Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Frank Tetzel, Hannes Voigt, Marcus Paradies, Wolfgang Lehner

An Analysis of the Feasibility of Graph Compression Techniques for Indexing Regular Path Queries

Erstveröffentlichung in / First published in:

SIGMOD/PODS'17: International Conference on Management of Data, Chicago 14.-19.05.2017. ACM Digital Library, Art. Nr. 9.

DOI: https://doi.org/10.1145/3078447.3078458

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-794494

SLUB Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 11, ISBN 978-1-4503-5038-9
https://doi.org/10.1145/3078447.3078458

An Analysis of the Feasibility of Graph Compression Techniques for Indexing Regular Path Queries

Frank Tetzel
Dresden Database Systems Group
Technische Universität Dresden
frank.tetzel@tu-dresden.de

Hannes Voigt
Dresden Database Systems Group
Technische Universität Dresden
hannes.voigt@tu-dresden.de

Marcus Paradies
SAP SE
marcus.paradies@sap.com

Wolfgang Lehner
Dresden Database Systems Group
Technische Universität Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

Regular path queries (RPQs) are a fundamental part of recent graph query languages like SPARQL and PGQL. They allow the definition of recursive path structures through regular expressions in a declarative pattern matching environment. We study the use of the K^2-tree graph compression technique to materialize RPQ results with low memory consumption for indexing. Compact index representations enable the efficient storage of multiple indexes for varying RPQs.

KEYWORDS

Regular Path Queries; Index Compression; Graph Processing

1 INTRODUCTION

Graph data management has become a major topic in the database community, in research as well as in industry. ICIJ's Panama Papers investigation[1], which used graph database technology, is a recent illustration of this trend [7]. Graph database systems are particularly suitable for querying multi-hop connections between entities, e.g., multi-hop stakeholder relationships between offshore profits and potential tax evaders. Commonly such queries are referred to as regular path queries (RPQs). RPQs are an important query class for graph database systems [20] and a major component of most graph query languages, such as SPARQL [19] and PGQL [18].

RPQs allow declarative querying of multi-hop connections, i.e., paths in a graph specified by a regular expression. More precisely, an RPQ finds all distinct pairs of vertices in a graph that are connected by at least one path conforming to the regular expression. Regular expressions of RPQs are formulated based on the set of edge labels (e.g., is-stakeholder-of) in the graph. A path conforms to a regular expression, if the concatenation of all edge labels along the path from start to end is in the language accepted by the regular expression. Regular expressions can be evaluated by traversing the data graph. An automaton representing the regular expression [4] restricts the edges that the traversal can use during execution.

With the extension of relational database systems with built-in graph data management support [15] an efficient evaluation of RPQs on graphs stored in relational structures becomes essential. Most graph engines based on a relational storage structure use a single table to store the edges of a graph. A graph traversal, which is necessary for evaluating the RPQ, results in a number of consecutive self-joins on the edge table. The number of required join operations depends on the graph diameter and the RPQ specification. Additionally, these joins typically produce large intermediate results even if the RPQ is selective, effectively making RPQ evaluation on a relational storage an expensive operation. Various techniques have been proposed to increase the efficiency of RPQ evaluation, e.g., dedicated query planning [22], adaptive traversals [14], exploiting selective labels [12], and adjacency indexing [11].

One particularly lightweight approach for speeding up RPQ evaluation is simply storing all distinct pairs of vertices of an RPQ in a *materialized reachability index* (MR-index). Such an MR-index allows answering the indexed RPQ solely by performing index lookups and reduces the number of necessary joins of every RPQ containing the indexed RPQ. An MR-index could be created like a regular secondary index by the database administrator or utilized online by a query optimizer for caching reachability results [22]. While this approach is appealing because it is simple and easy to implement, it comes with quadratic space complexity as potentially there could be paths between any pair of vertices in the graph. Hence, the feasibility of this approach depends on how efficient the reachability information is stored and which queries are actually indexed. Recent advances in graph compression suggest that reachability information can be stored with just 1 to 3 bit(s) per vertex pair [6], which can even make indexing of unselective RPQs feasible in terms of space consumption.

In this paper, we investigate the feasibility of graph–compression-based MR-indexes for RPQ evaluation in main-memory column stores. Therefore, we evaluate numerous RPQs on the LDBC dataset. We compare an in-memory column store baseline with uncompressed MR-indexing and compressed MR-indexing. Based on the measurements of query processing times, query selectivity, and storage consumption, we derive guidelines for the beneficial use of

[1] https://panamapapers.icij.org/

©2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *GRADES'17, Chicago, IL, USA.*
DOI: http://dx.doi.org/10.1145/3078447.3078458

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 11, ISBN 978-1-4503-5038-9
https://doi.org/10.1145/3078447.3078458

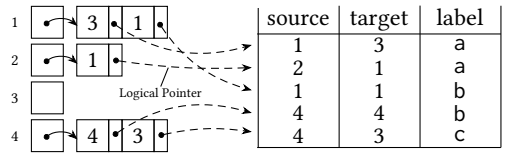


Figure 1: Relational, in-memory graph storage.

MR-indexes for RPQ processing in main-memory column stores. The contributions of this paper are:

- We conduct the first study of graph compression in the context of RPQ processing.
- We reveal the pros and cons of indexing RPQs with the help of graph compression.
- We outline where graph compression can provide a benefit for RPQ processing and were not.

In Section 2, we discuss the basics of RPQ processing. In Section 3, we detail the MR-index approaches that we consider in this study. We report on the experimental study and results in Section 4. Finally, Section 5 outlines related work and Section 6 concludes the paper.

2 RPQ PROCESSING IN COLUMN STORES

RPQs can operate on a range on different graph data models, such as the RDF and the Property Graph data model. For our investigation of MR-indexes, we focus on directed, labeled multigraphs. Formally, a directed, labeled multigraph is a graph $G = (N, E, \Sigma)$ where N refers to the set of vertices, Σ is the set of labels, and E is the set of edges with $E \subseteq N \times \Sigma \times N$.

Primary graph storage: We assume a relational, in-memory column-store representation of the graph as described in [11]. The graph is represented by an edge table with three columns *source*, *target*, and *label*. For faster adjacency lookups, the edge table is indexed as adjacency lists in forward and in reverse direction of the edges. Figure 1 illustrates the setup without the reverse index.

RPQs: An RPQ is given by a regular expression. Let R be a regular expression over the alphabet Σ and $L(R)$ the language described by the regular expression. Let $p = \langle v_1, e_1, ..., e_{n-1}, v_n \rangle$ be a path where $v_i \in N$, $e_j \in E$ and $e_j = (v_j, l_j, v_{j+1})$ for $i \in [1, n], j \in [1, n)$. The label sequence $l_1 \dots l_{n-1}$—the concatenation of all edge labels on the path—is called the label of path p, denoted as $\lambda(p) \in \Sigma^*$. A query $Q_R(G)$ searches for all distinct vertex pairs (u, v) such that there is at least one path p from a vertex u to a vertex v in the graph G which satisfies the regular expression R, i.e., $\lambda(p) \in L(R)$.

RPQ processing: For baseline RPQ processing, we assume automaton-based query plans as described in [13] without product automaton construction [12]. Essentially, the regular path expression of a given RPQ is converted into a deterministic finite automaton (DFA). Figure 2 depicts the DFA for the regular expression `(ab)+`. To process the RPQ the system performs a breadth-first search (BFS). During the BFS the DFA is used to track the current state of the regular expression for each exploration path. The exploration is restricted by the available transitions in the automaton such that the BFS traverses only edges that have labels matching a label of a valid state transition in the DFA for the current exploration path.

Every exploration that reaches a final state in the DFA yields one result of the RPQ. The exploration does not necessarily end with reaching a final state. Depending on the regular expression the final state of the DFA may have outgoing transitions. If an exploration path reaches the same vertex in the same DFA state a second time, it can be safely terminated as it will not produce additional results. The BFS terminates if no further exploration paths are left.

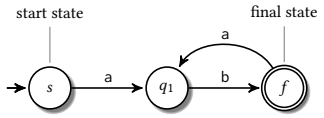


Figure 2: DFA for regular expression $(\mathsf{a}\ \mathsf{b})^+$.

The described query processing is equivalent to a left-deep tree of edge table self joins. However, our baseline query processing performs the BFS based on the adjacency list instead of using joins, which allow for faster traversals [11]. The edge table itself is only accessed for label lookups.

Remark: Our baseline RPQ evaluation strategy omits many well-known optimizations proposed for RPQ evaluation and is not meant to compete with these approaches. The focus of our study is on MR-indexes and in particular the feasibility of graph compression techniques to implement MR-indexes.

3 MR-INDEXES

An MR-index is the simplest form of reachability indexation. Given an RPQ R, the MR-index based on R contains all vertex pairs $(u, v) \in Q_R(G)$ that are in the result set of R. The result set of each RPQ forms a reachability graph $G_R(V, Q_R(G))$—a simple directed graph. The size of the reachability graph depends on the selectivity of the RPQ. In the worst case RPQs have quadratic result complexity, which gives MR-indexes a quadratic space complexity. In practice, the space consumption of an MR-index also depends on the data structure used to implement the index. Therefore, we consider an uncompressed and a compressed MR-index. Both variants include a dictionary, which filters out all vertices without an edge in the reachability graph and maps the remaining vertices $\{v \mid \exists e \in Q_R(G).\ e = (v, \cdot) \vee e = (\cdot, v)\}$ to a dense domain.

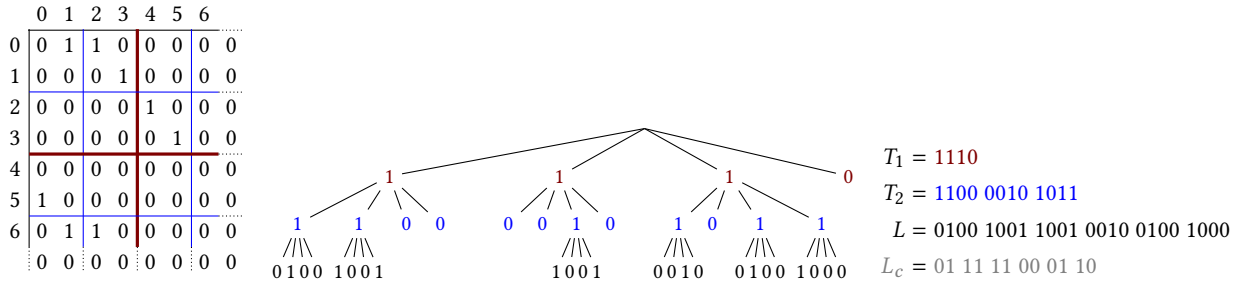
Uncompressed MR-index: The uncompressed MR-index uses an adjacency list to store the reachability graph. Adjacency lists are widely used as graph representation in many systems for primary graph storage. However, adjacency lists are not space-efficient as they require multiple bytes per edge—in our setup at least 64 bits per edge. For RPQ with large result sets an uncompressed MR-index can quickly become too large to fit into the memory the user is willing to reserve for indexing.

Compressed MR-index: The compressed MR-index utilizes a lossless graph compression technique K^2-trees [5] to store the reachability graph. Various graph compression techniques have been proposed in recent years, e.g., [2, 5, 8, 9] and allow storing large graphs with less than one byte per edge. In practice K^2-trees [5] have shown to offer the best compression ratio.

K^2-trees in detail: The graph compression technique K^2-tree provides a succinct encoding of the adjacency matrix of a graph. The adjacency matrix is recursively decomposed following a Quadtree-like strategy into k^2 square-shaped sub-matrices per decomposition step. Each sub-matrix is encoded in a k^2-ary tree. The root of the tree represents the complete adjacency matrix. Each internal node of the tree represents a sub-matrix and is assigned with a value.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 11, ISBN 978-1-4503-5038-9
https://doi.org/10.1145/3078447.3078458



(a) Adjacency matrix of a graph. (b) Conceptual K^2-tree with $k = 2$. (c) Bit representation of K^2-tree.

Figure 3: Example of adjacency matrix and the corresponding K^2-tree with its bit representation.

The node value is 0 iff all values in corresponding sub-matrix are 0, otherwise the node value is 1. Nodes with value 1 have k^2 successors in the tree, i.e., they are further decomposed. Nodes with value 0 have no successors—this is where the compression takes place. The children of each node are ordered from left to right and from top to bottom (z-order). Figure 3b shows the tree for the adjacency matrix shown in Figure 3a.

Internally the tree is stored in a variant of the LOUDS (level-ordered unary degree sequence) tree representation. Essentially, each level (except the root) is stored in left-to-right order as a bit list; bit list T_i stores level i and bit list L stores the leaves as shown in Figure 3c. Some additional information (5 % of extra space) is stored to allow efficient tree traversal on the T_i bit list. Additionally, leaf nodes in bit list L can be compressed using dictionary compression, illustrated as L_c in Figure 3c.

The compression ratio of an K^2-tree achieved for a given graph depends on the choice of parameter k and the order of the graph vertices in the adjacency matrix. We use a K^2-tree variant that allows individual k values per level with the settings used in [6] ($k = 4$ for inner nodes and $k = 8$ for leaves). For the vertex order, we use a BFS order over the reachability graph, which is a heuristic that demonstrated to achieve good compression rates [6]. The dictionary of the MR-index is used to implement the order independent of the vertex order in the data graph.

RPQs processing with an MR-index: Evaluating an RPQ materialized in an MR-index is straight-forward. The uncompressed MR-index simply scans the adjacency list. The compressed MR-index enumerates all vertices from the dense domain provided by the dictionary and poses a successor query to the K^2-tree for each vertex. Both variants map each reachability result back to the data graph vertex domain before emitting it.

4 EXPERIMENTAL STUDY

Compressed MR-indexes based on K^2-trees seem to be an appealing choice for MR-indexing, since they require significantly less space than their uncompressed counterparts. Our goal is to provide a better understanding when compressed MR-indexes are beneficial for RPQ processing and when they are not.

Data: We ran the experiments on the LDBC[2] dataset at SF1 (~3 million vertices and ~17 million edges) after loading the dataset completely into the in-memory column store. However, all RPQs effectively operate only on the directed, edge-labeled multigraph, i.e., on the *source*, *target*, and *label* columns of the edge table.

[2]http://ldbcouncil.org/

Table 1: Number of generated RPQs

n	non-closure	closure	total
1	15	4	19
2	124	17	141
3	609	77	686
total	748	98	849

Queries: We exhaustively generated RPQs from the LDBC data schema. Precisely, we generated all RPQs of the non-closure form $(l_1, \ldots, l_n)$ and the closure form $(l_1, \ldots, l_n)^+$, where l_i is an edge label or an edge label in reverse direction and $n \in [1, 3]$. We included only RPQs that have a non-empty result set on the schema graph, i.e., RPQs that have a chance to produce a non-empty result on the data graph. The closure RPQs $(l_1, \ldots, l_n)^+$ additionally required that the non-closure RPQ $(l_1, \ldots, l_n\ l_1, \ldots, l_n)$ has a non-empty result set on the schema graph, i.e. that the closure RPQs have a chance to produce a larger result set on the data graph than their corresponding non-closure RPQs $(l_1, \ldots, l_n)$.

Table 1 lists the number of generated queries for each category. Figure 4 shows the number of queries and the average number of query results over buckets of selectivity. We define query selectivity as the fraction of vertex pairs that are connected by a path of a given query w.r.t. the number of all possible vertex pairs, i.e., $|V \times V|$. Each bucket represents a half order of magnitude in query selectivity. As can be seen, the majority of queries produces quite large result sets. The reachability graph produced, ranges from about two orders of magnitude smaller than the original data graph to about one order of magnitude larger. Hence, MR-indexes typically have to deal with result sets of this size range. Only a few queries produce result sets smaller or larger than that.

Measurements: For each of the generated RPQs we measured the space consumption of the uncompressed MR-index (ADJ) and the compressed K^2-tree-based MR-index without leaf compression (K2) and with leaf compression (K2C). We also measured the runtime of the query with the baseline RPQ processing (cf. Section 2), the uncompressed MR-index, and the compressed MR-index. Based on the individual measurements, we consider also the effect of the

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 11, ISBN 978-1-4503-5038-9
https://doi.org/10.1145/3078447.3078458

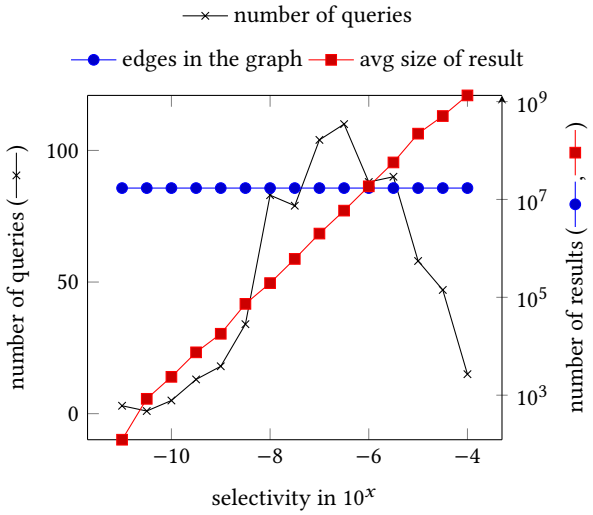


Figure 4: Selectivity distribution of queries.

different MR-indexes on workloads consisting of multiple queries. We discuss the findings in the following.

Environment: We conducted all measurements on a Haswell machine (Intel(R) Xeon(R) CPU E5-2660 v3) running at 2.6 GHz. The machine has 128 GB of RAM.

4.1 Space Consumption

Figure 5 shows the average space consumption of the three MR-index variants. For the uncompressed MR-index, the space consumption shows the lower bound of 64 bits per edge. The space advantage of the compressed MR-index becomes clearly visible. While for very selective RPQs that produce a small reachability graph, the space savings achieved by the compressed MR-index variants is small, it quickly grows with the size of the result set of the RPQ that is indexed. Figure 6 supports this observation. It shows the average space savings, i.e., the space savings due to compression relative to the size of the uncompressed MR-index, as well as the average compression rate for both compressed MR-index variants over buckets of selectivity. As can be seen the space savings effect of the compression increases up to over 90 % with and over 80 % without leaf compression. The achieved compression rate is on average around 10 bits per edge with and around 20 bits per edge without leaf compression for the majority of queries. For RPQs with very large results, the average compression even drops to around 10 bits per edge with and around 5 bits per edge without leaf compression. Note that the first two selectivity buckets (10^{-11} and $10^{-10.5}$) are not very representative as they comprise only 3 and 1 queries, respectively. We also found (not depicted in the figures) that the achieved compression rate can vary significantly, easily between from 1 to 3 bits per edge up to 30 bits per edge within one selectivity bucket. For no RPQ we found a compression rate worse than the lower bound of the uncompressed MR-index.

Summary: The compressed MR-index achieves significant space savings on large RPQ results sets. To that end, K^2-trees are a promising technique to make MR-indexing affordable also for less selective queries. However, the compression effect varies significantly, so the technique works not equally well for all queries.

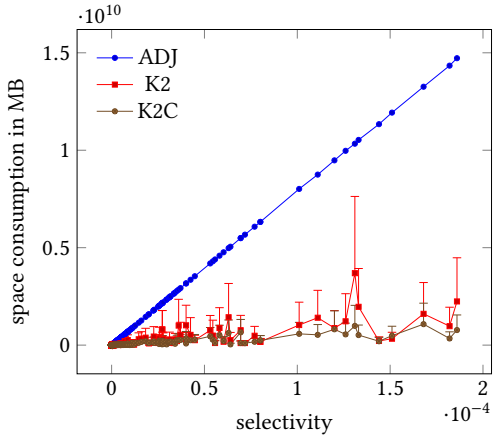


Figure 5: Space consumption of queries.

4.2 Single Query Performance

We summarize the query runtimes of individual RPQs in Figure 7, which depicts the average query runtime over buckets of selectivities. The compressed MR-index is shown in two variants, with and without leaf compression. As can be seen, all MR-index variants outperform the baseline except for queries of very low selectivity. For very low selectivities ($\sim 10^{-4}$) the compressed MR-index variants perform equally well or worse than the baseline. Another general observation is that across the board the uncompressed MR-index outperforms the compressed variants. The whiskers in the plot show the maximum query runtime per bucket. The runtime of the compressed MR-index heavily depends on the achieved compression, which varies from query to query. Reachability graphs resulting from RPQs may expose different topologies, such that the BFS-based vertex ordering can lead to varying compression ratios.

Summary: In the single-query experiment, the uncompressed MR-index significantly outperforms the compressed MR-index variants. The space savings achieved by compression are paid at query runtime. Nevertheless, the compressed MR-index provides query speedups over the unindexed baseline except for queries with very low selectivities, i.e., for large result sets.

4.3 Multiple Query Performance

To simulate the execution of multiple queries in a batch we sampled randomly queries from the generated RPQs. We chose three different batch sizes: 50, 100, and 300. The sampling was repeated a hundred times for each scenario to include different sets of queries. Additionally, we set a memory budget of 100 MB, 1 GB, and 10 GB, respectively, which can be spent on index structures for the queries included in the batch. We used a greedy algorithm to determine the index configuration for each workload. For the scenarios denoted as ADJ, K2 and K2C the algorithm decides for each query in the batch if the respective MR-index should be used or if the evaluation falls back to the baseline. For the scenario denoted as *best* the algorithm can freely choose among all MR-indexes. In all cases the greedy algorithm picks from all queries the index with the best ratio of query runtime benefit to space consumption that fits into the remaining memory budget. This process is repeated until no more indexes can be added to the index configuration.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 11, ISBN 978-1-4503-5038-9
https://doi.org/10.1145/3078447.3078458

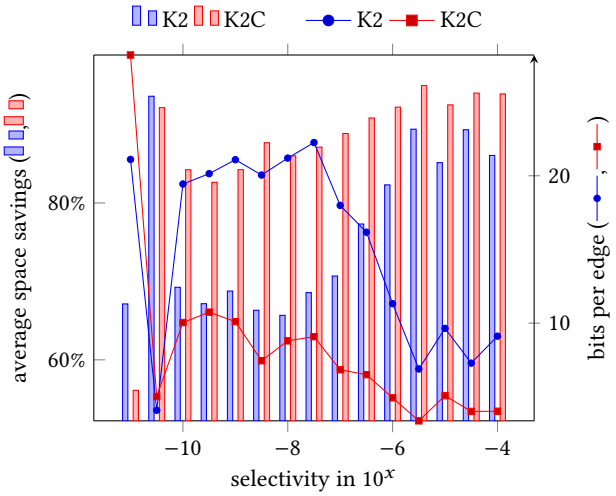


Figure 6: Space savings and bit per edge.

Figure 8 shows the results for each sampling scenario. The execution time of a batch is the sum of the execution times of all queries in the batch. As can be seen across all workload samples, the compression allows storing more MR-indexes in the memory budget and indexing more queries of the given workload, such that a better total workload runtime can be achieved.

The leaf compression for K^2-trees (K2C) achieves the lowest index size. Therefore, considerably more MR-indexes can be stored in the memory budget and used to improve the evaluation time. Nonetheless, the leaf compression comes with a considerable overhead for querying the index which results in a worse execution time compared to the K^2-tree without leaf compression (K2).

As seen in Section 4.2, the uncompressed MR-index has a very good lookup performance. When it comes to processing multiple queries the size of the index is a limiting factor: the index can only be stored for a few queries with large result sets, where the index lookup would give the most benefit. For queries with a small result set it is still beneficial to use the index but the gain in reduced total execution time of the batch is smaller.

Choosing among all index variants leads to the best overall execution time. Figure 9 shows which index variants are chosen by the algorithm. Most of the indexes used are K^2-trees with leaf compression as they have the best compression ratio and therefore make the best use of the limited space available for indexes. Other queries are better served by the uncompressed MR-index, e.g., queries with very small result sizes which also require a very fast lookup. The overhead introduced by the leaf compression leads to higher execution times, which is not proportional to the small reduction in size. Therefore, the compressed MR-index without leaf compression is used for queries where the result size cannot be compressed good enough with leaf compression to compensate the increased lookup performance.

Summary: It is not beneficial for every query to compress the MR-index as much as possible. The reduced space consumption is paid with an increased lookup time in the index during query time. A greedy algorithm, which can choose from all MR-index variants and pick the best representation for each query, results in the best overall execution time across all batch sizes and memory budgets.

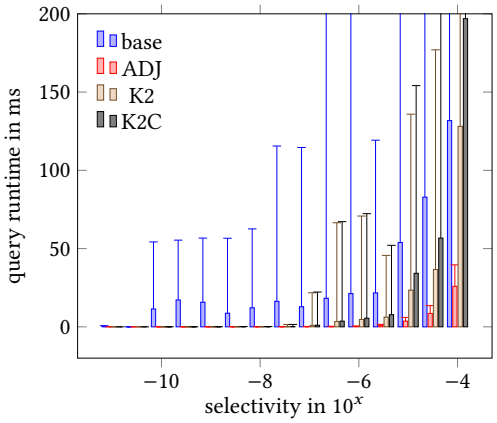


Figure 7: Single query execution time.

5 RELATED WORK

Various techniques have been proposed for speeding up RPQ evaluation. One elegant way is by exploiting selective labels in the middle of the path [12]. Here, the query processor looks for labels in the regular expression outside a Kleene star that are selective over all edges. If such labels exist, the query processing can use them to find vertex bindings within the path and use bidirectional BFS between these bindings. Obviously, the approach requires the presence of selective labels in the data and more importantly in the regular expression. Regular expressions have to be rather complex for this approach to have an impact.

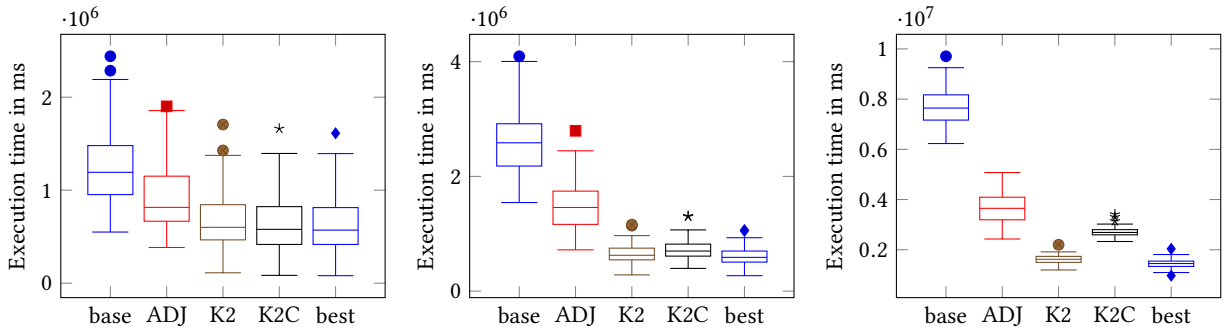
Yakovets et al. point out that existing approaches for RPQ processing consider only very limited possible query plans. They propose a query evaluation strategy that allows for a large range of different query plans, so-called waveplans [21, 22]. Waveplans can express conventional RPQ query plans, such as the left-deep join trees of our baseline or plans based on a dedicated transitive closure operator. Some of these waveplans cache common sub-paths to share intermediate results between different parts of an RPQ. Common sub-paths can also be shared between different RPQs [1] Such a common sub-path cache essentially creates an MR-index. Our study can help to make informed decisions about which data structure to use when implementing such a cache.

Gubichev et al. [10] discuss reachability indexing in the context of RDF-3X. Instead of indexing query results, the reachability is materialized for label-induced subgraphs only. They make use of a compact reachability index called FERRARI [16]. The FERRARI index is not well suited as an MR-index. First, like most reachability indexes, it only works on acyclic graphs (DAG). A cyclic graph has to be converted to a DAG first by graph condensation and the condensed graph has to be stored next to the index as FERRARI requires access to the condensed graph during lookup. Secondly, only simple queries about the reachability between a vertex pair are supported by the index, e.g., is vertex v reachable from vertex u? The efficient extraction of multiple reachable vertex pairs as required for an MR-index is not supported.

Another recent approach worth mentioning is the use of landmark indexing for label constrained reachability queries, which is a subset of RPQs [17]. The data graph is indexed in a single structure for a query instead of materializing individual query results.

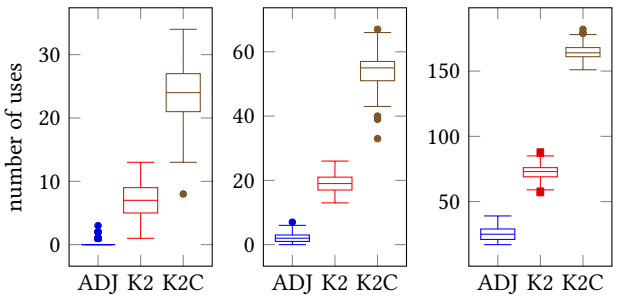
Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 11, ISBN 978-1-4503-5038-9
https://doi.org/10.1145/3078447.3078458



(a) 50 queries, 100 MB (b) 100 queries, 1 GB (c) 300 queries, 10 GB

Figure 8: Execution time of the batched queries for three different batch sizes.



(a) 50 queries, 100 MB (b) 100 queries, 1 GB (c) 300 queries, 10 GB

Figure 9: Number of indexes used in the *best* scenario.

6 CONCLUSION

RPQ is an important query type for graph data. A simple yet effective approach for speeding up RPQ evaluation is the materialization of the result of an RPQ into an MR-index. Such an MR-index can be used if the same query has to be answered again or for answering RPQs whose regular path expression contains the index path. Since RPQs can have results larger than the original graph, we investigated graph compression techniques for a space-efficient implementation of MR-indexes. In particular, we investigated the use of K^2-trees, which have shown to offer the best compression ratios [5]. Our key findings are:

- K^2-tree-based MR-indexes provide a significantly lower memory footprint over uncompressed MR-indexes based on adjacency lists.
- The query runtime on compressed MR-indexes is always worse than on uncompressed MR-indexes. When pure query performance matters most and sufficient memory is available, compressed MR-indexes are not advisable.
- For multiple queries and a limited memory budget, compressed MR-indexes offer a better trade-off between query runtime and memory footprint.
- Not all queries can be compressed well by K^2-tree based MR-indexes. In particular selective queries with a small result set benefit less from compressed MR-indexes. Hence, it is advisable to consider mixed settings of uncompressed and compressed MR-indexes for a given workload.

For the future, we plan to extend our study to different kinds of reachability indexes, more datasets, and the use of synthetic benchmarks, such as gMark [3].

REFERENCES

[1] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, and Mark H. Chignell. 2016. SwarmGuide: Towards Multiple-Query Optimization in Graph Databases. In *Alberto Mendelzon International Workshop on Foundations of Data Management*.
[2] Alberto Apostolico and Guido Drovandi. 2009. Graph Compression by BFS. *Algorithms* 2, 3 (2009), 1031–1044.
[3] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *TKDE* 29, 4 (2017), 856–869.
[4] Gérard Berry and Ravi Sethi. 1986. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science* 48, 3 (1986).
[5] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k^2-Trees for Compact Web Graph Representation. In *SPIRE*. 18–30.
[6] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2014. Compact representation of Web graphs with extended functionality . *Information Systems* 39 (2014), 152–174.
[7] Mar Cabra. 2016. How the ICIJ Used Neo4j to Unravel the Panama Papers. Neo4j Blog: https://neo4j.com/blog/icij-neo4j-unravel-panama-papers/. (May 2016).
[8] Francisco Claude and Gonzalo Navarro. 2010. Extended Compact Web Graph Representations. In *Algorithms and Applications*. 77–91.
[9] Szymon Grabowski and Wojciech Bieniecki. 2011. Merging Adjacency Lists for Efficient Web Graph Compression. In *ICMMI*. 385–392.
[10] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling Kleene: Fast Property Paths in RDF-3X. In *GRADES*. 14:1–14:7.
[11] Matthias Hauck, Marcus Paradies, Holger Fröning, Wolfgang Lehner, and Hannes Rauhe. 2015. Highspeed Graph Processing Exploiting Main-Memory Column Stores. In *Euro-Par*.
[12] André Koschmieder and Ulf Leser. 2012. *Regular Path Queries on Large Graphs*. 177–194.
[13] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995).
[14] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. 2015. GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems. In *SSDBM*. 29:1–29:12.
[15] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *BTW*.
[16] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*. 1009–1020.
[17] Lucien D.J. Valstar, George H.L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *SIGMOD*.
[18] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *GRADES*. 7:1–7:6.
[19] W3C. 2013. SPARQL 1.1 Overview. http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/. (March 2013).
[20] Peter T. Wood. 2012. Query Languages for Graph Databases. *SIGMOD Rec.* 41, 1 (April 2012), 50–60.
[21] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. WAVEGUIDE: Evaluating SPARQL Property Path Queries. In *EDBT*. 525–528.
[22] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query Planning for Evaluating SPARQL Property Paths. In *SIGMOD*. 1875–1889.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden