Path Indexing in the Cypher Query Pipeline

Jochem Kuijpers
contact@jochemkuijpers.nl
TU Eindhoven, Netherlands

George Fletcher
g.h.l.fletcher@tue.nl
TU Eindhoven, Netherlands

Tobias Lindaaker
tobias.lindaaker@neo4j.com
Neo4j, Sweden

Nikolay Yakovets
n.yakovets@tue.nl
TU Eindhoven, Netherlands

ABSTRACT

We investigate how a state of the art *path index* can be integrated into the Cypher query pipeline of the industrial Neo4j graph database. We identify the characteristics of practical use-cases where application of path indexes is beneficial to query evaluation and performance of index maintenance. Through in-depth empirical evaluation, 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 significantly accelerate query performance. As such patterns arise naturally in querying on real graphs, we can conclude that path indexes are a valuable method for improving the performance of graph database systems in practice.

1 INTRODUCTION

A common operation in graph databases is pattern query evaluation, i.e., looking for all matches of a query graph in a data graph [1]. This operation searches for sub-graphs in the data with a structure that is constrained by the query. A state of the art index on paths has been introduced in prior work [7, 17]. It has been shown that this index can be effective in accelerating query evaluation by multiple orders of magnitude and can be effectively maintained as the underlying data graph is updated [4, 12]. Current graph databases struggle with scalability, as graphs continue to grow in size and complexity [14]. Path indexes are a promising technique to help address query performance in practice.

In this short paper, we present experiences gained from the practical integration of a path index into the Cypher query pipeline of the Neo4j graph database management system. Cypher is a de facto industry standard query language for graph databases; Neo4j is one of the most popular and widely-deployed graph databases in industry [8]. We explore practical use-cases where path indexes can significantly improve query processing performance, and analyse scenarios when this query acceleration is achieved through an in-depth empirical evaluation. We conclude that path indexes are most effective when used on *selective* patterns that allow the query planner to avoid high intermediate state cardinality and thus significantly accelerate query performance. This result is not immediately obvious for contemporary graph database systems, and to our knowledge has not been observed before. As such patterns arise frequently in applications due to correlations in the structure of real world graphs, we can conclude that path indexes are indeed a valuable and practical method for scaling graph data management in industrial systems.

While our experiments were made using Neo4j, the results are immediately applicable to any graph database management system that implements Cypher or any other graph query language such as SPARQL or G-CORE that supports matching path patterns comprising a sequence of node and edge labels [1].

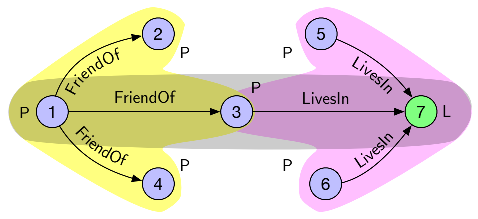

Figure 1: Example of a simple property graph. Nodes labeled *P* represent *Person* nodes. The node labeled *L* represents a *Location*. Shaded paths represent path indexes.

2 BACKGROUND

Property Graphs. Neo4j uses the *property graph* data model [1, 8]. A property graph is a graph where: every node can have an arbitrary number of labels; every relationship is directed between two nodes and has exactly one type; there may be multiple relationships of the same type between the same nodes, i.e. it is a multi-graph; and, every node and relationship can have an arbitrary number of associated attribute-value pairs. As an example, Figure 1 shows a property graph (suppressing attribute-value pairs) representing a small social network.

Cypher Query Language. Cypher is a declarative graph query language that is loosely based on SQL [8]. It contains familiar SQL keywords such as `WHERE` that function essentially 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 arrows. Query variables are declared by their inclusion in one or more pattern expressions and can be used in other clauses. For example, `(x:Person)` matches all nodes `x` with label `Person` and `(p:Person)-[r:Lives_In]->(c:City)` matches all directed relationships `r` from nodes `p` to nodes `c`, with labels `Lives_In`, `Person`, and `City`, resp.

Path Patterns. A *path pattern* is a sequence of alternating node labels and relationship patterns starting and ending with a node label. A *relationship pattern* contains both a relationship type and its direction (either forward, $\rightarrow$, or reversed, $\leftarrow$). E.g., given the node labels $\{A, B\}$ and the relationship type R, the following notation describes a path pattern of length 2: $\langle A, (R, \rightarrow), B, (R, \leftarrow), B \rangle$, counting relationships to determine the length.

A path pattern describes a set of constraints that can be applied to paths of the same length. Given a k-length path pattern

$\langle N_0, (R_1, D_1), \ldots, (R_k, D_k), N_k \rangle$, then a path $\langle n_0, r_1, \ldots, r_k, n_k \rangle$ in the graph satisfies the pattern if and only if all nodes n_i have a N_i label for $0 \leq i \leq k$ and all relationships r_i are of type R_i and direction D_i for $1 \leq i \leq k$. As an example, the pattern $\langle P, (\mathit{FriendOf}, \rightarrow), P, (\mathit{LivesIn}, \rightarrow), L \rangle$ would yield one query result on the graph in Figure 1.

Prior Work on Path Indexing. Querying graph databases using indexes is a complex field of study. A detailed contemporary survey of existing techniques can be found in [1]. The *k-path index*, introduced by Fletcher et al. [7, 13, 16, 17], demonstrated that significant orders-of-magnitude query performance improvements are possible with path indexing. Here, the intuition is to index all, or a selected subset of all, paths of length up to k, where k is the count of relationships in the indexed paths. Fletcher et al. built the k-path index by concatenating edges in the graph and storing the resulting paths in a relational database [7]. The resulting table was indexed and used to speed up path queries.

Sumrall et al. [17] engineered a k-path index directly implemented using a B$^+$-tree and demonstrate the potential for accelerated query processing. Sumrall also studied how one could index paths with specific patterns (e.g., as given in a workload) rather than all paths of length k.

Persson [12] 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 maintenance computations on graph updates, which could not be afforded in the described use-case.

De Jong [4] demonstrated how indexes can be more efficiently maintained by maintaining sub-patterns as separate indexes. This allows for a significant speed-up of index maintenance, at the expense of increased storage overhead.

Boncz et al. [3] and Luo et al. [10] give a broader overview of graph query processing with indexes, both in contemporary systems and in research, as well as the role of query selectivity in effective graph query processing. To our knowledge, ours is the first study of path indexing in industrial systems.

Path Indexes. A *path index* is a data structure that indexes all paths in the data graph which satisfy a chosen path pattern. This path pattern is called the *indexed pattern*. The paths are not stored directly in the path index, instead a sequence of references is stored. For every indexed k-length path $\langle n_0, r_1, \ldots r_k, n_k \rangle$, the index stores references to the nodes and relationships of the matching paths as a sequence of identifiers: $\langle n_0^{id}, r_1^{id}, \ldots, r_k^{id}, n_k^{id} \rangle$, where n_i^{id} and r_i^{id} are the i-th identifiers of nodes and relationships in the path respectively.

These sequences of references are converted into a single key by concatenating the fixed-width identifiers (8 bytes each), which are stored in a B$^+$-tree. This allows logarithmic-time location, insertion and deletion of entries in the index. Since the identifiers are concatenated, the B$^+$-tree also supports prefix searches. Given the first m elements of a path, we can locate the first entry that starts with this prefix, and scan all paths that match the prefix in linear-time with respect to the number of results returned. The worst-case space complexity for the index is $O(E^k)$ where E is the number of edges in our graph and k the length of the path pattern that is indexed. While the size of the index is linear to the number of identifiers in the key, each index is defined for a fixed size key which keeps the size bounded. It is also worth noting that when indexes are chosen to represent selective patterns, the size of the index will naturally remain small.

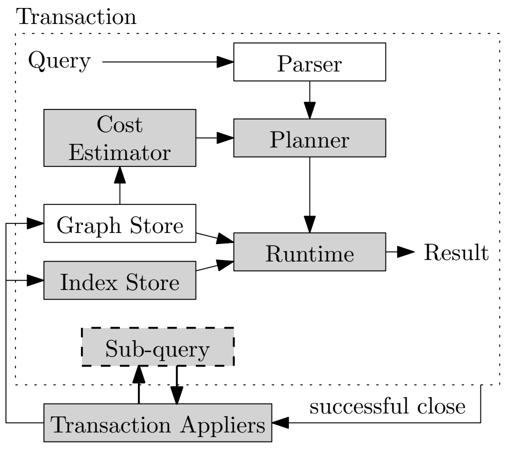

Figure 2: An overview of the query pipeline architecture. Shaded boxes represent our changes, while dashed outlines represent a new component.

The shaded parts on Figure 1 represent path indexes. An index on $\langle P, (\mathit{FriendOf}, \rightarrow), P, (\mathit{LivesIn}, \rightarrow), L \rangle$ (shaded grey) contains just the single path through nodes $\langle 1, 3, 7 \rangle$, an index on $\langle P, (\mathit{FriendOf}, \rightarrow), P \rangle$ (yellow) will contain the paths through nodes $\{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle\}$ and an index on $\langle P, (\mathit{LivesIn}, \rightarrow), L \rangle$ (purple) will contain the paths through nodes $\{\langle 3, 7 \rangle, \langle 5, 7 \rangle, \langle 6, 7 \rangle\}$.

3 USING PATH INDEXES

Our Extensions. We next discuss the modifications and extensions to the query pipeline architecture necessary to support path indexes (Figure 2). Firstly, path index operators have been added to the *Planner* component which can scan or selectively read paths from path indexes. This also requires new costing heuristics in the *Cost Estimator* for these operators. The new operators are also implemented in the *Runtime* component as runtime-specific operators that perform the read operations on the path index store. Further the *Index Store* was modified to hold the new type of path index.

When a transaction is committed, the update commands are translated into paths that have to be added to or removed from the path indexes through sub-queries on the path patterns of those indexes. These sub-queries are inserted into the query pipeline as normal and might use other path indexes to resolve the query, depending on the context. Full details can be found in the extended report [9].

4 IMPLEMENTATION

We next discuss 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.

4.1 Modified Pipeline Components

The implementation of our path index into the query pipeline required modifications in several components. Figure 2 shows an overview of the components, where a shaded background means the component required some modification. The dashed block “Sub-query” 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 global data statistics.

4.1.1 Query-based path index Maintenance. De Jong [4] 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 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 sub-patterns of 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.

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.

There are some important things to consider. Firstly, Neo4j 3.5[1] binds a transaction along with its transaction state to the thread that opened the transaction. That means that, while we are applying the outer transaction, the inner query we want to execute is filtered through the transaction state during maintenance. As a work-around to this behavior, we store the transaction state of the outer transaction and reset it for the maintenance queries. After processing the updates, we restore the old transaction state such that other Neo4j operations are not affected.

Further, since we are in the middle of applying a transaction, some of the path indexes might not be up-to-date while other path indexes may already have been updated. We introduce a sort order of path indexes by length, small-to-large, to ensure that any maintenance query plan for a path of length k will itself only include path indexes of lengths smaller than k, which by then have already been updated. When we remove a relationship from the graph, we want the maintenance query results to still include it so we know which paths to remove from the index. That is why the query for removals is done before modifying the underlying data. For relationship additions, the reverse holds. We want to include it in the maintenance query results in order to add these paths to the index, therefore we must query these after the underlying data has been updated.

Similar maintenance steps can be applied for node label updates. Node additions and removals can be ignored, as those are only allowed for disconnected nodes, making it impossible to affect path index maintenance.

4.1.2 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 [16] 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 1 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 $T_{old}$.
```

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 $I$
```

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

5 EXPERIMENTAL SETUP

Baseline Planner Extension. The 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 few ways to selectively scan or seek based on relationship types. Our path index implementation will have these abilities. Therefore we apply an extension to the baseline planner which includes an operator that scans relationships by type. It is introduced with the same cost heuristics as the most similar node-based operator. Our path index query plans are compared with this extended baseline.

Hardware and Software. Our experiments were performed on a server with four Intel Xeon E5-4610 v2 CPUs running at 2.30GHz, 500GB of DDR3 RAM at 1600MHz. We used its 260GB NVMe SSD for data storage. The server ran Ubuntu 16.04.3 LTS and the Oracle Java (TM) SE Runtime Environment (version 1.8.0-151). We prototype on the Neo4j 3.5 community code base [11].

Methodology. Our experiments ran with a pre-allocated heap of 100GB. Each experiment ran until running time converges, which indicates that hot code paths were optimized by the JVM. Then we ran the experiment five times, triggering a garbage collection cycle between each run and flushing the data from memory without restarting the JVM as this would lose hot code path optimizations. We then discarded the highest and lowest running time and averaged the remaining three results. For data set sizes, we summed the total data file sizes on disk. Path index stores were measured separately and transaction logs were excluded.

Datasets. We use four data sets in our experiments: two synthetically generated and two real-world data sets. The first synthetic data set is referred to as the *correlated* data set, as it has high structural correlation. It has 125K nodes and 12.6M relationships and was created by interconnecting 25 000 copies of the same path, making it a highly selective pattern. The second synthetic data set is referred to as the *independent* data set as there are no structural correlations in the connections between nodes. It has 250K nodes and 5M relationships. The first real-world data set is the YAGO data set [15], containing 77M nodes and 100M relationships. The second real-world dataset is the GeoSpecies data set [5], containing 225K nodes and 1.5M relationships. Coming from distinct application domains, these graphs allow us to gain practical insights into the robustness of our methods.

6 RESULTS

Our hypothesis is that our path index is not suited for application on high-cardinality path patterns, since the worst-case space requirement is exponential in path length. Indeed, we observe that path indexes are especially useful for highly selective paths on correlated data, as the cost of the path index is very low compared to the computation of intermediate state that can be *skipped* by using the index.

To test this hypothesis, we first run two controlled scenarios of queries on highly correlated and uniformly distributed synthetic data sets. We generate our own data sets, rather than using a benchmark such as LDBC SNB [6], so as to finely control the structure of the data. This was sufficient for our goals here, but we note that off-the-shelf generators such as gMark could also have been used [2]. Then, we verify our findings by applying our path index to two real-world scenarios: a selective, correlated path query and a high-cardinality path query.

Finally, we show that path indexes can be applied to index maintenance in some cases, and the effect of selective, correlated index path patterns.

6.1 Synthetic Query Benchmarks

Our first experiments show that choosing the right path index can significantly improve query performance. The available indexes on this data set are described in Figure 3 (Correlated synthetic). The query pattern matches that of the full index. The result of this query when planned with different indexes can be seen in Figure 4 (Correlated synthetic). The full index is clearly the best as it essentially pre-computes the answer. However, sub-index S_1 has similar performance for a smaller index, which may offer more re-use capabilities.

The second experiment, illustrated in Figure 4 (Independent synthetic), shows the same technique applied to a uniformly distributed data set. The indexes available here are described in Figure 3 (Independent synthetic). Because there is no selective, structural correlation in this data set, the query produces many more results. Indexing these results, even in sub-indexes, provides no significant speed improvement.

Both of these experiments show dependency between the running time and the maximum intermediate cardinality. This indicates that indexing selective patterns can significantly reduce the maximum intermediate cardinality during query evaluation, and thus the running time of the query.

6.2 Real-world Query Benchmarks

After findings on synthetic datasets, we applied our technique to real-world data sets. Our first dataset, YAGO [15], has a file containing query workloads. We used the cardinality estimation model of the Neo4j planner that assumes independence between elements to find the query that was most mis-predicted, as our assumption was that this query would be highly correlated, since this is exactly the type of query that will yield mis-predictions by this cardinality estimation model.

We then applied path indexes that matched parts of the query to speed up query evaluation [9]. The heuristic cost estimator we used was built on the assumption of independence. For the YAGO experiments, the resulting query plans were of insufficient quality. We have manually created better query plans to show what an improved cost estimator could achieve with our indexes. The indexes are described in Figure 3 (YAGO dataset) and the benchmark results are shown in Figure 4 (YAGO).

The full index on the query pattern significantly speeds up query evaluation time compared to our manually optimized baseline. The plans using smaller sub-indexes further improve performance, even though it requires more steps to fully answer the query in these plans. This can be explained by the reduction of intermediate cardinality in later stages of the operator tree. The high cardinality of the S_2 and S_3 plans is caused by the first node scan operator which is reduced early on in the execution of the query plans, this explains the faster execution time despite the initially higher cardinality compared to the F plan. The path index efficiently produces the full path on this reduced state, achieving fast total execution times.

We have also applied this to another data set with a less selective query, expecting our path index would not be able to speed up query evaluation performance as much. And indeed, our experiment results (shown in Figure 3 and Figure 4 under

Correlated synthetic

Name	Indexed pattern	Cardinality	Size (MB)
G	-	–	413.97
F	(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)	25 000	3.92
S_1	(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)	25 000	3.17
S_2	(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)	25 000	3.17
S_3	(:A)-[:X]->(:A)-[:X]->(:A)	12 524 000	970.56
S_4	(:A)-[:X]->(:A)-[:Y]->(:B)	25 000	2.39
S_5	(:A)-[:Y]->(:B)-[:X]->(:A)	6 274 500	471.59
S_6	(:A)-[:X]->(:A)	6 299 500	364.95
S_7	(:A)-[:Y]->(:B)	6 274 500	250.27
S_8	(:B)-[:X]->(:A)	25 000	1.55

Independent synthetic

Name	Indexed pattern	Cardinality	Size (MB)
G	-	–	171.24
F	(:A)-[:V]->(:B)-[:W]->(:C)-[:X]->(:D)-[:Y]->(:E)	862 345	97.92
S_1	(:A)-[:V]->(:B)-[:W]->(:C)-[:X]->(:D)	280 050	33.97
S_2	(:B)-[:W]->(:C)-[:X]->(:D)-[:Y]->(:E)	295 337	35.55
S_3	(:A)-[:V]->(:B)-[:W]->(:C)	111 532	10.42
S_4	(:B)-[:W]->(:C)-[:X]->(:D)	102 812	9.72
S_5	(:C)-[:X]->(:D)-[:Y]->(:E)	129 410	8.70
S_6	(:A)-[:V]->(:B)	40 039	2.45
S_7	(:B)-[:W]->(:C)	40 227	2.47
S_8	(:C)-[:X]->(:D)	40 613	1.97
S_9	(:D)-[:Y]->(:E)	40 220	1.84

YAGO dataset

Name	Indexed pattern	Cardinality	Size (MB)
G	-	–	20 947.05
F	(a)-[w]->(b)-[v]->(c)-[x]->(d)-[y]->(e)-[z]->(f)	2 320	0.45
S_1	(a)-[w]->(b)-[v]->(c)-[x]->(d)	7	< 0.01
S_2	(b)-[v]->(c)-[x]->(d)-[y]->(e)	12 323	1.58
S_3	(c)-[x]->(d)-[y]->(e)-[z]->(f)	366	0.01

GeoSpecies dataset

Name	Indexed pattern	Cardinality	Size (MB)
G	-	–	117.99
F	(a)-[x]->(b)<-[y]-(a)-[x]->(b)	334 126	32.13
S	(a)-[x]->(b)	24 814	1.54

Figure 3: The available indexes on the benchmarked datasets with their cardinality and storage size. Here, G denotes the size of the whole graph, F the index for the full path, and S indexes for sub-paths.

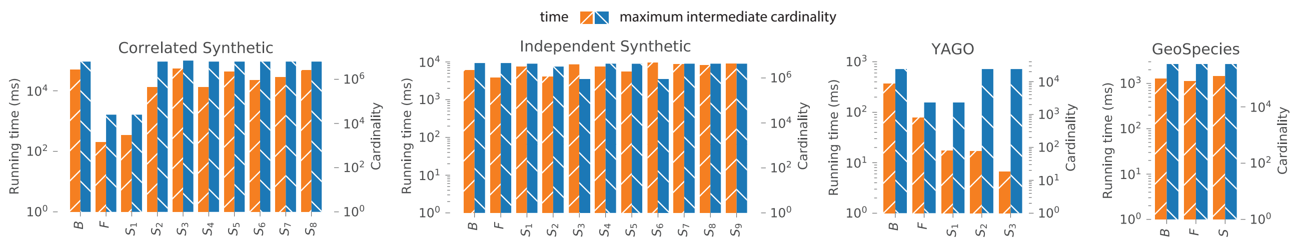

Figure 4: Benchmark results on the data sets. B denotes the baseline, F indexing the full path, and S indexing sub-paths.

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×

(a) Correlated data

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×

(b) Independent data

Figure 5: Results of the maintenance experiment on correlated (left) and independent (right) 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.

GeoSpecies) show that, because the result cardinality is the highest cardinality in the query evaluation, our path index was not able to *skip over* large intermediate cardinalities, and thus no real performance gain was achieved.

6.3 Maintenance Using Sub-Indexes

Not only can sub-pattern indexes provide performance benefits during query execution. As De Jong [4] 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 then also be used to speed up maintenance when applicable.

In this experiment, we first look at the performance benefits on our synthetic correlated data set for index maintenance, as we provide one of the sub-pattern indexes from Table 3 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. Fig. 5 (a) 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 for the *Full* index maintenance. This sometimes results in a slower maintenance plan as only some query plans are considered. We may assume that the planner would not use the sub-index for maintenance in that case, if given the choice, though the figures give an indication of the effect on maintenance of the sub-index. 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.

We then perform the same set of transactions, by removing an Y-labeled relationship in a transaction and adding it in another transaction, leading to index maintenance on the synthetic independent data set. We observe that similar modest speed improvements can be achieved the sub-pattern indexes on the full index maintenance in Fig. 5 (b), while the forced plans for some sub-indexes perform considerably worse as well.

7 LESSONS LEARNED

Indexing paths in graphs often makes sense in practice. Since the number of (potential) paths in a graph grows exponentially with path length, it might seem too prohibitive (wrt. worst-case space complexity) to index path patterns in large graphs. In this work, on the contrary, we found out that, in practice, indexing *strategically-chosen* path patterns can, in fact, greatly improve query performance in both synthetic and real datasets at a *small* storage overhead. The practicality of this approach is underscored by our integration of our work in the Neo4j system.

Patterns with high structural correlation are most beneficial to index. Patterns where there is high correlation in the connections between the nodes in the data will result in a relatively low number of paths matching the pattern. The number of edges that have to be explored to match the same pattern through direct traversal of the graph would be substantially larger. In such situations the cardinality of the intermediate result is substantially larger than cardinality of matches to the whole pattern. Our experiments show that when correlated patterns are indexed, this high cost of computing these intermediate results of high cardinality is avoided. Furthermore the size of the index for such a highly selective pattern over correlated data is small as well. This turns out to be a sweet spot for path indexes, where the benefit of the index is high and the overhead of the index is low.

In contrast, for patterns matching uncorrelated data, the size of the index is proportional to the cardinality of the intermediate result, which in many cases grow exponentially in the size of the underlying graph. In these cases we experience not only a prohibitively large storage overhead for the index, but also no tangible performance benefit, since enumerating the paths from the index is proportional to enumerating the paths by direct traversal of the underlying graph.

Patterns to be indexed should be chosen with care. Hence, one should take advantage of structural correlations which naturally occur in graphs in order to choose path patterns that have (1) low cardinality and (2) help to cut down on the cardinality of intermediate results during the evaluation of queries in the workload. Finding path patterns that satisfy both (1) and (2) is not trivial and is ultimately a constrained optimization on the given workload and usable storage.

8 CONCLUSIONS

We have reported on our practical experiences integrating a state of the art path index into the query processing pipeline of Neo4j, a popular industrial graph database. Through extensive empirical study, we found that selective path indexes can greatly accelerate query evaluation performance. This is especially true in those cases where the query engine would otherwise require computations on large intermediate state to arrive at a relatively small result set. In these scenarios, which arise commonly in practice due to correlations in the structure of real world graphs, we have shown that even though path indexes in the worst case require exponential storage, these selective path indexes can be very small relative to the total graph size. These are optimal scenarios for path indexes since the path index is able to provide a significant performance improvement with a low space overhead. Our extended report [9] contains further details and results, such as technical aspects of the integration into Neo4j, the technical challenges encountered, and the engineering lessons learned.

Looking ahead, there are several interesting directions for further research. We close by indicating two of these: (1) investigate more deeply query planning in the presence of path indexes, including cardinality estimation and costing techniques for path indexes; and, (2) study methods for selecting which patterns to index, balancing space costs and performance benefit, e.g., with respect to a given query workload.

REFERENCES

[1] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying graphs.* Morgan & Claypool Publishers.

[2] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.

[3] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. 2014. Advances in Large-Scale RDF Data Management. In *Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project.* LNCS, Vol. 8661. 21–44.

[4] Niels de Jong. 2019. *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.

[5] Peter DeVries. 2009. The GeoSpecies Knowledge Base ontology. http://rdf.geospecies.org/geospecies.rdf.gz. Accessed in March 2019.

[6] Orri Erling et al. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD.* 619–630.

[7] George Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *EDBT.* 636–639.

[8] Nadime Francis et al. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD.* 1433–1445.

[9] Jochem Kuijpers. 2020. *Path Indexing in the Cypher Query Pipeline.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[10] Yongming Luo et al. 2012. Storing and Indexing Massive RDF Datasets. In *Semantic Search over the Web.* Springer, 31–60.

[11] Neo4j Inc. 2019. Neo4j 3.5 source code. https://github.com/neo4j/neo4j/tree/3.5. Accessed in January 2020.

[12] Anton Persson. 2016. *The Shortcut Index.* Master's thesis. Lund University, Lund, Sweden.

[13] Jeroen Peters. 2015. *Regular path query evaluation using path indexes.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[14] Siddhartha Sahu et al. 2019. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal* (2019).

[15] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A Core of Semantic Knowledge. In *WWW.* 697–706.

[16] Jonathan M. Sumrall. 2015. *Path indexing for efficient path query processing in graph databases.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[17] Jonathan M. Sumrall et al. 2016. Investigations on path indexing for graph databases. In *PELGA @ Euro-Par.*