Regular path query evaluation using path indexes

Peters, J.

Award date:
2015

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Regular path query evaluation using path indexes

MSc Thesis

Author:
J. Peters

Supervisors:
dr. G.H.L. Fletcher
Prof. dr. A. Poulovassilis

Assessment committee:
dr. G.H.L. Fletcher
Prof. dr. P.M.E. De Bra
dr. O. Türetken

August 31, 2015

Abstract

Graph structured data has lately become a bigger part of our lives. Not only in social network structures, but in many other areas we also see the rise of graph structured data. With the upcoming of this type of data, come different techniques and problems in order to use and analyse the data.

One of those problems is the regular path query (RPQ). Here, regular expressions over edge labels are used in order to define a path, for which the begin- and endnode need to be found. Though various different approaches exist for RPQ evaluation, path indexes have never been used before. Path indexes store all the paths (up to length k). With path indexes, paths are pre-computed, increasing RPQ evaluation speed.

In this thesis, we show what specific sub-problems need to be dealt with in order to use path indexes for RPQ evaluation. Four different methods, two that make use of a histogram, are introduced. Furthermore, ways to reduce the storage size of the index have been explored, since the storage size is a bottleneck of the path index approach. One of the developed approaches, together with a recursive PostgreSQL implementation, is treated as the baseline. We show that, using the developed methods, we can increase the performance of RPQ evaluation compared to the baseline.

Contents

1. Introduction

A graph is a datastructure, consisting of nodes and edges. The nodes are objects and edges are relations between nodes. Graphs have various applications in fields as the semantic web, social network analysis, scientific workflow provenance and many others. Lately, the use of graphs has rapidly grown. We will study one of the problems for such graphs, namely the querying of graph databases, and in particular the regular path queries (RPQ) [8]. As an undirected graph can easily be changed to a directed graph by changing the undirected edge between two nodes to two directed edges, we will focus on directed graphs in this paper.

Regular path queries are navigational queries that check the existence of a path between two nodes. The path between these nodes must satisfy a certain regular expression condition over the edge or node labels. Therefore, the RPQ can be expressed as (xRy), where R is a regular expression over the labels and x, y are variables. The result of a RPQ is the set of paths for which the concatenation of labels conforms to the regular expression R. The main novelty of RPQs lies in the fact that for regular expressions we can use the Kleene star operator, which denotes the fact that a label may occur zero or more times. In order to solve a query with this operator, we need to support recursion.

One of the problems that RPQs bring is the fact that they cannot be easily expressed by the relational query languages, such as SQL, because limited recursion is allowed. Despite of that, several navigational languages for graph databases can be embedded into the relational language Datalog [4]. Datalog is also used in [9, 25], where Kleene star is translated into recursive Datalog programs or recursive SQL views, as we will see in Chapter 2.

A possible way to efficiently evaluate regular path queries might be a "path index". A path index is a data structure on a given graph G which, provided a list of edge labels $l_1, ..., l_n$, returns all pairs of nodes (s, t) such that $l_1, ..., l_n$ are the labels along some path from s to t in G. Since we can express regular expressions as lists of edge labels, path indexes might be able to efficiently evaluate RPQs.

Various papers have studied RPQs and some extensions to RPQs (e.g. [5, 6, 16]). Though, few practical solutions have been provided and, to our knowledge, path indexes have not been used before for regular path queries. In this thesis, the **main goal** lies in answering the following question: "How to use Path Indexes in order to provide an efficient way of evaluating Regular Path Queries?"

The **main contributions** of this thesis are that we show how path indexes can be used for RPQ evaluation. We show different ways to use a path index, with or without usage of

an additional histogram. The path index achieves better performance compared to the case where we only query the edges. Furthermore, we show that using the histogram is in most cases beneficial.

In Chapter 2 we will discuss the current approaches for regular path query evaluation. Furthermore, we will further elaborate on path indexes. After that, the research question will be presented in Chapter 3. There, we will also provide sub-questions and we will divide the problem into sub-problems. The different sub-problems will be discussed in Chapter 4 and Chapter 5. Next, in Chapter 6 the experiment setup will be discussed, for which we show and discuss the results in Chapter 7. Finally, we will conclude in Chapter 8.

2. Literature Review

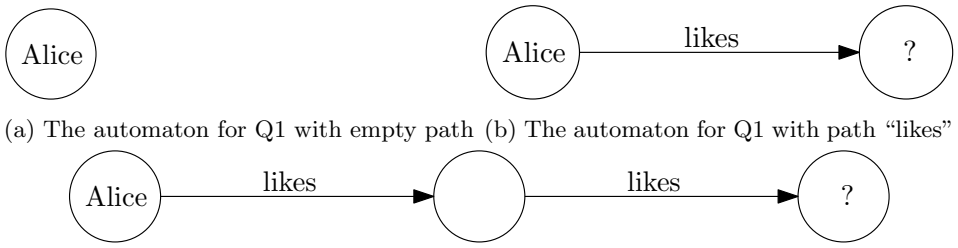
In the literature, there are several different approaches for regular path query evaluation. These are: automata-based processing, search-based processing, Datalog-based processing and reachability-index-based processing. Automata based and datalog based evaluation are briefly described in [25, p. 5-6].

2.1 Automata-based processing

For the automata-based approaches, the regular expression is converted to an automaton as is the graph. Next, by mapping the states of the query automaton on the graph, we get the intersection of the graph and the query automata. This gives an automaton, which is the subgraph specified by the query, hence giving us the result of the RPQ. In [13], an automata based algorithm that divides the search space into smaller pieces by looking at the rare labels is provided. The rare labels are uncommon labels in the graph. By partitioning the graph on rare labels that should be in the result, basically using these rare labels as begin-, end- and waypoints, they decrease the search space and hereby get a more efficient approach to the RPQ evaluation. Downside here is that this algorithm is dependent on the choice and presence of rare labels. In case of a poor choice of the "rare" label or relatively long queries, the complexity can reach $O(n^2)$. Another automata based solution is presented in [23]. This solution mostly focusses on distributed graphs. The overall downside of the automata based approach is that the states are mapped onto the graph, which may cause the run time complexity to go to $O(|V| * |S|)$, where $|V|$ denotes the number of nodes in the graph and $|S|$ the number of states in the automaton, basically showing that we try to map every state of the automaton on every node. Furthermore, multiple automata may need to be mapped to evaluate a single query.

In [21] another automata based approach is presented. Here, the "APPROX" and "RELAX" methods are used to alter the original query within certain cost. Furthermore, the result set may be limited to only give the top results, starting with the results with lowest cost. Various optimizations, such as using multiple automata and not expanding the search, for example when this has a cost but we already have reached the result limit with results with lower costs, have been discussed and succesfully applied.

Waveguides, as presented in [26], also use automata as underlying technique. The regular expression is translated into different wavefronts, that all solve a part of the query. Here, several techniques are introduced to reduce the amount of mappings needed. One of these techniques is the reuse of data for loops in the automata, called loop caching.



(a) The automaton for Q1 with empty path (b) The automaton for Q1 with path "likes"

(c) The automaton for Q1 with path "likes likes"

Figure 2.2: The three different automata necessary for answering Q1

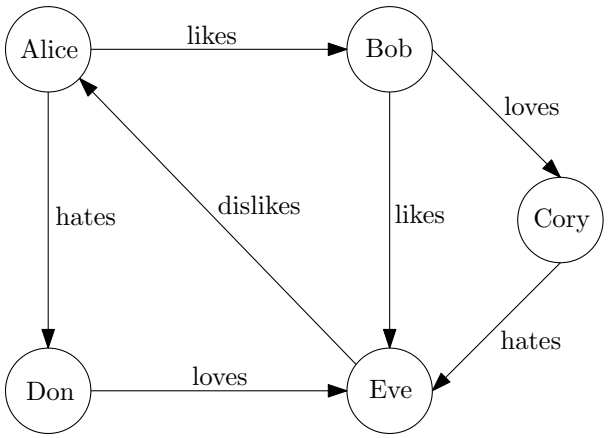


Figure 2.1: An example of a simple directed graph

As an example, we present a simple directed, labeled graph in Figure 2.1. Take example query Q1: "Select all the people that are on a path containing only edges labelled 'likes', starting from Alice". If we express this as a regular path query over the labels of the edges, we get: likes*. In order to use this on "Alice", we simply can use "Alice" as the start node to evaluate the expression on. For the automata based approach, we would make an automaton for this query. Notice that, since this expression basically denotes that we have a path that contains "likes" 0 or multiple times, we have to make multiple automata and for every of these automata, we need to try to map them. See Figure 2.2 for some of the automata for Q1. Note that we want to get all persons that could be placed at the question marks. Now by trying to map the automata on the graph, we find that "Alice", "Bob" and "Eve" satisfy Q1.

2.2 Search-based processing

Next we have search-based approaches. Here, examples are pattern matching and using a distance matrix. Several ways of evaluating regular path expressions are described and evaluated in [10]. Two classes of queries are defined. For the simpler class (RQ), distance matrices or bi-directional search provide a solution. The harder class (PQ), of which RQ is a subset, uses the solutions for RQ in order to solve its queries.

The distance matrix here is a 3-dimensional matrix over begin nodes, end nodes and edge

labels. This means that, in order to solve a query over multiple edge lables, we have to do multiple lookups. Also, the storage will be (at least) quadratic in the number of nodes.

With the bi-directional search, a breadth first search is done started from both potential begin- and end-nodes. The path denoted will be followed from these nodes until an intersection is found or until the paths cannot be expanded any further. For multiple edge labels, the complexity is worse than cubic in the number of nodes, namely $(O(|V|^2(|V| + |E|)))$, where $|V|$ denotes the total number of nodes and $|E|$ denotes the number of edges.

For PQ, two algorithms that make use of the distance matrix or the bi-directional search are explained. These algorithms basically divide the problem in solvable pieces by the distance matrix or the bi-directional search. As these algorithms both try to match nodes, the worst time complexity is cubic in the node size.

Though these algorithms are reported to scale well, they often have lengthy pre-processing steps (in quadratic complexity) and run time complexities that are at least quadratic in the number of the nodes. Looking at Figure 2.1 and Q1, assuming pattern matching on the edges, an algorithm may first find all the edges / combinations of edges that satisfy the given relation. Next, for the found matches, we can find the edges originating in "Alice" to answer the query. Note that finding the first set of edges may be a very costly process considering the fact that edges may occur many times. Also note that, since we use the Kleene star, we would have to verify a lot of patterns. For the case of a distance matrix, we know that we have a path starting at "Alice". Though, we do not know the exact lengths of the paths. Thus, evaluating this query using a distance matrix would be very inefficient, since the distance matrix here does not contain enough information to efficiently evaluate the query.

2.3 Datalog-based processing

Another approach is Datalog-based processing, where Datalog is extended such that regular expressions can be evaluated. This is done in [9], where some extensions in order to take provenance into account are also added.

In order to express Q1 about Figure 2.1 in Datalog, we assume that graph edges are interpreted as facts. One of these facts would be: likes(Alice, Bob). This fact denotes that Alice likes Bob. The rest of the expression can now be stated as:

likes*(X,Y): likes(X,Y)
likes*(X,Y): likes(Y, Z), likes*(X,Z)
? - likes*(Alice, Y)

Here, the last line denotes the actual query call. Note that we first express the likes* relationship in terms of another relation, namely likes. Then, we state that we can add an Y to the solutions of the query if there exists a Z such that Y and Z like each other, and Z has this likes* relation with X. By replacing the X with "Alice", we evaluate this relation for Alice. As you can see, this would work for paths of length 1 or more. Though, the node "Alice" can trivially be found.

Node	Reachable nodes level 1	Reachable nodes level 2
Alice	Bob	Eve

Table 2.1: "Likes" reachability table for the graph of Figure 2.1

2.4 Reachability-based processing

The last approach is based on reachabilities. A reachability table is constructed for each edge label, such as shown in Table 2.1 for the edge label "likes". This table gives us candidates, for which we can then check if the paths match the given regular expression. Reachability tables for the purpose of RPQ evaluation have been used in [12]. Some promising results are reported compared to the RDF store Virtuoso, for which the new approach performs 3-5 times faster which is using a reachability structure to speed up the actual querying. Though, we need to store a reachability structure for each predicate label and some results are not exact, for which we need to perform additional depth first search steps. Further more, a reachability index structure does not take the regular expression into account, this has to be processed later on. Another downside is that, as we have multiple labels in our query, we have to join at least once for every label occuring in the query.

For the example of Figure 2.1 and Q1, we would make such a reachability structure. This structure might look like a table, as shown in Table 2.1. As you can see, we have denoted 2 levels in the reachability graph for the "likes" label. Level 1 denotes the reachable nodes with a path of length 1. Level 2 denotes the reachable nodes with a path of length 2. Looking at the table, we see that the "likes" label has been taken into account, hence we can evaluate this query. For queries containing more labels, we need more tables, and joins on those tables, though.

2.5 Path Indexes

The approaches to date for RPQ evaluation all seem to have their downsides. In this thesis, we investigate the use of path indexes to evaluate regular path queries. In [27], various kinds of index structures for graph databases have been studied. For example, feature-based indexes, where paths, nodes/edge labels, trees or subgraphs can be used in order to index a graph. All the structures can be indexed, or only frequent or discriminative structures. This type of indexing can be used for e.g. substructure search, where a certain structure in the graph is used, or structure similarity search, where the goal is to find all subgraphs that approximately contain the query graph.

Next, in [17] the focus lies on different indexing structures for RDF graphs. Here, three basic perspectives, namely relational, entity and graph-based perspective, are introduced and indexes for these perspectives are discussed. The relational perspective looks at storing RDF graphs in a relational database, while the entity perspective represents the data as objects and attributes. The graph-based perspective represents the RDF graph as a graph. As the entity perspective typically returns a subset of entities for a given query, we will not look any further into this.

Now, for the relational perspective, multiple representations are possible. For example the vertical representation, where the relational schema contains stores the RDF triples as (*subject, predicate, object*). This provides various options for indexing, such as both clustered and unclusterd BTree indexing. Another representation is the horizontal representation, where we have a column per predicate and every row starts with a subject. Now, a certain row contains one subject and all the objects that apply to this subject, stored in the columns of the corresponding predicate. To minimize storage overhead, this table might be divided in several sub-tables to minimize empty rows.
With the graph-based perspective, both general graph-based indexing methods and structural indexes are used. Index methods may include suffix arrays, distance-based indexing, dataguides and the usage of bisimilarity.

A path index seems to be a logical choice, since we are dealing with regular expressions that match paths in the graph data. We can use the index to find the begin- and end-nodes of the path corresponding to the regular expression. Furthermore, path indexes have been studied in the context of many kinds (e.g. tree, relational) of databases (e.g. [11, 19, 24]).

In [24], dataguides are explained. Here, the graph is reduced in size by mapping the same sort of nodes on one node in the dataguide. This sort of index is usefull for checking the existence of paths.
Next, T-indexes are introduced. With a T-index, the main idea is that rather than indexing all the paths, it is more convenient to index those having high percentage accessed queries. This is achieved using the bisimulation relation.
Index fabric is also shown, which uses a Patricia structure. Index fabric is a layered method, like B-trees, based on the Patricia trie. The structure enables for storing a large number of keys in few layers and for needing only 1 index I/O per layer.
Last, various (k)-indexes are shown. A(k)-indexes are based on k-bisimilarity, grouping nodes based on the local structure, like the length of incoming paths, up to k. The D(k)-index is basically a generalization of the A(k)-index and the 1-index (a T-index). The difference with the A(k)-index is that different types of data nodes can be assigned different bisimilarity requirements. The M(k)-index extents the idea of different bisimilarity even further, here it is possible to define different k values for different index nodes with the same tagname.

Looking at the construction time complexity and storage complexity, it is shown that DataGuide may require exponential time, while basic Index Fabric construction takes only $O(m)$ where m is the number of edges. For T-index it depends on the actual path template selected, but a 1-index can be constructed in $O(m \log n)$ with m the number of edges and n the number of nodes. The (k)-indexes take $O(km)$, $O(l + m)$ for A(k)-index and D(k)-index respectively, where l is the amount of edges in the A(0)-index the D(k)-index is constructed from. The M(k)-index depends on the actual data.
For the storage, DataGuide requires $O(m)$, Index Fabric $O(pn)$ where p is related to the structure of the source. 1-index needs $O(m)$ size, just as the (k)-indexes.

In [11], the approaches for indexing are divided in three parts: The structural indexes on XML data, the relational approach for structuring data and the native (XML) approach. In the first part, most techniques that are shown are also discussed in [24], such as DataGuide,

A(k)-indexes and D(k)-indexes. It also shows color schemes, such as Dewey coding, where each node is associated with a vector that consists the path from the root to the node, and PrePost coding, where the start and end tags are given a number (which represents the place of occurence) and these numbers are added to the node. In the relational part, the data is stored based on the edges, either by denoting the edges with their source and target, or by having a seperate table per edge in order to save space and improve query performance. Another approach is to store the node, where the start and end level of the XML data are also stored. Downside of both approaches is that they require additional joins for certain query types. Another way is to store the paths, which can be done in various ways. The upside of this approach is that less joins will be needed. The last relational approach is the DTD approach. Here, information of the underlying structure is used in order to get more compact storage and more efficient querying.
As for the native approaches, it starts with the join approach. This basically looks like the node approach, only now the nodes are stored in an inverted list and pregroup the table by the labels. This does imply that a relational storage can be used for this approach. The sequence approach encodes the data and query as a sequence, so that subsequence matching can be used in order to evaluate the query. The navigational approach searches by traversing the data tree along tree edges. This method is fit for streaming XML data or for filtering.

In the literature, we have seen many ways to index semi-structured and graph-based data. We have seen many different approaches. Interesting to see is that, even though they deal with graph data, many of the approaches still use a relational approach in order to store the data or an index on the data. As discussed before, we would like to use a path index, which also uses a relational database. More precisely, we will investigate the use of a k-path index, for which we store the paths up to length k.

As an example, we will consider a path index for the graph shown in Figure 2.1. Such a structure might look like in Table 2.2. As you can see, the paths are listed and the nodes between these paths occur are denoted. Note that it is also possible to add longer paths here over multiple edges, such as shown for the path "Likes Likes" (we have left out the other longer paths for simplicity). Furthermore, the example denotes an undirected graph. For directed graphs, we would need a "From node" and a "To node" column instead of the "Between nodes" column here.. In order to verify Q1 now, we can first look at the rows with "Likes" as path in combination with the node "Alice" to get "Bob". Next, we can look at the "Likes Likes" rows to find "Alice" in the "Between nodes" column and see that "Eve" is the answer to our query. Trivially, "Alice" is also a valid evaluation of the query.

The major benefit of this solution is that we immediately evaluate the query using paths. Therefore, we can simply output the answers we find in the structure. Also, the lookup should not be very expensive, thus this method might be quite efficient. On the other hand, we need an index structure for this that may get very big, meaning that we need to take storage and pre-processing times into account.

Note that there are various sorts of path indexes. We will focus on k-path indexes, which stores paths with size at most k. This way, any query with length as most k can easily be evaluated with a single look-up in the index. For larger queries, we have to split paths into smaller ones and use multiple look-ups in the index.

Path	**Between nodes**
Likes	Alice, Bob
Likes	Bob, Eve
Loves	Bob, Cory
Loves	Don, Eve
Dislikes	Eve, Alice
Hates	Alice, Don
Hates	Cory, Eve
Likes Likes	Alice, Eve

Table 2.2: Path index for the graph of Figure 2.1

3. Research Question and Approach

Looking at the possibilities that path indexes offer and, in addition to that, the fact that path indexes have never been used before to evaluate RPQs, we identify the following research question:

"How to use Path Indexes in order to provide an efficient way of evaluating Regular Path Queries?"

The "efficient" part of the question points at the fact that there already are some approaches for RPQ evaluation. The goal of using path indexes for this purpose is then to improve some properties of the already known approaches, e.g. running time or storage. Furthermore, the focus lies on the Kleene star operator, since that is the main novelty of RPQs. As mentioned before, we will tackle this problem using the k-path indexes, stored in a relational database. We will do this with k-paths, since storing all the paths will be bad for the storage and the paths may contain cycles, giving a possibly infinite amount of paths if we would store all the possible paths.

3.1 Sub-questions

In order to come to an answer for this research question, we have to find answers for the following sub-questions as well:

- How to translate RPQs to paths?
- How to evaluate an arbitrary long path using the k-path index?
- How can we use a histogram of statistical information to optimize query plans for RPQ evaluation?
- Is there a relation between storage size of the index and performance?

Here, the first question is about writing out the paths a RPQ denotes. Since we use a path index, this will come in handy while working towards the evaluation of the query. The next question directly follows up on the first question. When the first two questions have been answered, we should be able to evaluate any RPQ using the path index. The following question then aims at adding some efficiency. Can we use a histogram to improve the performance, and how should we use this histogram? Last of all, we should also keep the storage into account. Since we might get a lot of paths, we should make sure that we store the paths in an efficient way and maybe even leave out some paths.

In the end, the sub-questions provided will help us working towards solving the problem, while also looking at efficiency and storage.

3.2 Dividing the problem in sub-problems

In order to address these problems on how to evaluate regular path queries by means of a path index, we have to design several features. Therefore, we will divide the problem into sub-problems. The first problem is initializing the path index. We want to be able to define the value of k when creating a path index, such that we can pick the k value that we think is best. We will describe the exact initialization of the path index in Chapter 4. We will also shortly address sub-question four here.

Next, we will describe the usage of the path index in Chapter 5, where we will describe various parts in order to evaluate RPQs. We want to parse the regular expression into the possible paths, as described in the first sub-question. Every regular expression over the edge labels evaluates to one or multiple paths. In order then to solve the total RPQ, we need to evaluate every path we derive and combine the results. This is described in Section 5.1.

In addition to that, we need to find a way to use the k-path index in order to find the begin- and endnodes for the paths. Since the paths can be larger than k, we need to divide them into index resolvable pieces and join these pieces together to get an answer. We want to define the join order ourselves, for which we will use histograms. In Section 5.2 we define how the histogram is build. In Section 5.3 we show how we break down the paths into index resolvable pieces, in order to evaluate the RPQ. These Sections provide a part of the answer to sub-questions two and three.

Last of all, we need to find a way to go from the pieces of a path to an actual result. We want to get the begin- and endnodes for a certain path from the path index, meaning that we should find some way to use the order of the histogram and the way the path is broken down to achieve a query capable of evaluating the RPQ. This, together with the dividing of the paths and the usage of the histograms provides a solution to sub-questions two and three. We discuss this in Section 5.4.

4. Path index

In this section we will discuss how the initial edges are put in a table, how we expand the set of edges with path length 1 to a set where the maximal path length is k and how we get an index out of this. In the end, we store the path and the source- and target-nodes for all paths of length at most k.

The path index will be a table with the columns *path*, *beginnode*, *endnode* and *length* for every row. The path will be a string, such that we can denote the path in the graph in a readable format. The begin- and endnode columns, as well as the length column, will contain integers.

4.1 Initial edges

In order to build the path index, we first load the initial edges in a relational database (Postgres). We also add the reversed edges, such that for every normal edge with label l, begin node a and end node b, we have a reversed edge with label l, begin node b and end node a. We add an identifier to the label, such that we know that this label depicts a reverse edge.

The input is delivered by a text file. For the current implementation, every line depicts an edge. The first part of the line is the identifier of the source node. Then a space comes in, after which we can find the label for the target node. Last, after another space, the label of the edge is denoted. The node labels are integers and the edge label is treated as a string, but this might also be an integer.
We process every line of that file, and thus every edge, by writing out two lines to another file. One line is for the original edge, the other line is for the reversed edge. The line will contain the path, source node and target node. When we have processed all the lines and created a correct file, we can load this file in using the "copy_from" command of the cursor. Note that the initial edges and the reverse edges form the paths of length 1. By writing it to a file, we make sure that we do not have to keep feeding insert instructions to Postgres, as we do this with one command now.

4.2 Additional levels

After we have loaded in all the edges, or paths with length one, we might need to add additional paths with larger length to get to k. All paths with the same length will be refered to as a certain level of the path index. For example, if we load in all paths with length two, we

Path	Beginnode	Endnode
a	1	2
b	2	3
c	3	1

Table 4.1: $k = 1$ pathindex

Path	Beginnode	Endnode
a_b	1	3
b_c	2	1
c_a	3	2

Table 4.2: Level $k = 2$ of a pathindex

will refer to loading in level two of the path index.

For loading in more paths, we start working with the edges of length 1. Note that, due to the reverse edges, we have to take care of a lot of possibilities. For example, when we want to have all the paths of length 2, we get the following options (we denote a reverse edge here by l^{-1}, where l is an edge label):

- $l \circ l$
- $l \circ l^{-1}$
- $l^{-1} \circ l$
- $l^{-1} \circ l^{-1}$

Though, as we have also added these reverse edges to the edges with length one, we can make an SQL script to get all these paths with correct begin and end nodes easily. This SQL script looks as follows:

"COPY (SELECT (e.path || '_' || e1.path) as path, e.beginnode, e1.endnode, 2 as length FROM edges as e, edges as e1 WHERE e.endnode = e1.beginnode) TO 'D:/level2' CSV"

Here, "path" denotes the path. Note that we divide two different labels by a "_". Next, we set the length to two. Furthermore, we couple the begin and end node of the two different paths of length one in order to get a valid path of length 2. Note that this query already takes all the options into account, as we have also added the reverse edges to the table with paths of length one (edges). By adding all the paths derived by the SQL code, we can extend the path index with $k = 1$ to a path index of $k = 2$. We do this by storing all the paths in a file. After that, we load the file in to the table. In order to keep the selection times low, we choose to store the $k = 1$ paths in the index and in a different table. The different table is used to generate additional levels. In Table 4.1 we show a $k = 1$ pathindex, without the inverses for simplicity, for which we have shown the $k = 2$ level in Table 4.2. As shown, level two of the path index can easily be derived from the initial edges.

For $k = 3$, we have to do another extension. This extension looks much like the extension from $k = 1$ to $k = 2$. We only now have to add 3 paths of length 1 together. We have a similar setup as for deriving the paths of length 2. For $k = 3$, the query looks the following way:

"COPY (SELECT (e.path || '_' || e1.path || '_' || e2.path) as path, e.beginnode, e2.endnode, 3 as length FROM edges as e, edges as e1, edges as e2 WHERE e.endnode = e1.beginnode

AND e1.endnode = e2.beginnode) TO 'D:/level3' CSV"

We can easily generalize the formula shown above to get the formula for $k = s$. First of all, we have s instances of *path*, giving us p_i for $1 \leq i \leq s$. Then, the path becomes: $\sum_{i=1}^{s} p_i.path$. This holds if we define the summation of strings as a concatenation, such that element 1 is always in front and element s is the last element. Note that, in our case, we add "_" between every two labels. Last of all, we need the next constraint to finish the query:
$\forall_{i=1}^{s-1} p_i.endnode = p_{i+1}.beginnode$.

This gives the following total query:

"COPY (SELECT ($\sum_{i=1}^{s} e_i.path$) *as path, e1.beginnode, es.endnode, s as length FROM edges as e1, ..., edges as es WHERE* $\forall_{i=1}^{s-1} e_i.endnode = e_{i+1}.beginnode$*) TO 'D:/levels' CSV"*
s here denotes the level of the path index that you want the generate and hence is a number.

Using the defined expressions, we can easily generate extra levels of the path index. Note that, as the k gets higher, levels will start to take more time to be generated, since the run time complexity is $O(n^s)$ where n is the number of edges and s the level that is generated.

Note that, for example for creating the level $k = 3$, we can also do this using paths of length 1 combined with paths of length 2. Though, this way we need to query the path index. As the path index may get very large, querying for all the paths of length 2 might take longer than two times looking for edges of length 1 and combining those in a different table (where only the edges of the graph are stored, so paths of length 1). Making different tables to store paths of different lengths in addition to storing it in the path index may provide a solution, though this requires more storage. We might just store all the levels in different tables, such that we have to combine different tables when doing a query, instead of self-joining.

4.3 Creating the index

In order to speed up the lookup of paths, we want to use a B-Tree index to store them. The final index can be created by Postgres using the following query:

"CREATE INDEX pathindex ON paths (path, beginnode, endnode)"

Here "*pathindex*" is the name of the index, "*paths*" the name of the table and "$path, beginnode, endnode$" the name of the columns we want to create the index on. This returns a B-tree as index, that will be used when the query planner thinks we can use it. This also means that we do not explicitly have to call the index, the query planner will take care of that.

We choose *paths*, *beginnode* and *endnode* as columns in that specific order, since we will be querying paths. Therefore, this should be the first item in the path index. Furthermore, we want both the beginnodes and the endnodes belonging to paths to be stored in the B-Tree, so that we can do index only lookups.

4.4 Reducing storage

As storing strings for a path may get very costly in terms of storage, we can translate the several paths to integers. Of course, we would like to do this in such a way that the lexicographical properties remain intact. This also makes the usage of the histogram easier, as we will show in Section 5.2.

In order to accomplish this, we will make a map, mapping the paths as string to integers. We map entire paths and not only the edge labels. We want to initialize the map already when we only have got the initial edges. This way, we do not have to translate the paths after obtaining them, but we do this beforehand. At the moment of initialization, if we know the k value we want to go to, we can determine the different paths we will need to map. This way, we can also determine the mapping needed. Note that, when initializing levels of the path index, we have to map the generated paths to the mapped paths. In order to achieve this, we store the map and query this in order to get the correct integer corresponding with the generated path. The part generating the path in the query now becomes:

"(SELECT m.int FROM map as m WHERE m.path = $\sum_{i=1}^{s} e_i$*.path) as path"*
instead of
"($\sum_{i=1}^{s} e_i$*.path) as path"*

Furthermore, we can get rid of the length by having an initial table with all the paths of length 1. We only get further levels by using this table. This also comes in handy since, if we would get additional levels from the index, we would have to translate the integer paths to string paths, concatenate those and then return to int again. Now, we only have to concatenate length 1 paths and turn the total path to an int by the mapping.

4.4.1 "Half" path indexes

Another way to reduce the storage is to store fewer paths in the index. Note that, due to the construction of the path index, for every edge we store both the edge and its inverse. We can easily leave the inverse out, such that whenever we need the inverse, we search for the edge and turn the source / target around. For example, if query p gives (source, target), then query p^- gives (target, source).

We can extend this idea to not only work for single edges, but for paths. By construction, every path has an inverse in the path index. It also holds that, for every occurence of this path with (source, target) pair that conforms to this path, there exists exactly one inverse with (target, source) pair. This holds for every path. Note that this means that we should be able to get rid of almost half the paths in the pathindex and still be able to evaluate every query, as some paths' inverse is the path itself (e.g. "$a \circ a^-$"). We will call this index the "half" path index, since it contains almost half the amount of paths.
In order to make query evaluation simple, we store, for a given path p and its inverse p^-, the path that is lexicographically the smallest. This means that, when checking if we should put a path in the index or if a path is in the index, we can check if the path is lexicographically smaller than its inverse. If this is the case, we can use it. Otherwise, we use the inverse. Experimental results for "half" path indexes can be found in Section 7.1.3 and Appendix D.

5. Using the path index for RPQ evaluation

In this Section, we will describe all the different elements we need in order to use the path index for RPQ evaluation. These elements have been introduced in Section 3.2.

5.1 Parser

One of the first steps towards RPQ evaluation is the parsing of the regular expression. The parsing is necessary in order to get the full paths, such that we can then divide these paths into solvable pieces. In this section, we will explain how we parse the regular expression to get full paths.

5.1.1 Regular expression language

First, we design a regular expression language. We want all the "basic" regular expression operators to be there, so we get the following operators (with R_1, R_2 either regular expressions themselves or labels):

- $R_1 \circ R_2$: R_1 will be followed by R_2. We denote the $\circ$ operator by having a separator between two adjacent expressions. If R_1 contains the pair (m, n) and R_2 contains the pair (n, o), then $R_1 \circ R_2$ contains the pair (m, o).
- $(R_1)^-$: $(R_1)^-$ is the reverse of R_1. This means that if R_1 contains the pair (m, n), then $(R_1)^-$ contains the pair (n, m).
- $R_1|R_2$: We either have R_1 or R_2, so we get $R_1 \cup R_2$. If R_1 contains the pair (m, n) and R_2 contains the pair (r, s) then $R_1|R_2$ contains the pairs $\{(m, n), (r, s)\}$.
- $(R_1)^{x,y}$: $\bigcup_{x \leq z \leq y} R_1^z$, where R_1^z denotes $\underbrace{R_1 \circ \cdots \circ R_1}_{z \text{ times}}$ and R_1^0 denotes ϵ

5.1.2 Parsing the language to paths

We handle the regular expressions by first handling the different labels and then applying the operators on the labels. This is done the following way: We go over the regular expression. When we encounter a "(", we look for the corresponding ")" and recur on what is in between. After a set of "(,)", "^{x,y}" can occur or a "$^-$" can occur. If this is the case, we handle these. If we encounter a "|", we add the part so far to the output and process the remaining. We can do this, because we know it should be the one (before the "|") or the other (after "|"), thus both parts (before and after) give a solution to the expression.

For example, for the regular expression "$a(b|c)^{1,2}$", where a, b, c are labels, we would process "a" and then recur on "$(b|c)$". Then, we would evaluate b and c and combine these to get $(b|c)$. Next, we can use this result to express $(b|c)^{1,2}$, which gives us the paths $\{b, c, bb, bc, cb, cc\}$. Next, we add a to the found paths in order to get all the paths. We get the paths $\{ab, ac, abb, abc, acb, acc\}$.

5.2 Histogram

In order to speed up the queries, we will make a histogram which we will use to determine the query plan. Using the histogram, we can estimate the size of the result set for a certain query and use this to pick the queries with least support in the dataset and join on these queries first. By doing this, we hope to keep the join times low, hereby speeding up the query. Of course, when the total query is smaller than k, we do not need to use the histogram, since no joins will be necessary.

5.2.1 Obtaining the counts

In order to get the correct histogram, we first of all check for every path how many times this occurs in the table. In order to get this, we can use the "COUNT" function, which will count the amount of occurences of a path. We use the following query:

"COPY(SELECT path, count(path) FROM paths GROUP BY path ORDER BY path) TO 'D:/SomeFile' CSV"

Here, we get the count of every path and store it to "D:/SomeFile". Next, we use the results obtained from this query to construct the histogram. Note that, since we have ordered the paths, we can query using the lexicographical properties.

5.2.2 Different histograms

Two different types of histograms are possible, namely the equi-width and the equi-depth versions. These versions either take the "occurences" of a path into account, which denote how many times the path is present in the table, or the amount of paths. Both versions work with a budget, which either determines how many paths or how many occurences are in a certain bucket, hereby indirectly determining the number of buckets. Here, depending on the data, we can either pick the equi-depth or the equi-width histogram. We fill the buckets in lexicographic order, meaning that all paths in a certain bucket fall between the begin- and endpath. Furthermore, we store the total amount of occurences and the amount of paths in the bucket. This is done for both versions. The way we initialize the buckets is different though.

Equi-width

With the equi-width version, the budget denotes the maximum amount of items in a bucket. This means that in some buckets, we have got a lot of occurences, while on other buckets we have little occurences.

We generate the equi-width histogram by going over every line of the file generated in Section 5.2.1. When we have had a certain amount of paths, the budget, we write the begin path and end path of the bucket and the total number of occurences of the paths in this bucket and the total number of paths in the bucket to the histogram. Then, we start a new bucket. We do this until we have had all the paths.

Equi-depth

For equi-depth, we fill a bucket until there are at most l occurences in there. The budget here is l, because this denotes how many occurences we want the most. Going over budget is not possible, the path that causes this will be in a new bucket. An exception to this is a path that has more occurences than the budget, this path will get a bucket of its own. This way, every bucket has got around the same amount of occurences. Note that, by construction, no path is present in two different buckets.

We generate the equi-depth histogram by going over every line of the file generated in Section 5.2.1. When we have had a certain amount of occurences, at most as much as the budget, we write the begin path and end path of the bucket and the total amount of occurences of the paths in this bucket and the total amount of paths in the bucket to the histogram.

5.2.3 Usage

Now, every histogram, equi-width or equi-depth, contains the following columns: *beginpath*, *endpath*, *number of occurences* and *number of items (paths)*. If we now want to estimate the amount of occurences for a single path, we can use the following query to get the number of occurences and the number of items in the bucket:

"SELECT count, items FROM histogram WHERE beginpath <= 'path' AND endpath >= 'path'"

Count represents the number of occurences of the paths in the bucket, while items represent the number of paths in the bucket. Here, we use the lexicographical order in the buckets in order to get the correct bucket. By stating that the path should be bigger than or equal to the beginpath and smaller than or equal to the endpath and using the lexicographical properties, we know that the path should fall within one bucket.

In order now to estimate how many times a certain path occurs in the index, we use the query noted above to get the count and the items. Now, we get the estimate by:

$$estimate = counts/items$$

5.2.4 Determine amount of buckets

In the first part of Section 5.2, we have specified two different histograms and we have shown how to use them. What now remains to be specified is a way to determine the amount of buckets. This will be a trade-off between memory and precision. For example, we might pick a bucket per path as this leads to precise estimates, but this will get very costly in terms of memory usage. We could also pick only one bucket, but then the estimations will be very

imprecise.

In order to determine the amount of buckets, we will setup a small experiment. For this experiment, we will try to find the best amount of buckets while taking the memory these buckets cost into account. We will start of by checking the percentage the estimations are off compared to the actual values when using i buckets, where i can go from 1 to the total amount of paths.

The experiment will now be as follows:

- Obtain *total number of occurences* different histograms with 1 to *pathsize* buckets
- Estimate the occurences of every path for every histogram
- Check how much the estimations are off from the actual values

For this experiment, we will only use equi-depth histograms. This is the case, since it has been shown that equi-depth histograms perform better than equi-width histograms [20].

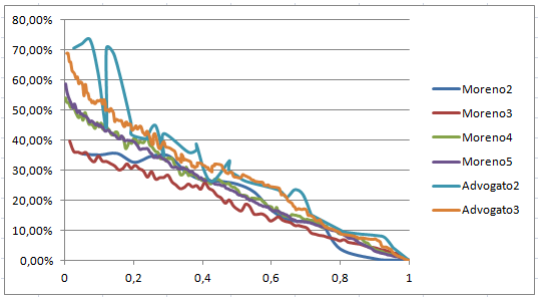


Figure 5.1: The histogram test. On the y-axis the error percentage and on the x-axis the ratio of the number of buckets to the number of paths.

In Figure 5.1, we have shown the result for this experiment. On the y-axis, the average percentage the estimation is off is shown. On the x-axis, the amount of buckets divided by the total number of paths is shown. What we see here is that the relation seems to be linear. We, furthermore, see some data specific events, such as the fact that the estimation for the histogram of Advogato2 goes up and down. This may be related to the fact that some specific choice of buckets may be better, even though we have less buckets. The individual graphs can be found in Appendix B.

By looking at the linear relation we found with the histogram test and taking the storage / performance tradeoff into account, we can determine the amount of buckets we want to use. While doing this, we need to take into account that our memory is using memory blocks. Hence, it would be a waste if we would use an amount of buckets such that one of our blocks will not totally be filled. We know that every row in a bucket contains 4 integers, thus using 16 bytes. If we now look up the block size, 4096 bits in my case, we can calculate that we always need to use a plurality of 32 (4096 / (8 * 16)) buckets. So, when picking a budget, we should take this into account.

5.3 Breaking down paths

After we have parsed the regular expression, as denoted in Section 5.1, into the specific paths, we can start to evaluate these paths. When these paths have length at most k, we only need one lookup in the k-path index. Though, when the path is longer, we need to break the path down in different pieces of length smaller or equal to k and perform joins on these pieces to get the begin- and endnode for the total path. We will discuss different methods for breaking down the paths to pieces in this Section.

5.3.1 Naïvely breaking down paths

The most simple case of breaking down paths is when we only use a k-value of one. This means that every single edge label is a piece. By joining all these pieces, we get the total path and the begin- and endnode that correspond to this path. We will let Postgres determine the join order, which means that we do not use the histogram. We will call this method the **naïve approach**.

Another way would be to use the path index we build, using the maximal k-value, without using the histogram, meaning that we start at the begin of the path and break it into pieces of length k. This means that, at the end of the path, we get one piece of length (length(path) modulo k). We will call this the **semi-naïve approach**.

5.3.2 Breaking down paths using the histogram

The methods above both do not use the histogram for breaking down the paths into pieces. Postgres will try to determine the join order when joining all the different pieces in the most optimal way, but we might be able to do this better. Therefore, we want to use the histogram in order to break down the path into index resolvable pieces. As picking pieces of size k will make sure that less joins are needed, the intuition goes out to only picking pieces of size k. Another method would be to keep picking the pieces resulting in the smallest set, meaning that for that piece, we get the smallest amount of begin- and endnodes that evaluate to the piece relative to the other pieces. Taking all this into account, we have develop two methods of breaking down the paths into index resolvable pieces.

Least support

For the first method, we use the histogram to get the piece of the path of size k with the least support (so it occurs the least in the table compared to the other possible pieces). Next, we recur on the left and right sides that are left. This way, keep picking pieces of size k. Though, we may get pieces that are smaller than k, wasting some of the possibilities. For example, for the path $a \circ b \circ c \circ d \circ e \circ f$ and $k = 2$, if we first pick $d \circ e$ and then $a \circ b$ (assuming these have the least support), we end up with c and f, giving us four joins as we have four pieces, while we could have done it with three. We will call this approach the **minSupport approach**.

Least joins

In contrast to the method shown in Section 5.3.2, the next approach will try to limit the amount of joins necessary. Note that any path with length l can be divided in $\lfloor l/k \rfloor$ pieces

of size k and one piece of size l *modulo* k. We get these pieces by looking at the start or end point of a piece. We can pick the begin of every piece at an index i such that i *modulo* $k = 0$. This gives us the piece that is not size k at the end. Though, we have also got the symmetrical case for which the piece that is not size k is at the beginning. This means that every piece starts at index j such that j *modulo* $k = l$ *modulo* k, where l is the length of the total path. Out of the possible pieces, we again pick the piece that has least support and recur on the left and right side. Note that one of the sides now can be solved by pieces of size k only. Once the piece of size smaller than k has been picked, all pieces should have length k, yielding one piece with size smaller than k and therefore no extra joins. Note that, as we recur on the pieces not picked, the pieces that is not size k does not have to be at the begin / end of the actual path we solve, it will be at the begin / end of one of the sub-paths on which we recur. We will call this the **minJoin approach**.

For example, for the path $a \circ b \circ c \circ d \circ e \circ f \circ g$ and $k = 2$, if we first pick $d \circ e$, we recur on $a \circ b \circ c$ and $f \circ g$. The piece $f \circ g$ can be directly evaluated. For the piece $a \circ b \circ c$, we need to split it into pieces. Say we now pick c, we remain with $a \circ b$ which we have stored in the index.

Note that by the way we use the histogram in breaking down the paths, we also get an order. This order defines the pieces that should be joined first, since these pieces have least support in the table, therefore yielding the smallest sub-tables after joining. We will show how this order is translated to an exact query in Section 5.4.

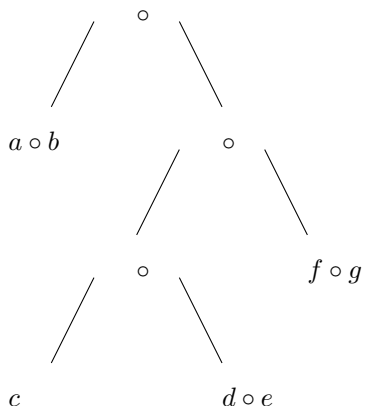


Figure 5.2: Query tree obtained by using the histogram

For the example above, path $a \circ b \circ c \circ d \circ e \circ f \circ g$ and $k = 2$, we first pick $d \circ e$ by the histogram, so this will give the smallest result set. Next, we pick the item with smallest result set, which will be c. This basically tells us that the first steps we should take when solving the query should be to fetch the paths with $d \circ e$ and then join this with the paths with label c. Next, we join this with the smallest of $a \circ b$ and $f \circ g$ to finish with the other one to get the final solution for the query. This may result in the query tree as shown in Figure 5.2.

5.3.3 Summarizing the approaches

We will now summarize the approaches introduced in Sections 5.3.1 and 5.3.2. This summary can be found in Table 5.1.

Method	*k*-level	Usage of histogram
Naïve	1	no
Semi-naïve	1 and above	no
minSupport	1 and above	yes, keep resultsets minimal
minJoin	1 and above	yes, keep amount of joins minimal

Table 5.1: Summary of the different methods

5.4 Querying

In Section 5.1 we have shown how to parse the regular expression over the labels into a path in the graph. In Section 5.2 we define a histogram, which is used in Section 5.3 to break the paths down in pieces and define an order over these pieces. What is left to do now, is making a query using this order. We will show how this is done in this Section.

5.4.1 Fixing the query plan

When handing a query to Postgres, Postgres itself will try to find the optimal way (in terms of computation time) of solving the query. Some heuristics are used for this and, even though this has been well developed, this does not mean that the optimal way will be found. Our assumption is that, since we use a histogram, we can use this to find a more optimal plan than the query planner from Postgres can. Only, in order to test this, we need to somehow fix the order of joins, because the query planner will otherwise pick what it thinks is best.

There are two basic ways of fixing the join order. Either we use the "with" clause (also known as CTE's) to define a new table for every join, or we set the joins exactly as we wish and change a parameter in Postgres such that this order is obeyed.

When using the "with" clause, we define a table for every join. When we have, for example, a "with" clause that defines a table "W", which uses the table "Y" defined by another "with" clause, we first need to calculate "Y" before we can use "W". Hereby, we can fix the join order. For example, if we first want to join A and B and then join this with C, you would get the following:

*"WITH D as (A JOIN B ON (A.s = B.t)), SELECT * FROM D JOIN C ON (D.s = C.r)"*

Here, by first using A JOIN B and renaming this to D, we ensure that we first compute A JOIN B before joining with C.

For the other way, we need to define the order in which we want to join by using joins. We will do this by first defining a join between two tables. Next, we define a join, possibly using the result of the last join, again with renaming, and continue until we are done. Though, normally, the query planner will determine the best join order and execute the joins in that order. Therefore, we need to set the parameter "join_collapse_limit" to 1. This basically ensures that the query planner will only rewrite a join if it results in no more than one from

clause, which will never be the case leading to the join order we have defined.
For example, if we have the same example as we just had, where we first want to join A and B and then join C on the result, we get:

(A JOIN B ON (A.s = B.t)) as D JOIN C ON (D.s = C.r)

Since during some minor tests we have seen that using the "with" clause is best performance wise, we have chosen to use this way of fixing the query plan. This also ensures that we do not have to set parameters in our relational database.

5.4.2 Obtaining the query

As discussed in Section 5.3.1, for some methods (naïve, semi-naïve) we do not need a join order, we let Postgres determine this for us. Here, we only need to join the path index with itself as many times as we have pieces. Every query for a piece gets the begin- and endnode that this piece evaluates to. We then use these to "tie" this part of the path to other pieces, by stating that the endnode of the first piece should be the same as the beginnode of the second piece. We do this with a join. For example, if we have two pieces, a and b with certain values for the path attribute, in SQL, we add: *"WHERE a.endnode = b.beginnode"*. This way, we know that a.path can be before b.path since the endnode of a equals the beginnode of b.

For the other methods (minSupport, minJoin) we already receive a join order, since we have the order of pieces of the path that should give the smallest result set, as shown in Section 5.3.2. The method of obtaining queries is to get the list of possible pieces of the path and then pick the join with the smallest expected result set. We then add the result of the join to the possible pieces and recur on the list. Once the list only contains 1 piece, we know the exact join order. We then form the query such that it uses either the "WITH" or "JOIN" method as discussed in Section 5.4.1 in order to fix the join order.

For example, say we have $k = 2$ and the path is $a \circ b \circ c \circ d \circ e \circ f \circ g$. We break the path down to pieces of length k and get $\{a \circ b, c \circ d, e \circ f, g\}$. We now use the histogram to estimate the result set and we get the following respective values: {3, 4, 2, 5}. For estimating the resultset of a join, various options are possible. For simplicity, we will use size of set A * size of set B when joining set A and set B. In practice, we use *size of set A * (size of set B / selectivity)*, where we pick 10 for the selectivity [22]. Thus, the join resulting in the smallest resultset will be joining $c \circ d$ with $e \circ f$. This gives us $c \circ d \circ e \circ f$ with an estimated size of 8. We then add this to the list of pieces again to get: $\{a \circ b, c \circ d \circ e \circ f, g\}$. The estimated sizes now are: 3, 8, 5. Thus, the next join will be $a \circ b$ with $c \circ d \circ e \circ f$ to get an estimated size of 24. Last of all, we join $a \circ b \circ c \circ d \circ e \circ f$ with g.

After receiving the join order, we should look into the sort of joins we should use. As first join, we want to use the merge join. In order to do this, we will make sure that we get a $(target, source)$ pair instead of a $(source, target)$ pair from the left side of the join, such that we can merge. For the rest of the joins, we let Postgres determine the join sort. In practice, we do not even have to switch the $(source, target)$ pairs, Postgres uses a merge join for all the joins.

6. Experiment setup

6.1 Experiments

In this Section the different experiments will be described. We have shown a few different ways of breaking down paths in order to create a query in Section 5.3, all of which we will test.
Regarding the different methods, we have the following expectations:

- The higher the k-value, the better the performance. Since less lookups and especially less joins need to be done, we assume the queries should be quicker.
- Since the naïve approach does not make use of the higher k-levels and does not use the histogram, it should perform the worst compared to the other approaches. An exception is the case when $k = 1$ and we compare with the semi-naïve approach, since this is almost the same.
- The minJoin and minSupport approach should have better performance than the semi-naïve approach, since these approaches make use of the histogram.

In order to be able to check the expectations, we come to the following approaches we need to test per dataset:

- Naïve approach for $k = 1$
- Semi-naïve approach for all different k-values
- minSupport approach for all different k-values
- minJoin approach for all different k-values

We will use the datasets described in Section 6.4 and the queries described in Section 6.5 and compare the different running times. Furthermore, we will compare against the case in which $k = 1$ (the naïve approach) and against the Datalog approach, which we will introduce here. For all these tests, we will use path indexes without storing the length and with the path converted to an integer.

Furthermore, we will also test "half" indexes, where the inverted paths are not stored. We will test for the reduction in terms of rows of the table and for the performance. We can use the same approaches as for the whole path index, such that we can compare these.

6.2 Datalog approach

As discussed in Section 2.3, an approach towards solving RPQs is to use Datalog. In order to test this approach, we would need to change the data from a graph to Datalog facts. Instead of doing that, we will use recursive SQL views to simulate the behavior of Datalog. We will use the $k = 1$ path index for the respective tables, as the Datalog facts would also only depict the original edges.

Since these recursive views only differ from the naïve approach for recursive queries, we will only test them against the queries containing recursion.

6.3 Environment

In this section, the actual database and the connection to the database will be discussed.

6.3.1 Postgres

Even though we are dealing with graph data, we will use a relational database in order to store the path index. The relational database makes a lot of sense to store the path index, since the path index used in this thesis will hold the path, the begin and the end node of that path. This can easily be stored in a relational database. This does mean that we need to translate the data, as it used to be a graph. In Chapter 4 we explained how this is done.

For the exact relational database, we choose Postgres version 9.2.5. We use Postgres, because it is well-developed, open source and well documented. Furthermore, Postgres allows us to manipulate various different settings, which might be useful in order to improve the performance, e.g. while manipulating the join order. Though Postgres is used here, most, if not all, of the paradigms used in this thesis will be applicable to other relational databases as well.

6.3.2 Python and Psycopg2

We will use python for path index and histogram generation, parsing, query generation and outputting of the results. Though, the actual query evaluation is done by Postgres. In order to feed the query to Postgres, pscyopg2 is used.

Psycopg2 is a module of python, which can be used to connect to a Postgres database. A connection needs to be defined with the database. Next, we can get a cursor from the database. With this cursor, we can execute queries and fetch the results.

For large results, which can occur when querying large datasets or with high k values, psycopg2 might give an out of memory error in some cases. In order to prevent this, we have chosen to use a server side cursor. The main advantage of this cursor is that it does not load all the data in, but it lets you fetch the data in pieces. This way, psycopg2 will not run out of memory.

Dataset	*Advogato*	*Moreno*
#Nodes	6541	70
#Labels	3	2
k = 1 #paths	102252	732
k <= 2 #paths	6952065	7576
k <= 3 #paths	200602767	59352
k <= 4 #paths	-	392889
k <= 5 #paths	-	2264945

Table 6.1: Datasets

6.4 Datasets

Since an undirected graph can be represented by a directed graph by replacing every edge in the undirected graph with two directed edges, we will focus at directed graphs. Furthermore, since we are dealing with RPQs, we need to have labels over the edges, thus we want labeled graphs.

One of the datasets will be the Advogato dataset [1, 18], obtained from the Konect repository of datasets[1][14]. Furthermore, we will use the Moreno [2, 7], also obtained from the Konect repository. Some information about these datasets is shown in Table 6.1. For the Advogato dataset, we will go up to a k-value of three. For Moreno, as this is a very small dataset, we will get to a k value of five.

6.5 Queries

We will use queries containing a Kleene star, since this is the main goal of this thesis. Though, we also want to be efficient on "normal" queries, therefore we will also evaluate the performance of the path index using some queries that will not contain the Kleene star. We want to have a combination of large / small and recursive / non-recursive queries. We define a small query as a query with a resulting path of size at most 4 (containing at most 4 labels). Large queries will have at least 4 and at most 8 labels. This gives us the following combinations:

- small query without recursion
- small query with recursion
- large query without recursion
- large query with recursion

For some datasets, we might choose to get even further than the large queries. This will be noted in the section for the queries for that dataset. We will now shortly describe the different queries per dataset. The actual queries can be found in Appendix A. In the actual experiments, we will run every query fifteen times. The first five times are the warmup tests, these are not considered for the final results. The next ten tests are used for the results.

[1] http://konect.uni-koblenz.de/

Furthermore, we have ran "VACUUM ANALYZE" before running queries, such that all the statistics of Postgres are up to date.

- **Advogato:** The Advogato dataset contains 3 different labels: {.6, .8, 1}. We will use these to get the different queries as described above. We will have 3 small queries without recursion and 2 of all the other query types.
- **Moreno:** The Moreno dataset is rather small. Therefore, we will not use small queries here, but instead use large queries and very large (with over 8 labels) queries. Furthermore, the Moreno dataset has got 2 different labels: {1, 2}. We will use these to get the different queries.

7. Results

We will discuss the main results of the experiments. Though, not all results will be shown here. The rest of the results can be found in Appendix C and Appendix D.

7.1 Experiment Results

7.1.1 Advogato

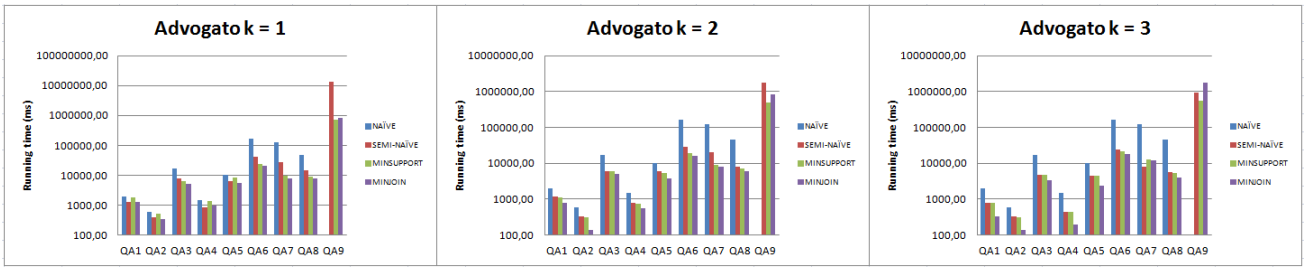


Figure 7.1: Results for the different queries in milliseconds

In Figure 7.1, the results for the different Advogato queries and the different methods are shown. We see here that the naïve case always performs the worst, which is as expected. The semi-naïve case is second-worst in all cases but 2. The minJoin and minSupport approach perform similarly, with minJoin generally outperforming minSupport. We see that our assumptions mostly hold. Only the two cases where the semi-naïve case is better than minSupport / minJoin. This is probably because the query planner does find a good plan already, and our heuristics cannot improve this. Furthermore, the time for getting these heuristics will make the minSupport / minJoin approach slower. We also see that, compared to the naïve approach, the minSupport / minJoin approach perform up to five times faster for the $k = 3$ case.

In Figure 7.2, we see the minJoin approach executed for the different queries on the Advogato dataset. We see that, for most of the queries, increasing the k level does decrease the running time, as expected. Though, for two queries, we see $k = 2$ performs better than $k = 3$ (queries 2 and 6) and for one query we even see that $k = 3$ performs worse than $k = 1$ (query 7). For query 2 (QA2:.8 ∘ 1|.6 ∘ .8), this seems logical, since any k higher than one results in the same amount of lookups and joins, namely two lookups and no joins. Since for higher k levels the table grows, a lookup will take more time, hence explaining why $k = 2$ performs better than $k = 3$. We see that the minSup approach behaves like the minJoin approach. For the

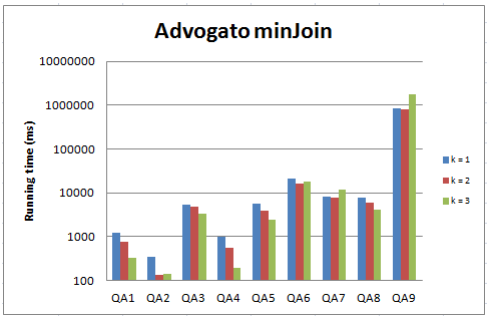


Figure 7.2: Results for the different queries for different k levels on Advogato using the minJoin approach

semi-naïve approach, we see a decrease in running time when we increase the k value for every single query. The data and graphs for these approaches can be found in Appendix C.

7.1.2 Moreno

For the Moreno dataset, we see that there are a lot more differences regarding our assumptions. Note that the differences between methods / k-values are not really big. For example, we see that the higher k-value is not always the best and that the naïve case is not always the worst, which is also the case in 7.3. The higher k-value not always being the best can be explained by the lengths of the paths in the query in some cases. For example, QM1 $(1 \circ 1 \circ 2 \circ (1 \circ 2)!)$ has length five, and we see that the $k = 5$ path index performs the best. QM2 $(2 \circ 2 \circ 2 \circ 1 \circ 2 \circ 2)$ has length 6, and we see that the $k = 3$ path index performs the best here. This is due to the fact that the $k = 3$ path index requires the least amount of joins, namely one, and that it is the smallest of the path indices that require one joins. The $k = 4$ and $k = 5$ path indices both require one join as well, but they are both bigger in size, since they contain more paths. The $k = 2$ path index has a smaller size, but requires more joins.

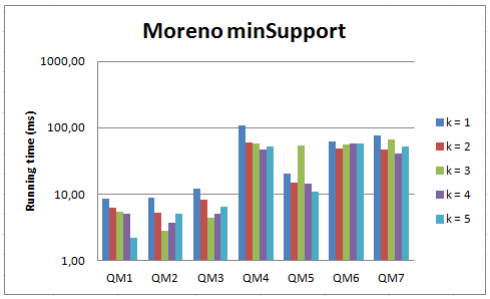


Figure 7.3: Results for the different queries for different k levels on Moreno using the minSupport approach

The full results for the Moreno dataset can be found in C.

7.1.3 "Half" path index

Considering the "Half" path index, as described in Section 4.4.1, the reduction in storage and the actual performance of the index are important factors. In Table 7.1, we have shown the

Dataset	*Type*	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Advogato	**Regular**	102252	6952065	200602767	-	-
	"Half"	51127	4742822	101568173	-	-
	Reduction (%)	50,00%	31,78%	49,37%	-	-
Moreno	**Regular**	732	7576	59352	392889	2264945
	"Half"	366	4825	30713	209059	1145087
	Reduction (%)	50,00%	36,31%	48,25%	46,79%	49,44%

Table 7.1: Rows and reduction of the "half" path index compared with the regular path index for the different datasets and different k-values.

amount of rows per table for all the different datasets for both the regular path index and the "half" path index. This table shows that using a "half" path index indeed helps minimalizing the storage. The "half" index can contain up to 50% less rows.

Next, we will analyze the performance of the "half" path indexes. As a sanity check, we look if the assumptions hold. In general, we see that this is the case. The relevant tables for this can be found in Appendix D. The real experiment here is to see how some paths not being available changes the running times. In order to show this, we have run the same experiments as for the regular path index and we compare the running times with the running times we found there. Basically, there are a few possible queries and a few assumptions that go with these possibilities. If the query contains only paths that are in the path index, the assumption is that the "half" path index' performance is better, due to the fact that this contains less rows, leading to a decrease in lookup and join times. Though, when working with multiple paths for which we have to look up the inverse, the performance will be less compared to the regular path index.

In Table 7.2 we have compared the performance of the "half" path index with the regular path index in terms of running times. The table shows the percentage the "half" path index performance is off, the calculation for this is as follows: (running time "half" / running time regular) - 1.
What we see here is that the percentages can differ by quite a lot over the different k-levels and different methods. This is probably due to the fact that the query is broken up in different pieces depending on the method selected and the k-value. This also determines the amount of times we have to look up an inverse, hereby indirectly influencing the running times. In general, we can state that the performance of the "half" path index will be worse on average. Though, depending on the queries, a "half" path index might bring a solution when not enough storage is present.
The full results of the "half" path index can be found in Appendix D.

7.2 Datalog approach

In this Section, we will discuss the comparison with the Datalog approach, as described in 6.2. We will compare the recursive SQL queries with the best approach for that query. In Tables 7.3 and 7.4 the results for the recursive SQL queries can be found compared to the

Dataset	*Type*	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Advogato	**Semi-naïve**	217,11%	-3,99%	-21,07%	-	-
	minSup	-11,62%	7,83%	-3,85%	-	-
	minJoin	1,80%	21,78%	-4,65%	-	-
Moreno	**Semi-naïve**	-5,96%	11,18%	12,41%	25,79%	-72,07%
	minSup	9,93%	28,20%	248,92%	339,59%	263,63%
	minJoin	-3,14%	16,54%	108,61%	213,59%	226,07%

Table 7.2: Average percentage the "half" path index' performance is off compared to the regular path index.

Query	*Datalog approach (ms)*	*Best approach (ms)*	*Best approach (approach)*
QA5: $(.8\|1 \circ .6)\{1,2\}$	5861885,96	2441,14	minJoin ($k = 3$)
QA8: $(1)\{1,4\}$	91685,95	4083,48	minJoin ($k = 3$)
QA9: $(.8 \circ 1\|(.6 \circ 1)!)\{2,3\}$	15188650,74	499501,86	minSupport ($k = 2$)

Table 7.3: Recursive sql queries for the Advogato dataset compared to the best path index approach

best approach of the path index. For the Advogato dataset, we see that the Datalog approach is a lot worse compared to the path index approach. For the Moreno dataset, the difference is less convincing. For the last query, we even see that the Datalog approach performs better. Though, in general we can state that the path index outperforms the Datalog approach.

Query	*Datalog approach (ms)*	*Best approach (ms)*	*Best approach (approach)*
QM3: $(2 \circ 1 \circ 2)\{1,2\}$	18,10	2,90	semi-naïve ($k = 4$)
QM4: $(2 \circ 2\|(1 \circ 1)!)\{2,3\}$	67,78	31,73	semi-naïve ($k = 4$)
QM7:$(1 \circ (2)! \circ 1)\{3,4\}$	79,31	47,98	minSupport ($k = 2$)
QM8:$(1 \circ 2)\{3,6\}$	31,81	40,35	minSupport ($k = 4$) / minJoin ($k = 4$)

Table 7.4: Recursive sql queries for the Moreno dataset compared to the best path index approach

7.3 Maximal and average speedup

In this Section, we will look at the maximal and the average speedup achieved per dataset. The maximal speedup will be the amount of times the **best** (either semi-naïve, minSupport or minJoin) approach is faster than the naïve or Datalog approach. For the average speedup, we have taken the average amount of times the different approaches (semi-naïve, minSupport and minJoin) where faster than the naïve or Datalog approach. This is the averages over all the different approaches, for all the different queries. The results can be found in Table 7.5.

Dataset	*Compared against*	*Maximal speedup*	*Average speedup*
Advogato	Naïve approach	15,5	4,87
	Datalog approach	2401	1211
Moreno	Naïve approach	802	112
	Datalog approach	6,2	2,7

Table 7.5: Maximal and average speedup for different datasets compared to the naïve / Datalog approach

8. Conclusion

In this thesis, we have shown how to use a path index for RPQ evaluation. We started with setting up the path index. Here, we also looked at some ways to reduce the storage size. Next, we started with parsing the RPQs to paths over the edges of the graph. By breaking these paths in pieces (if necessary), we have shown how to get a query plan with or without the use of a histogram.

After that, we have introduced a parser, able to translate RPQs to paths and hereby providing a solution to the first sub-question. We first defined the regular expression language that we were going to use. Next, we showed how to translate these RPQs into paths usable by the path index.

Furthermore, we have introduced four different methods to use the path index. Where the naïve approach is the most basic approach, since it does not use the histogram and only uses a $k = 1$ path index, the semi-naïve approach already makes use of higher levels of the path index. The minSupport and minJoin approaches both use the histogram. By introducing these approaches, we have shown how to evaluate arbitrary long paths and how to use the histogram in order to do so, as stated in sub-questions two and three. Where the minSupport approach focusses at always picking pieces of the path with least support, the minJoin approach keeps the number of joins in the query minimal. We have compared our approaches with the naïve method, which is like automata based methods, and the Datalog approach. The performance of our approach was better in most cases. Hereby, we have shown that path indexes provide an efficient way towards RPQ evaluation.

The expectations stated at the beginning of Chapter 6 were shown in general to be correct. In most cases using a path index with higher k-value increases performance, but in some cases, for example where a path index with higher k-value still requires the same amount of joins, we have seen that a path index with lower k-value can outperform the higher k-value path index. Furthermore, in general, we see that the naïve approach performs the worst, followed by the semi-naïve approach. The minSupport and minJoin approaches perform similarly and show how we can use the path index in addition to the histogram to increase RPQ evaluation performance.

8.1 Limitations and Future work

One of the issues of this thesis is the selectivity used for the histograms. Even though a factor of 10 is used more often, it is too general and the histogram should be able to perform better with a more specific selectivity. Even though this is not the general problem of the thesis,

this is an essential item for the running times, which explains why the selectivity should be addressed better. Future work lies in improving the selectivity, for example by storing the amount of beginnodes and endnodes for the paths in a certain bucket, such that we know how many nodes we can expect.

Regarding the storage size, we have seen that this can cause some trouble. For certain datasets, we have seen that we could not get the path index up to a certain level due to storage size restrictions while testing. For these datasets, the path index would run out of disk space. We mapped the strings denoting a path to integers and left out the length of the path in order to decrease storage size. Furthermore, we also introduced the "half" path index. In this index, either the path or its inverse is stored, were in the normal index both would be stored. We see that there is a relation between the storage size of the index and the running time of the RPQs. Even with the additions, the storage size remains to be a problem.

Therefore, another topic for future work is the size of the path index. As most databases nowadays mostly run only a few specific queries, a way to keep the storage size small can be to only store the parts necessary to solve the specific queries. Though, the exact improvements and storage size should be further investigated.

Furthermore, more research on the "half" path index is also necessary. Methods that limit the amount of times an inverse has to be looked up may contribute to the performance of the "half" path index. Therefore, it would be interesting to see if we can increase the performance of "half" path indexes by the described methods.

Last, more experiments on even larger datasets may give more insights in the performance of the path index. Some experiments have already been performed on the larger DBLP dataset [3, 15], but we experienced challenges with index size and query processing. In order to apply the methods of this thesis in practice, the support of such large datasets is important. Hence, accommodating larger graphs is clearly a basic direction for future investigation.

Bibliography

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

[2] Highschool network dataset – KONECT, October 2014.

[3] Dblp network dataset – KONECT, May 2015.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[5] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 122–133. ACM, 1997.

[6] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, pages 176–185, 2000.

[7] James Samuel Coleman. Introduction to Mathematical Sociology. *London Free Press Glencoe*, 1964.

[8] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. A graphical query language supporting recursion. In *ACM SIGMOD Record*, volume 16, pages 323–330. ACM, 1987.

[9] Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. On implementing provenance-aware regular path queries with relational query engines. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 214–223. ACM, 2013.

[10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6(3):313–338, 2012.

[11] Gang Gou and R. Chirkova. Efficiently querying large xml data repositories: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10):1381–1403, Oct 2007.

[12] Andrey Gubichev, Srikanta J Bedathur, and Stephan Seufert. Sparqling kleene: fast property paths in rdf-3x. In *First International Workshop on Graph Data Management Experiences and Systems*, page 14. ACM, 2013.

[13] André Koschmieder and Ulf Leser. Regular path queries on large graphs. *SSDBM*, pages 177–194, 2012.

[14] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013.

[15] Michael Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *Proc. Int. Symposium on String Processing and Information Retrieval*, pages 1–10, 2002.

[16] Leonid Libkin and Domagoj Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, pages 74–85. ACM, 2012.

[17] Yongming Luo, François Picalausa, George HL Fletcher, Jan Hidders, and Stijn Vansummeren. Storing and indexing massive rdf datasets. In *Semantic Search over the Web*, pages 31–60. Springer, 2012.

[18] Paolo Massa, Martino Salvetti, and Danilo Tomasoni. Bowling alone and trust decline in social network sites. In *Proc. Int. Conf. Dependable, Autonomic and Secure Computing*, pages 658–663, 2009.

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

[20] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD Record*, volume 25, pages 294–305. ACM, 1996.

[21] Petra Selmer, Alexandra Poulovassilis, and Peter T Wood. Implementing flexible operators for regular path queries. In *CEUR Workshop Proceedings*, volume 1330, pages 149–156. CEUR Workshop Proceedings, 2015.

[22] Abraham Silberschatz, Henry F Korth, S Sudarshan, et al. *Database system concepts*, volume 6. McGraw-Hill Singapore, 2011.

[23] Dan Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems*, 27(1):1–62, 2002.

[24] Kam-Fai Wong, Jeffrey Xu Yu, and Nan Tang. Answering xml queries using path-based indexes: a survey. *World Wide Web*, 9(3):277–299, 2006.

[25] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1), 2012.

[26] N. Yakovets, P. Godfrey, and J. Gryz. Towards Query Optimization for SPARQL Property Paths. *ArXiv e-prints*, April 2015.

[27] Xifeng Yan and Jiawei Han. Graph indexing. In *Managing and Mining Graph Data*, pages 161–180. Springer, 2010.

A. Queries

A.1 Advogato

The different queries for the Advogato dataset:

- QA1: $.8 \circ 1 \circ .6$
- QA2: $.8 \circ 1 | .6 \circ .8$
- QA3: $(1 \circ .8 \circ 1 \circ .6)^{-}$
- QA4: $.6 \circ (.8 \circ 1)\{0, 1\}$
- QA5: $(.8 | 1 \circ .6)\{1, 2\}$
- QA6: $1 \circ .8 \circ .6 \circ .6 \circ .8 | (.6 \circ .8 \circ 1)^{-}$
- QA7: $1 \circ .8 \circ 1 \circ .6 \circ 1$
- QA8: $(1)\{1, 4\}$
- QA9: $(.8 \circ 1 | (.6 \circ .1)^{-})\{2, 3\}$

A.2 Moreno

The different queries for the Moreno dataset:

- QM1: $1 \circ 1 \circ 2 \circ (1 \circ 2)^{-}$
- QM2: $2 \circ 2 \circ 2 \circ 1 \circ 2 \circ 2$
- QM3: $(2 \circ 1 \circ 2)\{1, 2\}$
- QM4: $(2 \circ 2 | (1 \circ 1)^{-})\{2, 3\}$
- QM5: $(1 \circ 1 \circ 2)^{-} \circ 2 \circ 1 \circ (2 \circ 1 \circ 2)^{-} \circ 1$
- QM6: $(1 \circ (2)^{-} \circ 1)\{3, 4\}$
- QM7: $(1 \circ 2)\{3, 6\}$

B. Histogram experiment results

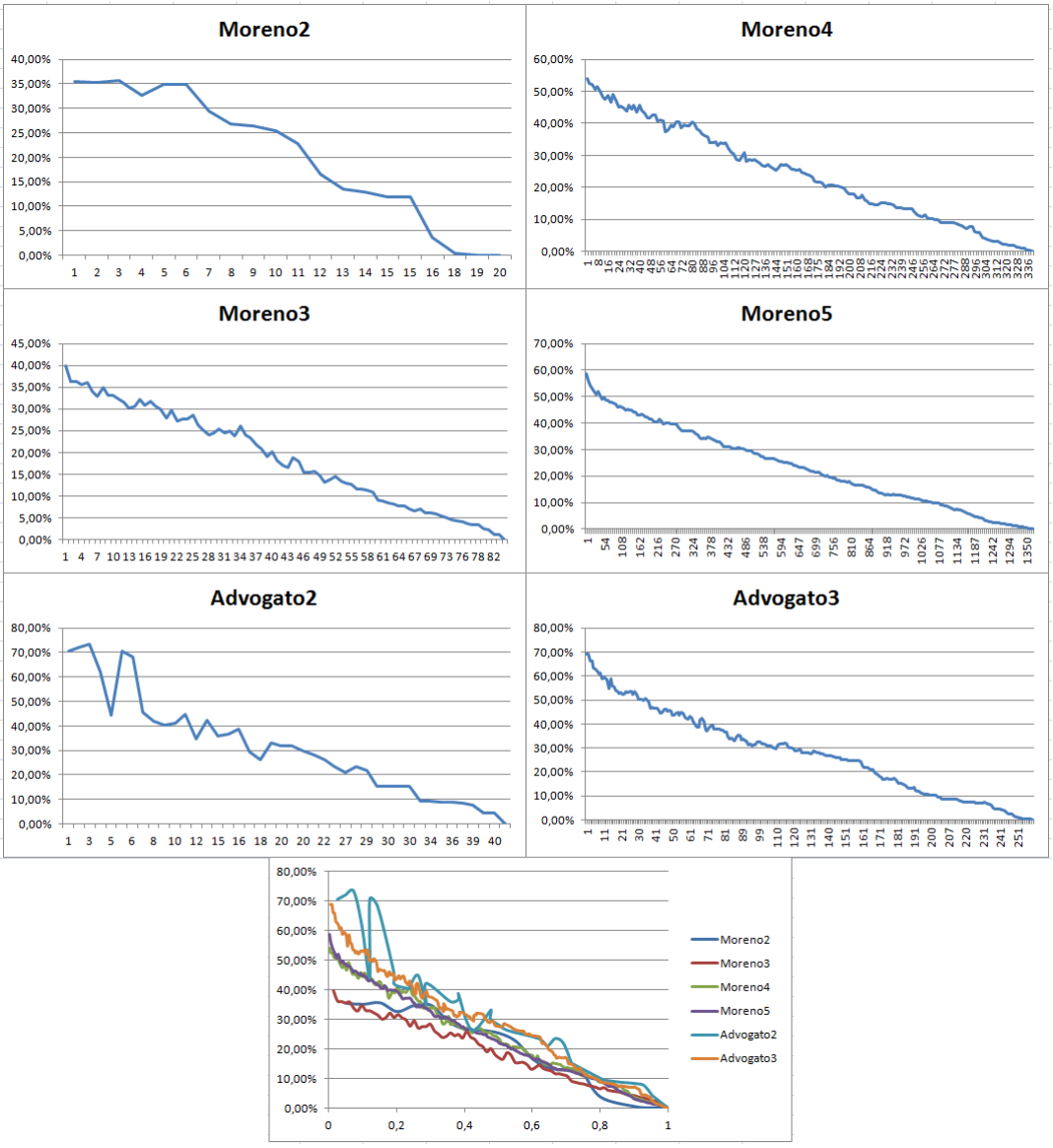


Figure B.1: The graphs for the histogram tests for all the different datasets. On the y-axis the percentage the estimate is off, on the x-axis the number of buckets.

C. Regular path index experiment results

C.1 Advogato results

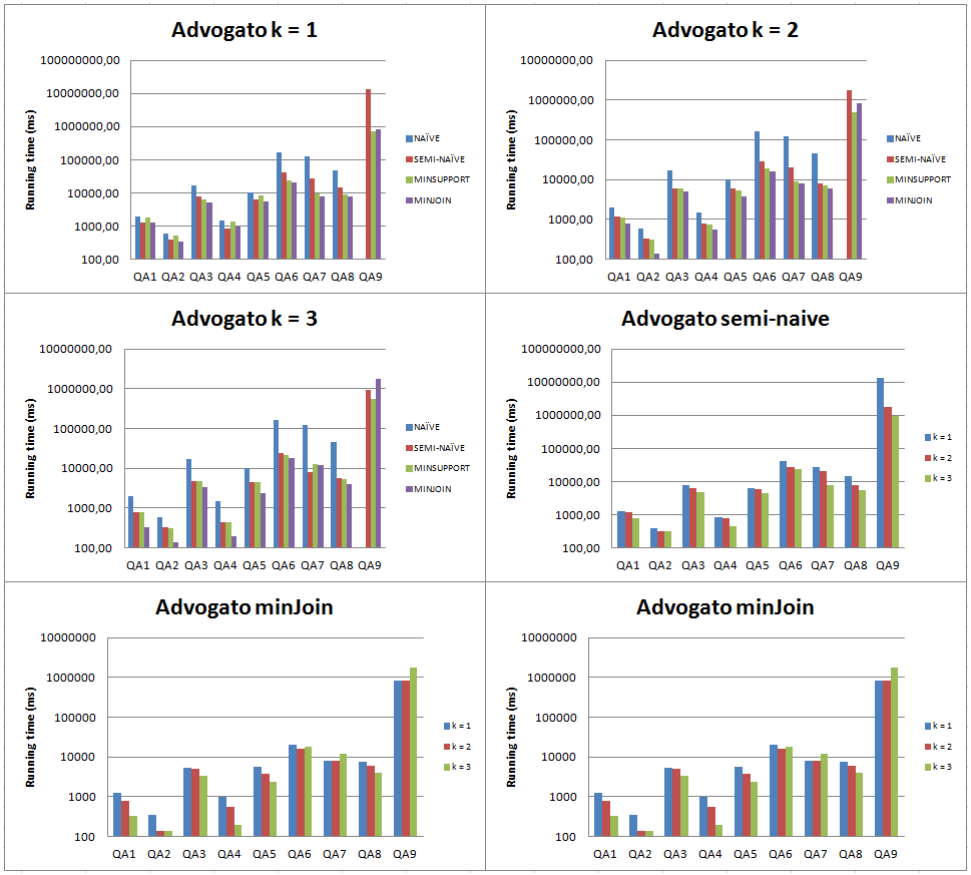


Figure C.1: Graphs for the different k-values and the different methods for the Advogato dataset.

C.2 Moreno results

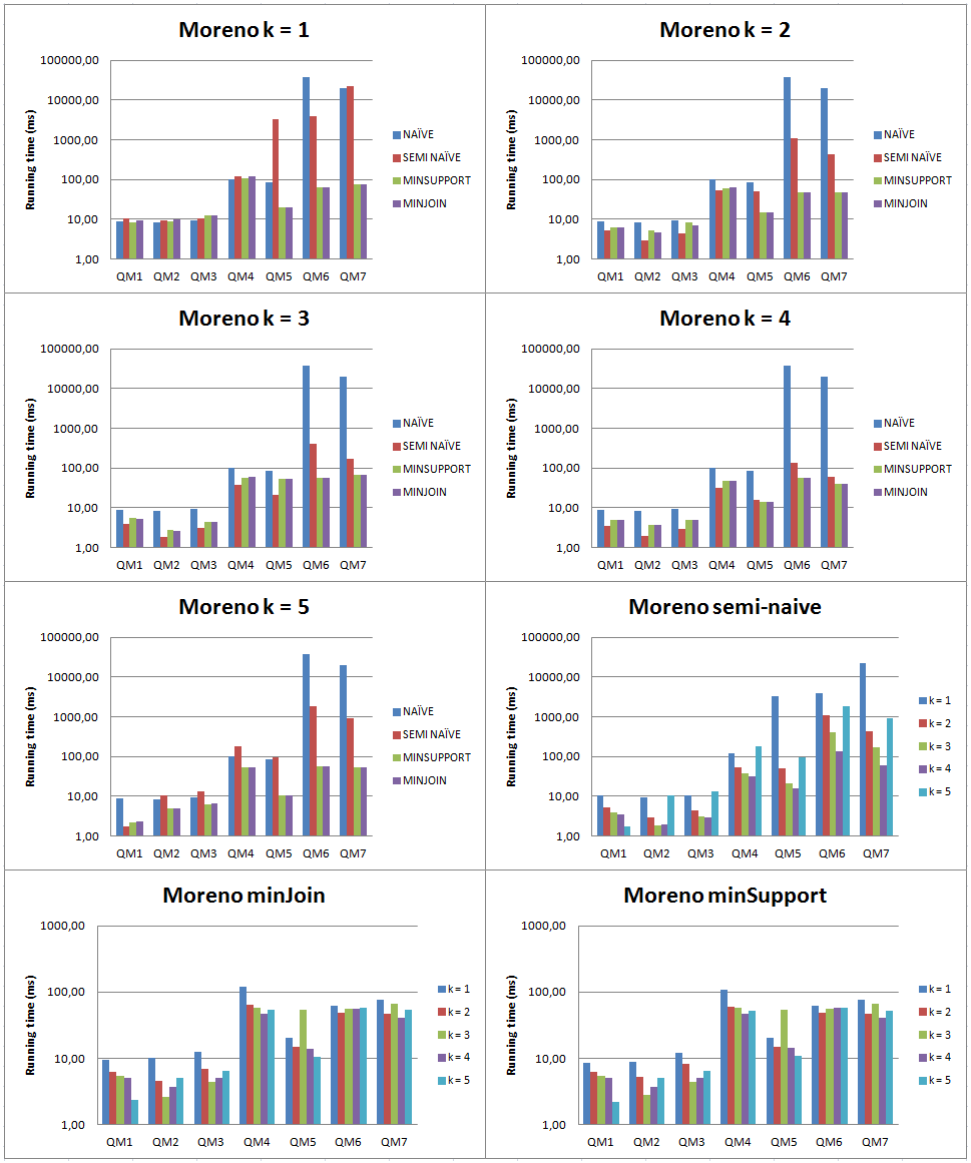


Figure C.2: Graphs for the different k-values and the different methods for the Moreno dataset.

D. “Half” path index experiment results

D.1 Advogato results

Figure D.1 and Tables D.1 to D.3 show the results for the “half” path index for the different approaches on the Advogato dataset.

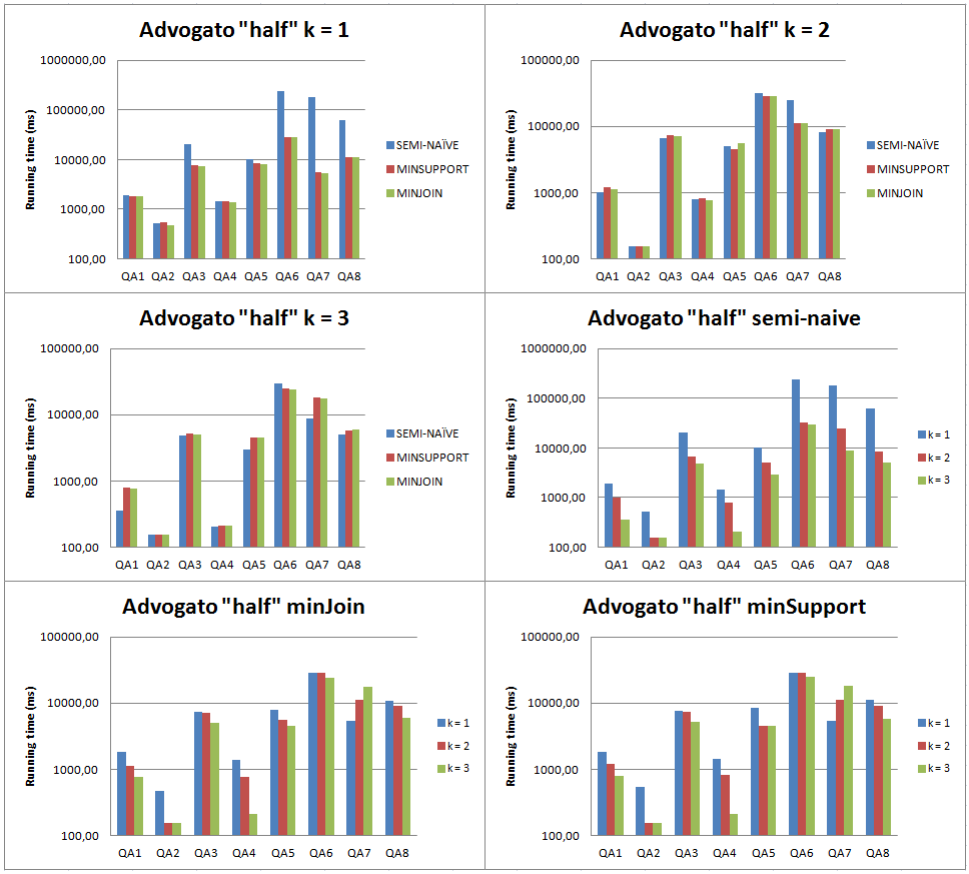


Figure D.1: Graphs for the different k-values and the different methods for the Advogato dataset for the “half” path index.

$k = \mathbf{1}$			$k = \mathbf{2}$			$k = \mathbf{3}$		
half	*full*	*percent off*	*half*	*full*	*percent off*	*half*	*full*	*percent off*
1889,20	1250,90	51,03%	1003,34	1151,54	-12,87%	360,33	787,54	-54,25%
525,01	399,99	31,26%	153,29	326,48	-53,05%	152,79	324,78	-52,96%
20326,65	7819,49	159,95%	6663,44	6115,13	8,97%	4886,60	4729,80	3,32%
1455,74	835,00	74,34%	788,19	785,55	0,34%	208,26	453,23	-54,05%
10192,19	6349,36	60,52%	5054,83	5898,86	-14,31%	2949,56	4564,94	-35,39%
238707,71	42153,84	466,28%	32363,70	27993,96	15,61%	29319,74	23551,50	24,49%
178048,51	26672,53	567,54%	24772,65	20694,75	19,70%	8807,19	8069,54	9,14%
63156,96	14826,78	325,97%	8305,96	8008,41	3,72%	5056,63	5547,25	-8,84%
average		217,11%	*average*		-3,99%	*average*		-21,07%

Table D.1: Performance of the "half" path index compared to the full index for the different k-value for the Advogato dataset using the semi-naïve approach

$k = \mathbf{1}$			$k = \mathbf{2}$			$k = \mathbf{3}$		
half	*full*	*percent off*	*half*	*full*	*percent off*	*half*	*full*	*percent off*
1844,48	1810,44	1,88%	1228,51	1147,29	7,08%	804,85	788,39	2,09%
536,13	539,43	-0,61%	154,65	324,44	-52,33%	157,36	328,83	-52,14%
7766,90	8842,41	-12,16%	7442,06	6064,28	22,72%	5147,28	4676,23	10,07%
1452,95	1307,48	11,13%	818,35	750,85	8,99%	213,79	456,56	-53,17%
8386,90	8242,05	1,76%	4528,45	5675,44	-20,21%	4592,83	4498,84	2,09%
28588,25	32566,63	-12,22%	28792,56	19751,44	45,77%	24709,68	21389,95	15,52%
5456,26	14686,14	-62,85%	11162,40	9041,55	23,46%	17924,35	12906,44	38,88%
11345,99	14157,56	-19,86%	9088,59	7149,26	27,13%	5801,35	5480,54	5,85%
average		-11,62%	*average*		7,83%	*average*		-3,85%

Table D.2: Performance of the "half" path index compared to the full index for the different k-value for the Advogato dataset using the minSupport approach

$k = \mathbf{1}$			$k = \mathbf{2}$			$k = \mathbf{3}$		
half	*full*	*percent off*	*half*	*full*	*percent off*	*half*	*full*	*percent off*
1832,11	1371,94	33,54%	1139,48	940,79	21,12%	765,23	789,48	-3,07%
473,04	389,83	21,35%	154,53	141,16	9,47%	153,06	319,69	-52,12%
7379,64	7574,50	-2,57%	7151,95	5802,98	23,25%	4973,79	4679,90	6,28%
1384,31	1139,86	21,45%	780,49	681,79	14,48%	214,28	447,75	-52,14%
8021,98	6433,26	24,70%	5650,09	4557,80	23,97%	4594,14	4440,68	3,46%
28434,93	29488,54	-3,57%	28394,39	21160,70	34,18%	24322,14	21385,23	13,73%
5303,21	15646,58	-66,11%	11360,56	9551,84	18,94%	17399,80	12937,81	34,49%
10937,91	12770,59	-14,35%	9005,79	6988,54	28,87%	6012,10	5359,14	12,18%
average		1,80%	*average*		21,78%	*average*		-4,65%

Table D.3: Performance of the "half" path index compared to the full index for the different k-value for the Advogato dataset using the minJoin approach

D.2 Moreno results

Figure D.2 and Tables D.4 to D.6 show the results for different approaches for the "half" path index for the Moreno dataset.

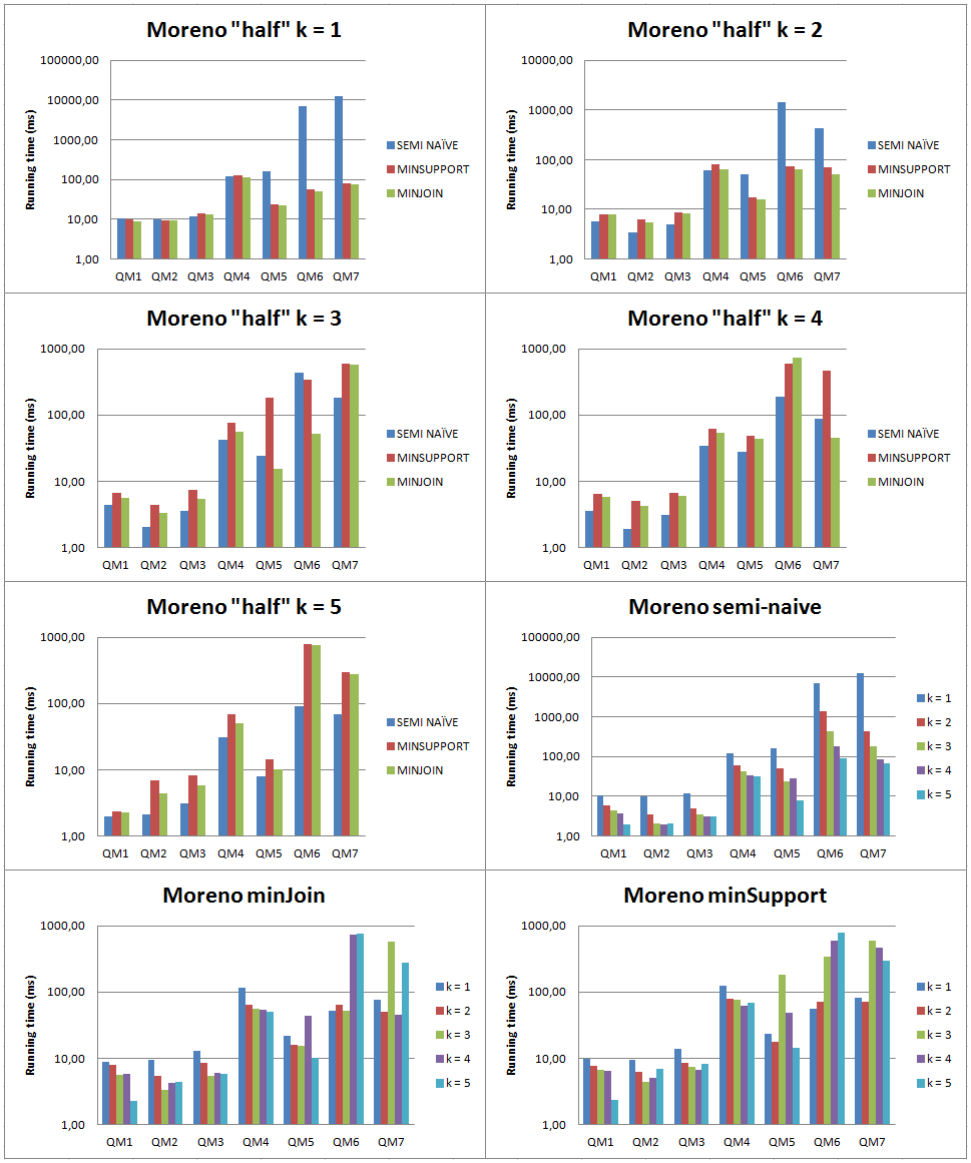


Figure D.2: Graphs for the different k-values and the different methods for the Moreno dataset for the "half" path index.

$k = 1$			$k = 2$			$k = 3$		
half	*full*	*percent off*	*half*	*full*	*percent off*	*half*	*full*	*percent off*
10,40	10,25	1,46%	5,85	5,20	12,50%	4,48	3,85	16,23%
10,16	9,15	11,07%	3,44	3,00	14,58%	2,04	1,80	13,19%
12,09	10,79	12,05%	4,90	4,48	9,50%	3,59	3,13	14,80%
120,46	117,84	2,23%	61,08	53,88	13,36%	41,79	36,70	13,86%
161,44	3359,31	-95,19%	51,59	49,79	3,62%	24,40	21,53	13,36%
6788,56	3921,93	73,09%	1409,89	1110,99	26,90%	437,53	410,23	6,65%
12172,99	22732,75	-46,45%	426,31	436,00	-2,22%	181,78	167,11	8,77%
average		-5,96%	*average*		11,18%	*average*		12,41%

$k = 4$			$k = 5$		
half	*full*	*percent off*	*half*	*full*	*percent off*
3,63	3,44	5,45%	1,95	1,70	14,71%
1,94	2,00	-3,12%	2,13	10,80	-80,32%
3,09	2,90	6,47%	3,14	13,46	-76,69%
34,48	31,73	8,67%	30,81	180,90	-82,97%
27,46	15,63	75,76%	8,00	93,81	-91,47%
185,73	132,94	39,71%	92,45	1892,48	-95,11%
86,76	58,78	47,62%	68,45	927,71	-92,62%
average		25,79%	*average*		-72,07%

Table D.4: Performance of the "half" path index compared to the full index for the different k-value for the Moreno dataset using the semi-naïve approach

$k = 1$			$k = 2$			$k = 3$		
half	*full*	*percent off*	*half*	*full*	*percent off*	*half*	*full*	*percent off*
9,80	8,53	14,96%	7,79	6,25	24,60%	6,59	5,44	21,15%
9,53	8,81	8,09%	6,29	5,33	18,08%	4,41	2,80	57,59%
14,11	12,20	15,68%	8,58	8,19	4,73%	7,35	4,40	67,05%
125,79	107,61	16,89%	79,88	60,89	31,18%	76,41	56,91	34,26%
23,50	20,26	15,98%	17,81	15,03	18,55%	180,20	54,04	233,47%
55,55	62,48	-11,08%	71,84	47,98	49,74%	340,29	55,60	512,03%
82,48	75,68	8,99%	70,58	46,89	50,52%	602,18	65,68	816,90%
average		9,93%	*average*		28,20%	*average*		248,92%

$k = 4$			$k = 5$		
half	*full*	*percent off*	*half*	*full*	*percent off*
6,50	5,08	28,08%	2,33	2,23	4,49%
5,06	3,70	36,82%	6,89	5,09	35,38%
6,66	5,00	33,25%	8,18	6,40	27,73%
62,94	46,86	34,30%	68,51	52,96	29,36%
48,41	14,20	240,93%	14,15	10,80	31,02%
605,08	57,06	960,37%	775,23	57,49	1248,51%
461,36	40,35	1043,40%	298,88	52,54	468,88%
average		339,59%	*average*		263,63%

Table D.5: Performance of the "half" path index compared to the full index for the different k-value for the Moreno dataset using the minSupport approach

$k = 1$			$k = 2$			$k = 3$		
half	*full*	*percent off*	*half*	*full*	*percent off*	*half*	*full*	*percent off*
8,88	9,43	-5,84%	8,06	6,18	30,57%	5,68	5,38	5,58%
9,46	9,99	-5,26%	5,41	4,63	17,03%	3,36	2,60	29,33%
13,13	12,36	6,17%	8,39	6,94	20,90%	5,44	4,39	23,93%
116,06	122,18	-5,00%	64,24	63,51	1,14%	55,64	58,38	-4,69%
21,91	20,48	7,02%	16,24	14,93	8,79%	15,66	53,94	-70,96%
51,33	62,64	-18,06%	64,00	48,88	30,95%	51,33	55,24	-7,08%
76,30	77,08	-1,01%	49,93	46,91	6,42%	580,55	65,66	784,14%
average		-3,14%	*average*		16,54%	*average*		108,61%

$k = 4$			$k = 5$		
half	*full*	*percent off*	*half*	*full*	*percent off*
5,78	5,10	13,24%	2,30	2,34	-1,60%
4,23	3,73	13,42%	4,40	5,09	-13,51%
5,94	5,00	18,75%	5,81	6,48	-10,23%
54,21	46,65	16,21%	49,60	53,10	-6,59%
44,55	14,10	215,96%	10,30	10,64	-3,17%
739,60	56,64	1205,85%	755,51	57,95	1203,73%
45,06	40,35	11,68%	274,08	53,34	413,85%
average		213,59%	*average*		226,07%

Table D.6: Performance of the "half" path index compared to the full index for the different k-value for the Moreno dataset using the minJoin approach