor. A large factor indicates that this pattern is not independently selective, therefore we conclude a structural correlation.

The query workload set does not include node labels. The data set however does provide node labels and our path index requires exactly one node label for each node in the pattern. In order to find the node labels that we can use in the pattern, we query the pattern and then pick the most selective label that is present on all nodes in the pattern for all results of the query. Those node labels are then used in the index pattern.

Using this process on all query patterns in the workload set, we find the pattern with the highest misprediction factor. Our assumption is that this is a good candidate for our full-pattern index as the misprediction factor indicates correlated data. The resulting query is as follows. Note that label and type names have been abbreviated slightly for readability. `Resource` is a label assigned to all nodes in this data set. We have to use it for node `f` because it does not have any other labels that were present on all results.

```
MATCH (a:wordnet_person)-[w:isAffiliatedTo]->
      (b:wordnet_person)-[v:wasBornIn]->
      (c:port_settlement_in_USA)-[x:owns]->
      (d:wordnet_artifact)-[y:isConnectedTo]->
      (e:wordnet_artifact)-[z:isConnectedTo]->
      (f:Resource)
RETURN *;
```

For this query pattern, a full index was created, as well as three sub-indexes of length-3. These available indexes can be found in Table 9. We see that the cardinality of these indexes is very low compared to the size of the graph itself, which indicates that this pattern is very selective.

Similar to previous experiments, we run the benchmark without any indexes, with only the full-pattern index and a benchmark for each of the sub-pattern indexes. We again force the planner to use

| Name | Indexed pattern | Cardinality | Size on disk | Total data size | Initialization |
|---|---|---|---|---|---|
| *Graph* | - | – | 171.24 MiB | – | – |
| *Full* | `(:A)-[:V]->(:B)-[:W]->(:C)-[:X]->(:D)-[:Y]->(:E)` | 862 345 | 97.92 MiB | 59.21 MiB | 1 280 ms |
| *Sub*$_1$ | `(:A)-[:V]->(:B)-[:W]->(:C)-[:X]->(:D)` | 280 050 | 33.97 MiB | 14.96 MiB | 429 ms |
| *Sub*$_2$ | `(:B)-[:W]->(:C)-[:X]->(:D)-[:Y]->(:E)` | 295 337 | 35.55 MiB | 15.77 MiB | 445 ms |
| *Sub*$_3$ | `(:A)-[:V]->(:B)-[:W]->(:C)` | 111 532 | 10.42 MiB | 4.25 MiB | 235 ms |
| *Sub*$_4$ | `(:B)-[:W]->(:C)-[:X]->(:D)` | 102 812 | 9.72 MiB | 3.92 MiB | 231 ms |
| *Sub*$_5$ | `(:C)-[:X]->(:D)-[:Y]->(:E)` | 129 410 | 8.70 MiB | 4.94 MiB | 303 ms |
| *Sub*$_6$ | `(:A)-[:V]->(:B)` | 40 039 | 2.45 MiB | 0.92 MiB | 74 ms |
| *Sub*$_7$ | `(:B)-[:W]->(:C)` | 40 227 | 2.47 MiB | 0.92 MiB | 87 ms |
| *Sub*$_8$ | `(:C)-[:X]->(:D)` | 40 613 | 1.97 MiB | 0.93 MiB | 107 ms |
| *Sub*$_9$ | `(:D)-[:Y]->(:E)` | 40 220 | 1.84 MiB | 0.92 MiB | 107 ms |

**Table 6: The available indexes in the query experiment on independent data, with their respective cardinality, storage size, data size and initialization time. Data size reflects the actual size of the data in the index. *Graph* shows the graph size.**

| Name | **Memory-cached** | | | **Cold** | | | Max. intermediate |
|---|---|---|---|---|---|---|---|
| | First result | Last result | Speed-up | First result | Last result | Speed-up | state cardinality |
| *Baseline* | 1 345.49 ms | 5 814.97 ms | - | 1 934.50 ms | 6 107.95 ms | - | – |
| *Full* | 3.10 ms | 2 901.87 ms | ≈ 2.0× | 100.69 ms | 3 821.86 ms | ≈ 1.6× | 862 345 |
| *Sub*$_1$ | 5.70 ms | 7 784.36 ms | ≈ 0.7× | 117.51 ms | 7 382.92 ms | ≈ 0.8× | 4 316 687 |
| *Sub*$_2$ | 2 810.25 ms | 3 704.29 ms | ≈ 1.6× | 3 148.84 ms | 4 038.80 ms | ≈ 1.5× | 862 345 |
| *Sub*$_3$ | 3.31 ms | 8 595.40 ms | ≈ 0.7× | 65.06 ms | 8 640.75 ms | ≈ 0.7× | 4 284 072 |
| *Sub*$_4$ | 3.41 ms | 7 085.58 ms | ≈ 0.8× | 62.00 ms | 7 508.96 ms | ≈ 0.8× | 4 284 072 |
| *Sub*$_5$ | 3 894.59 ms | 4 828.22 ms | ≈ 1.2× | 4 543.00 ms | 5 498.45 ms | ≈ 1.1× | 862 345 |
| *Sub*$_6$ | 3.57 ms | 9 700.52 ms | ≈ 0.6× | 54.08 ms | 9 663.02 ms | ≈ 0.6× | 4 316 687 |
| *Sub*$_7$ | 3.60 ms | 8 135.46 ms | ≈ 0.7× | 49.90 ms | 8 639.04 ms | ≈ 0.7× | 4 316 687 |
| *Sub*$_8$ | 3.29 ms | 8 023.38 ms | ≈ 0.7× | 43.62 ms | 8 227.96 ms | ≈ 0.7× | 4 316 687 |
| *Sub*$_9$ | 2.75 ms | 8 421.97 ms | ≈ 0.7× | 43.60 ms | 8 921.94 ms | ≈ 0.7× | 4 316 687 |

**Table 7: The results of the query experiment on independent data. The left-most column holds the names of the indexes. The middle sections show the query performance in a memory-cached and cold scenario and the right-most column shows the maximum intermediate state cardinality during query execution.**

| Name | **Relationship addition** | | **Relationship deletion** | | Average speed-up |
|---|---|---|---|---|---|
| | Full index | Sub index | Full index | Sub index | |
| None | 0.362 ms | – | 0.824 ms | – | – |
| *Sub*$_1$ | 412.406 ms | 0.269 ms | 419.108 ms | 0.742 ms | ≈ 0.00× |
| *Sub*$_2$ | 94.959 ms | 0.297 ms | 94.883 ms | 0.953 ms | ≈ 0.01× |
| *Sub*$_3$ | 0.303 ms | – | 0.808 ms | – | ≈ 1.07× |
| *Sub*$_4$ | 152.848 ms | 0.323 ms | 152.430 ms | 1.006 ms | ≈ 0.00× |
| *Sub*$_5$ | 36.943 ms | 0.280 ms | 42.162 ms | 0.570 ms | ≈ 0.01× |
| *Sub*$_6$ | 0.245 ms | – | 0.596 ms | – | ≈ 1.41× |
| *Sub*$_7$ | 0.334 ms | – | 0.930 ms | – | ≈ 0.94× |
| *Sub*$_8$ | 48.862 ms | 0.204 ms | 34.820 ms | 17.250 ms | ≈ 0.01× |
| *Sub*$_9$ | 0.526 ms | – | 1.340 ms | – | ≈ 0.64× |

**Table 8: Results of the maintenance experiment on independent data. The rows show the amount of time required to update the index, given the presence of a sub-pattern index named in the left-most column.**

| Name | Indexed pattern | Cardinality | Size on disk | Total data size | Initialization |
|---|---|---|---|---|---|
| *Graph* | - | – | 20 947.05 MiB | – | – |
| *Full* | `(a)-[w]->(b)-[v]->(c)-[x]->(d)-[y]->(e)-[z]->(f)` | 2 320 | 0.45 MiB | 0.19 MiB | 424 610.96 ms |
| *Sub*$_1$ | `(a)-[w]->(b)-[v]->(c)-[x]->(d)` | 7 | < 0.01 MiB | < 0.01 MiB | 474.28 ms |
| *Sub*$_2$ | `(b)-[v]->(c)-[x]->(d)-[y]->(e)` | 12 323 | 1.58 MiB | 0.75 MiB | 6 830.69 ms |
| *Sub*$_3$ | `(c)-[x]->(d)-[y]->(e)-[z]->(f)` | 366 | 10.42 MiB | 4.25 MiB | 158.18 ms |

**Table 9: The available indexes in the query experiment on the YAGO data set, with their respective cardinality, storage size, data size and initialization time. Data size reflects the actual size of the data in the index. *Graph* shows the graph size.**

| Name | Last result | Max. iterm. cardinality | Speed-up (Baseline) | Speed-up (Manual) |
|---|---|---|---|---|
| *Baseline* | 440 538.84 ms | 43 143 933 | – | – |
| *Manual* | 371.19 ms | 24 079 | ≈ 1 187× | – |
| *Full* | 77.43 ms | 2 320 | ≈ 5 690× | ≈ 4.79× |
| *Sub*$_1$ | 17.24 ms | 2 320 | ≈ 25 553× | ≈ 21.53× |
| *Sub*$_2$ | 432 157.17 ms | 43 143 933 | ≈ 1× | ≈ 0.00× |
| *Sub*$_3$ | 437 814.00 ms | 43 143 933 | ≈ 1× | ≈ 0.00× |

**Table 10: The results of the query experiment on YAGO data set. The left-most column holds the names of the indexes. The middle sections show the query performance in a memory-cached and cold scenario and the right-most column shows the maximum intermediate state cardinality during query execution. Note that *Manual* is a manually optimized query plan without using an index. The right-most column denotes the speed-up compared to the *Manual* plan instead of *Baseline.***

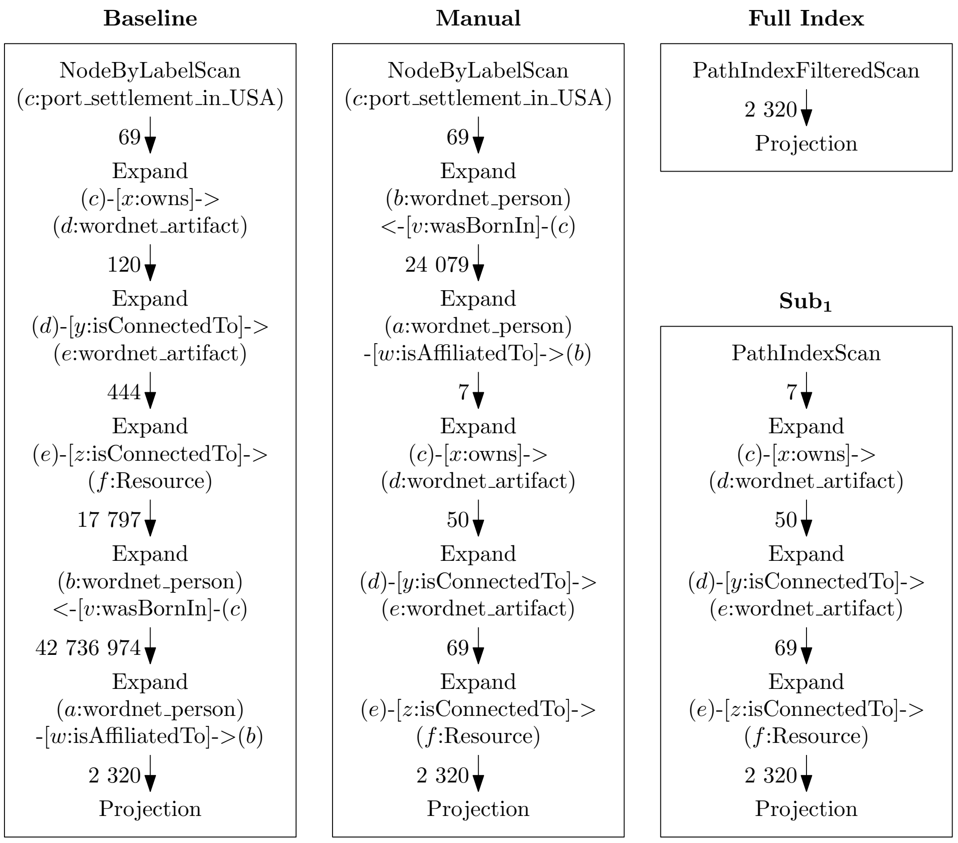
Baseline
NodeByLabelScan ($c$:port_settlement_in_USA)
69
Expand ($c$)-[$x$:owns]-> ($d$:wordnet_artifact)
120
Expand ($d$)-[$y$:isConnectedTo]-> ($e$:wordnet_artifact)
444
Expand ($e$)-[$z$:isConnectedTo]-> ($f$:Resource)
17 797
Expand ($b$:wordnet_person) <-[$v$:wasBornIn]-($c$)
42 736 974
Expand ($a$:wordnet_person) -[$w$:isAffiliatedTo]->($b$)
2 320
Projection
Manual
NodeByLabelScan ($c$:port_settlement_in_USA)
69
Expand ($b$:wordnet_person) <-[$v$:wasBornIn]-($c$)
24 079
Expand ($a$:wordnet_person) -[$w$:isAffiliatedTo]->($b$)
7
Expand ($c$)-[$x$:owns]-> ($d$:wordnet_artifact)
50
Expand ($d$)-[$y$:isConnectedTo]-> ($e$:wordnet_artifact)
69
Expand ($e$)-[$z$:isConnectedTo]-> ($f$:Resource)
2 320
Projection
Full Index
PathIndexFilteredScan
2 320
Projection
Sub$_1$
PathIndexScan
7
Expand ($c$)-[$x$:owns]-> ($d$:wordnet_artifact)
50
Expand ($d$)-[$y$:isConnectedTo]-> ($e$:wordnet_artifact)
69
Expand ($e$)-[$z$:isConnectedTo]-> ($f$:Resource)
2 320
Projection

**Figure 10: The *Baseline, Manual, Full* and *Sub*$_1$ query plans for the YAGO query, with intermediate cardinalities annotated. Filter operations are not visualized explicitly.**

a plan that contains these indexes so that we can compare their performance even though a baseline plan may be estimated to be faster.

The results of this experiment can be found in Table 10 and are illustrated in Figure 9. We see here that both the *Full* and *Sub*$_1$ index are very effective in reducing the maximum intermediate state cardinality, whereas the plans provided for the other indexes are hardly better than the baseline performance.

Due to the way we chose this pattern, we know that the cardinality estimator in the database has a hard time predicting the actual cardinality of this pattern, which may explain why it was unable to create a fast plan for the baseline performance. From the information presented, we conclude that the *Baseline* query plan was in fact a bad query plan, since it is possible to both construct the *Sub*$_1$ index using a query and then run the full query using this *Sub*$_1$ index in less than half a second, which is roughly 1 000 times faster than the current *Baseline* result.

Therefore we have manually planned an optimal plan for this query workload by looking at the actual cardinalities rather than estimates. This plan is called the *Manual* plan. It does not use any indexes. Yet we still see that *Full* and *Sub*$_1$ are significantly faster. This proves that the large speed-up factor was not only caused by a bad baseline plan, but in fact also by reducing the intermediate cardinality of the query plan using path indexes. Figure 10 visualizes the *Baseline*, *Manual*, *Full* and *Sub*$_1$ plans.

## 7.4 Using Path Indexes with GeoSpecies

We conclude our results with the following query on the GeoSpecies data set [3]. The query contains two nodes with a `Resource` label. Similar to the YAGO dataset, this label is applied to all nodes. From the relationship types we can see that these nodes will always denote locations. However, the GeoSpecies data set does not have a singular label for this type of node. Therefore we must apply this generic label.

```
MATCH (a:species_concept)-[x:is_expected_in]->
      (b:Resource)<-[y:was_observed_in]-
      (c:species_concept)-[z:is_expected_in]->
      (d:Resource)
RETURN *;
```

We compare three results on this data set. The first is the *Baseline* query plan. The second result uses an index on the entire query pattern. And since this query contains a sub-pattern twice, namely `(:species_concept) -[:is_expected_in]-> (:Resource)`, we also create an index on this specific sub-pattern. These indexes, along with their cardinality and size, are shown in Table 12.

The result set cardinality of this query is 334 126. Even though there is a structural correlation in this query, the query is not very selective. The maximum intermediate state during query execution occurs when producing the results. Therefore it is not possible to skip over high intermediate state while answering this query. As we can see in Table 11 and Figure 11, indeed the indexed plans have very similar performance to the *Baseline* plan, despite offering totally different query plans.

This shows our hypothesis that the main benefit of using path indexes is not in providing a faster way to output the result set, but mainly in reducing the computations required to arrive at the result set. Sequentially reading the results from the *Full* index is barely faster than producing the entire result set directly from the graph, as done by the *Baseline*. Additionally, the *Sub* query plan is slightly slower, even though part of the graph traversal is replaced by a sequential scan of the index.

| Name | Last result | Max. interm. cardinality |
|---|---|---|
| *Baseline* | 1 350.26 ms | 334 126 |
| *Full* | 1 172.95 ms | 334 126 |
| *Sub* | 1 425.78 ms | 334 126 |

**Table 11: Results of the GeoSpecies benchmark.**

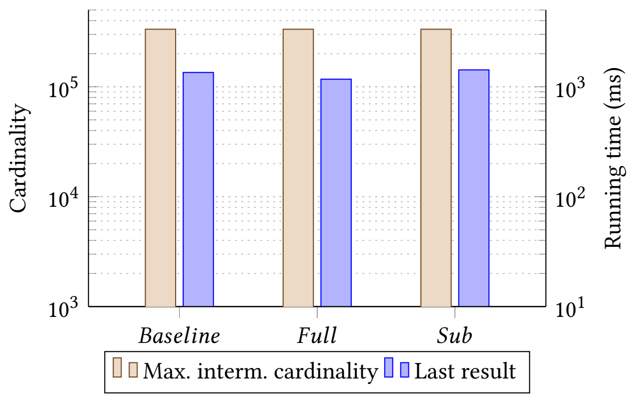

**Figure 11: The benchmark results from Table 11 on the GeoSpecies data set.**

## 8 CONCLUSIONS

We have found that selective path indexes can provide very significant query performance improvements for queries. This is especially true if the query engine would otherwise require computations on large intermediate state to arrive at the relatively small result set. In these scenarios, even though path indexes in general require exponential storage, these selective path indexes can be small relative to the total graph size. In terms of bang-for-buck, these type of scenarios are very well suited to our path indexes.

Further, we found that query-based maintenance can be an effective strategy in maintaining these queries. The major benefit over the methods proposed by De Jong [2] is that sub-pattern indexes may be used when they are available, but traversal-based query plans for maintaining the indexes are always available as a fallback. Other types of query execution optimizations, such as better plans due to an improved cardinality estimator, will also translate to faster maintenance computations.

We have shown three distinct operators that use our B$^+$-tree-based index. These operators allow the path index to be used both in leaf-operators, as well as extending the result set with a prefix search on the indexed pattern. Planning these operators correctly requires good cost heuristics and cardinality estimations.

During our study, we found that the most effective way to find good query plans using these path indexes is by analyzing and

| Name | Indexed pattern | Cardinality | Size on disk | Total data size | Initialization |
|---|---|---|---|---|---|
| *Graph* | - | – | 117.99 MiB | – | – |
| *Full* | `(a)-[x]->(b)<-[y]-(a)-[x]->(b)` | 334 126 | 32.13 MiB | 17.84 MiB | 4 042.54 ms |
| *Sub* | `(a)-[x]->(b)` | 24 814 | 1.54 MiB | 0.57 MiB | 467.53 ms |

**Table 12: The available indexes in the GeoSpecies data set, with their respective cardinality, storage size, data size and initialization time. Data size reflects the actual size of the data in the index. *Graph* shows the graph size.**

profiling the query plans without indexes to identify scenarios where computations on large intermediate state, which result in a smaller result set, can be skipped by pre-computing these paths and using a path index on these patterns. In the cases where we observed these conditions, path indexes were always able to speed up query evaluation performance.

## 9 FUTURE WORK

This study shows how important cardinality estimation is for good query plans. Interestingly, path indexes can provide accurate cardinality values for the patterns that they index. It is, however, not trivial to combine these exact cardinality values with the estimations provided by the cardinality estimation model in the query planner. Investigating this may yield better methods for more accurately predicting the cardinality of query plans, which can in turn help to improve the quality of query plans in general.

Path indexes have shown to be very effective in certain scenarios. However, their size requirements are exponential in the worst case and maintenance can be expensive as well. Therefore selecting which patterns to index is an interesting optimization problem. Cardinality estimation could also be an important aspect of an investigation into this problem.

## ACKNOWLEDGEMENTS

The author was employed by Neo4j for the duration of the project.

## REFERENCES

[1] Angela Bonifati, George H.L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying graphs*. Morgan & Claypool Publishers, 2018.
[2] Niels de Jong. Magpie (a maintainable graph pattern indexing engine): Towards a versatile path index for the industrial graph database. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2019.
[3] Peter DeVries. The geospecies knowledge base ontology. http://rdf.geospecies.org/geospecies.rdf.gz, 2009. Accessed in March 2019.
[4] George H.L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. In Ioana Manolescu, Evaggelia Pitoura, Amelie Marian, Sofian Maabout, Letizia Tanca, Georgia Koutrika, and Kostas Stefanidis, editors, *Advances in Database Technology - EDBT 2016*, pages 636–639. OpenProceedings.org, 1 2016.
[5] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.
[6] Neo4j Inc. Neo4j 3.5 source code. https://github.com/neo4j/neo4j/tree/3.5, 2019. Accessed in January 2020.
[7] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems (TODS)*, 25(1):43–82, 2000.
[8] Anton Persson. The shortcut index. Master's thesis, Lund University, Lund, Sweden, 2016.
[9] Jeroen Peters. Regular path query evaluation using path indexes. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2015.
[10] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 697–706, New York, NY, USA, 2007. Association for Computing Machinery.
[11] Jonathan M. Sumrall. Path indexing for efficient path query processing in graph databases. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2015.
[12] Jonathan M. Sumrall, George H.L. Fletcher, Alexandra Poulovassilis, Johan Svensson, Magnus Vejlstrup, Chris Vest, and Jim Webber. Investigations on path indexing for graph databases. In Pierre-Francois Dutot and Frederic Desprez, editors, *Euro-Par 2016*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 532–544, Germany, 1 2017. Springer.