# Regular Path Queries on Large Graphs

André Koschmieder
Humboldt-Universität zu Berlin
Dept. of Computer Science
Berlin, Germany
koschmie@informatik.hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Dept. of Computer Science
Berlin, Germany
leser@informatik.hu-berlin.de

## ABSTRACT

The significance of regular path queries (RPQs) on graph-like data structures has grown steadily over the past decade. RPQs are, often in restricted forms, part of graph-oriented query languages such as XQuery/XPath and SPARQL, and have applications in areas such as semantic, social, and biomedical networks. However, existing systems for evaluating RPQs are restricted either in the type of the graph (e.g., only trees), the type of regular expressions (e.g., only single steps), and/or the size of the graphs they can handle. No method has yet been developed that would be capable of efficiently evaluating general RPQs on large graphs, i.e., with millions of nodes/edges.

We present a novel approach for answering RPQs on large graphs. Our method exploits the fact that not all labels in a graph are equally frequent. We devise an algorithm which decomposes an RPQ into a series of smaller RPQs using rare labels, i.e., elements of the query with few matches, as waypoints. A search thereby is decomposed into a set of smaller search problems which are tackled in a bi-directional fashion, supported by a set of graph indexes. Comparison of our algorithm with two approaches following the traditional methods for tackling such problems, i.e., the usage of automata, reveals that (a) the automata-based methods are not able to handle large graphs due to the amount of memory they require, and that (b) our algorithm outperforms the automata-based approach, often by orders of magnitude. Another advantage of our algorithm is that it can be parallelized easily.

## 1. INTRODUCTION

A general regular path query (RPQ) is a regular expression $R$ over the (edge or node) labels of a graph $G$ [29]. Its result is the set of all cycle-free paths in $G$ whose concatenation of labels (edge or node) spells out $R$. Different flavors of RPQs are used in a wide range of applications. For instance, XPath supports a restricted form of RPQs on XML documents [26]. SPARQL supports a very simple form of RPQs for RDF graphs, and various proposals exist for enhancing SPARQL syntax with full RPQs (e.g., [5, 6, 10, 22]). [12] describes a restricted form of RPQs important for graph pattern matching, and [34, 35] describe languages supporting restricted forms of RPQs for studying social networks. A particularly important application domain for RPQs are the Life Sciences, where understanding the interactions of different biological entities is of great importance. Such interactions are typically modeled as graphs [4], and RPQs are used to find specific biochemical pathways between distant nodes [24].

None of these works support general RPQs on large graphs, but all focus on restricted languages which typically allow for more efficient evaluation. Evaluating RPQs on arbitrary graphs is an NP-hard problem [29]. We illustrate the problem and our main idea by an example. Suppose a graph of researchers (nodes), either labeled as **P**rofessors or S**T**udents, connected by directed edges such as **S**upervised or **J**oint work. In this graph, the query $P(JP)(JP)?$ finds all paths between a professor and direct or indirect co-workers. $(PS)(PS)+(P|T)$ finds all paths between a professor and his doctorate descendants. Now suppose we also model research prizes as nodes (such as **N**obel Prize or Sigmod **A**ward), and connect them to researchers with edges labeled **H**onored. Then, we can find the doctorate predecessors of all Nobel Prize winners using the query $(PS) + PHN$, and those of any prize winner using the query $(PS) + PH(N|A)$.

RPQs have been studied intensively for XML, where the predominant approach is to use automata [14]. Both the graph and the query are represented as automata, whose intersection automaton is the subgraph specified by the query. In this process, the graph needs to be translated into a DFA, which can be of exponential space and may need exponential construction time. Research in XML query languages has shown that automata-based RPQ evaluation works well for trees (XML) [32], but we will show that its space consumption is enormous on general graphs (see Section 6). Furthermore, automata-based approaches completely disregard the fact that certain labels are much more frequent than others, which can be exploited for speeding up query execution. For instance, to answer the query for Nobel Laureates, it is sensible to first search for nodes labeled with $N$, because (a) one such node must be in any matching path and (b) there are much less Nobel Laureates than professors. Having all $N$ nodes, complete paths can be computed easily by traversing the graph.

Such reasoning is the basis of the approach we propose in this paper; however, finding a good evaluation strategy is not always as easy as in the example, as a query usually contains various labels with different frequencies in the graph. To this end, we first gather all labels in the query that only occur a few times in the graph and use these as fix points for a series of bi-directional searches. Thus, we split the query at these rare labels into smaller queries, and answer these individually. We search all matching paths between each two adjacent rare labels as well as at the start and end of the query, and combine the results to answer the original query. The main advantage of this approach is

that we do not need to consider the whole graph, but only those fractions of it that lie between adjacent rare labels. In Section 6 we will show that this strategy, over a wide range of small to large synthetic and real-life graphs, is considerably faster and especially much less space-demanding than automata-based methods ([41, 15]).

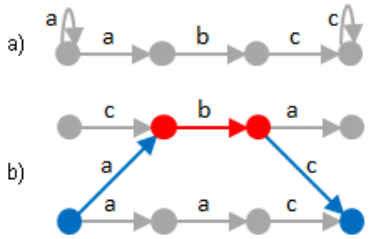


**Figure 1: a) The RPQ $a+\ b\ c+$ shown as a (nondeterministic) automaton; b) a path fulfilling the RPQ in a small exemplary graph.**

Figure 1 illustrates this idea. Suppose we want to answer the RPQ $a+\ b\ c+$ on a graph (Fig. 1b). Since there is only one edge labeled with $b$, we use it as rare label and split the query there. Now, the two smaller queries $a+$ and $c+$ have to be answered, using the $b$ edge as end point or start point, respectively. The result of the original query is the combination of the smaller queries and the rare label.

Our idea can also be used for variations of the general RPQ problem, such as finding all shortest matching paths spelling out a regular expression, or finding all matching paths between two given nodes. Especially the latter is important if RPQs are used as predicates in a general query language (see, for instance, [22]), where variables for defining the start and the end of a path often already have been bound by other predicates before RPQs are evaluated. For space constraints, in this paper we only describe in detail the algorithm for solving the general RPQ problem (finding all acyclic paths matching a given regular expression) without bindings for the start and end nodes, as this is the most complex case (see Section 6.9 for a comparison of runtimes with and without start and end bindings). Furthermore, we only consider RPQs over edge labels; extensions to include node labels are straight-forward.

This paper is structured as follows. Section 2 gives an overview on related work. In Section 3, we define the basic concepts. In Section 4, we describe our novel RPQ evaluation algorithm and give implementation details in Section 5. We evaluate our method in Section 6 using various real and synthetic graphs and conclude in Section 7.

## 2. RELATED WORK

The most common approach for answering RPQs is based on automata. One implementation of this idea are DataGuides by Goldman and Widom [14], based on Lorel [1]. Therein, the graph is considered as an NFA that is first converted into a DFA and then minimized. This minimized DFA (a DataGuide) is then used as an index. However, this index can become much larger than the original graph, which is a problem when dealing with arbitrary graphs (but not for regularly structured XML). Goldman and Widom therefore propose "Approximate DataGuides" [15] which reduce the index size using heuristics. As the implementation of Lorel (and DataGuides) is not supported any more since 2000, we re-implemented the algorithms and compare them to our approach in Section 6.

After the uptake of the semi-structured data model in XML, many proposals have been put forward to use automata in optimizing queries on XML (see e.g. [16, 26, 32]). However, these works mostly use tree automata [32] and are not applicable to arbitrary graphs. Additional index structures have been proposed, for instance, by Milo and Suciu [30] and Kaushnik et al. [21], but, again, they are designed to work with XML data and cannot be applied to non-tree graphs. Fernandez and Suciu present another interesting approach to speed up graph searching based on Graph Schemas [13]; however, these have to be created manually, a step that seems unfeasible for graphs with millions of nodes.

An area where RPQ queries on graphs are important is querying RDF data. However, SPARQL, the official W3C recommendation as an RDF query language, does not support regular path queries, which spurred research into extending SPARQL with RPQs. Alkhateeb et al. [5] developed the query language PSPARQL that includes RPQs, but the authors focus on formal semantics of RPQs on RDF and do only describe a proof-of-concept implementation based on backtracking. Detwiler et al. [10] present the GLEEN system, an extension to SPARQL including RPQs that is implemented as an extension to the ARQ library for SPARQL processing. Anyanwu et al. [6] present another extension to SPARQL that also includes RPQs, but no method for evaluating them is described. Zauner et al. [41] present a path language for RDF supporting RPQs and does provide an implementation; the system is based on automata, and we compare against it in Section 6. SPARQLeR is another RDF querying language encompassing RPQs, again based on automata, for which an implementation was published [22], but is not available anymore (even after multiple requests). Note that the runtimes reported in the latter paper range in the order of seconds for queries with bound start and end nodes on moderately sized graphs, a setting in which our algorithm only needs milliseconds (see Section 6.9).

There were also a number of proposals for general graph query languages that are not based on RDF. Mork et al. [31] propose a query language for semi-structured biological databases. Leser [24] proposes a query language for querying biological pathways. Both languages syntactically support RPQs, but neither of them describes a scalable resolution technique. Graphs-at-a-time is a query language based on graph grammars that is capable of expressing RPQs, but the presented implementation does not cover such predicates [17]. The query language proposed in [11], which also includes a type of RPQs, is not accompanied by any implementation. Several systems have been designed to support extremely large graphs (hundreds of millions nodes and edges), such as Pregel [27], GRAIL [40], or DEX [28], however, none of these systems support RPQs. Finally, Sevon and Eronen [36] describe a method for querying paths in labeled graphs using context-free grammars. They traverse the graph breadth-first and use a context-free parser to find matches. While context-free grammars are more powerful than regular expressions, [36] only focus on finding paths between fix start and end nodes and do not provide optimization techniques as we do.

Due to the high worst-case complexity of RPQs [7], researchers recently started to look into restricted forms of

RPQs which allow more efficient evaluation strategies. Fan et al. [12] show that a language supporting reachability and a restricted form of RPQs allows for an evaluation in cubic time. Jin et al. [20] present algorithms for finding paths which only consist of labels from a predefined set of labels. Ronen and Shmueli [34] describe a graph query language that supports conditions on labels being contained or not contained in paths and also supports ranking (provided that the edges are weighted) and aggregation over sets of paths. In contrast to these works, our approach supports full RPQs.

Matching regular expressions (REs) on strings is a related problem that was recently picked up by the database community. Examples are [8, 9], which both use index structures that are similar in spirit to our method, i.e., concentrating on rare characters in the query. Another line of related research is graph indexing. Here, the idea of using frequencies of labels has been used extensively, especially in mining and searching of subgraphs [38, 23]. The most related work along this line is the distance-join described in [42], describing a method to match subgraphs where edges may be matched to paths of a certain label and of restricted length.

In summary, despite a large body of research around evaluating RPQs on graphs, we are aware of only two available implementations supporting full RPQs as we do [41, 10]. In Section 6, we compare our approach to these methods and also to a re-implementation of the DataGuide system [14] and show that, for large graphs, they suffer from excessive memory consumption and are clearly outperformed by our method.

# 3. TERMS AND DEFINITIONS

We use labeled directed multigraphs, i.e., a graph $G$ is a tuple $G = (V, E, f, l, \Sigma)$, where $V$ is a finite set of nodes, $E$ is a finite set of edges, $l : E \to \Sigma$ specifies the edge labels $\Sigma$, and $f : E \to V \times V$ is the connection function, specifying which nodes are connected by which edges.

The topological properties of a graph can be measured through node degree and label distribution. A graph is called scale-free if the number of nodes with degree $k$ is $P(k) \sim k^{-\lambda}$ for large values of $k$. Scale-free graphs are the likely outcome of various random growth processes, and, indeed, many graphs discovered in biological research are scale-free [25]. The label distribution in a graph is called Zipfian if the frequency of the labels occurring in the graph follows the power law $F(k) \sim k^{-\delta}$, with $\delta \approx 1$.

A regular path query (RPQ) is a regular expression over $\Sigma$. We use the definition for regular expressions as in [18]. To evaluate regular path queries, a regular expression can be converted into an automaton that can be used to match paths. We assume definitions for deterministic (DFA) and non-deterministic automata (NFA) as in [3].

Several kinds of questions can be answered for a given regular expression $R$ and a given graph $G$. (1) Does $G$ contain any path fulfilling $R$? (2) Which is the shortest path in $G$ fulfilling $R$? (3) Is there a path in $G$ between two fixed nodes fulfilling $R$? (4) Which paths in $G$ fulfill $R$? In this paper, we discuss our proposal using the latter problem as show case as it subsumes all other types of queries. In Section 6, we will show that, for instance, fixing the start and end nodes of an RPQ allows to drastically speed-up query processing compared to the more liberal problem of returning all paths in the graph.

# 4. ANSWERING RPQS USING RARE LABELS

In this section, we present our novel algorithm for efficiently answering RPQs. The basic idea is to search the graph while simultaneously advancing in the query automaton. Compared to an approach which converts both the graph and the query into automata, our method has several advantages: No preprocessing of the large graph is needed, it only uses space linear in the size of the graph, the search is easily parallelizable, and it can be enhanced with techniques that take label frequencies into account.

## 4.1 Rare Labels

*Definition 1.* Let $G$ be a graph and $R$ be a RPQ. We call a label occurrence in $R$ *mandatory* iff it occurs in every possible result of $R$ in $G$. We call a label *rare* iff it occurs at most $m$ times in $G$ and is mandatory in $R$ (where $m$ is a parameter of the method, see Section 4.3).

For example, in the regular expression $a\ b{+}\ c{*}\ d?$, $a$ and $b$ are mandatory, while $c$ and $d$ are not. Finding all rare labels in a query is very fast if a list of all labels in the graph together with their frequencies is stored. If this list is indexed by labels, finding all rare labels for a given query is linear to the size of the query pattern.

If a query contains a rare label, then any match of the query in the graph must contain an occurrence of it. Therefore, we can use the occurrences of rare labels as way-points during the search process. If we can find two or more rare labels in a query, we can use a two-way search algorithm to find all matching paths between their matches in the graph. Every additional rare label further reduces the search space.

This idea can be visualized intuitively assuming a graph of randomly distributed nodes in a 2D space, where every node is connected to its $k$ nearest neighbors. In such a graph, the number of nodes that are visited in a breadth-first search correlates with Euclidean distance, i.e., with the size of a circle around a node. In Figure 2a), we assume that a query contained two rare labels. Thus, we perform a two-way search between any occurrences of these labels in a breadth-first manner, during which we visit a number of nodes that correlates with the size of the circles in the figure (reality is more complex, as we also need to search from the rare labels to possible start and end points; furthermore, rare labels usually are not unique). In Figure 2b), we assume a third rare label, which reduces the number of visited

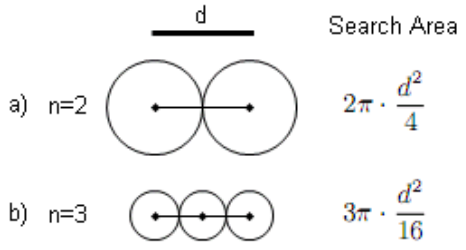


**Figure 2: Illustration of the area that needs to be searched in a two-way search with different numbers $n$ of known way-points (only valid for certain types of graphs, see the explanation in the text).**

nodes by a factor which, for these special graphs, is proportional to the decrease in covered space. In such graphs, the search area for $n$ nodes is $S = \frac{n \cdot \pi}{(n-1)^2} \cdot \frac{d^2}{4}$ and thus shrinks linearly to the number of known nodes between the start and end nodes. Note that this example is only given as illustration and that the formulas are not valid for arbitrary graphs. However, the number of visited nodes always correlates with the distance (in hops) to a known node, and thus the general idea also holds for arbitrary graphs.

Note that our implementation (see Section 5.2) also treats disjunctions (regular expressions of the form $(a|b|...)$) as mandatory by searching for any occurrence of the labels. A further, not yet implemented optimization would be to rewrite expressions of the form $R\ S{*}\ T$, where $R$, $S$, and $T$ are regular expressions, as two expressions $RT$ and $R\ S{+}\ T$. Thus, the original expression could be answered by evaluating two expressions, one of which is shorter and the other contains an additional mandatory label for our optimization.

## 4.2 Searching the Graph using Rare Labels

For queries that include at least one rare label, we split the query at these rare labels and use them as fix points in the search. Searching the graph and advancing in the regular expression at the same time, we search all paths between each two adjacent rare labels, all paths from the first rare label backward to the start of the regular expression, and from the last rare label forward to the end. As shown above, the number of nodes that need to be visited during this search shrinks with every additional rare label but grows with increasing numbers of occurrences of rare labels.

Besides keeping the search space smaller, rare labels often also allow for early stops. If there is no path between any two adjacent rare labels, then there can be no path fulfilling the original query, and the search can be stopped immediately.

In the following, we use the term *first rare nodes* for all nodes that are starting point of an edge of which the label is the first rare label, according to the regular expression. Analogously, *last rare nodes* are the end nodes of all edges with the last rare label. Answering RPQs using rare labels is done in the following 6 steps.

1. Gather all rare labels for the query in the graph.
2. If more than one rare label exists, find the paths between the first and second rare label, the second and third etc. using a two-way search algorithm. If no path can be found in any of these search processes, stop the search and return an empty result for the query.
3. If more than one rare label exists: Using the results from step 2, find all paths from the *first rare nodes* to the *last rare nodes* and remove all rare nodes that are on no path, as these cannot be on a result path.
4. Beginning at all remaining *first rare nodes*, find all paths to the beginning of the regular expression, searching backward.
5. Beginning at all remaining *last rare nodes*, find all paths to the end of the regular expression (forward).
6. Using the results, enumerate all paths in the graph that fulfill the regular expression and return the result.

Figure 3 shows the principle of the algorithm. On a sample graph (edge labels and directions omitted), the query $a{+}\ b\ c{+}\ d\ e{+}$ is executed, assuming that $b$ and $d$ are rare labels. In step 1, rare label edges are gathered ($b$ and $d$ edges). In step 2, we search all paths between the end nodes of the

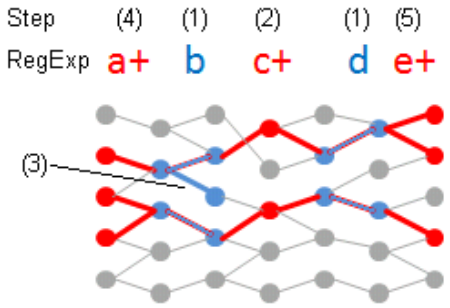


**Figure 3: Search process example for the query $a{+}\ b\ c{+}\ d\ e{+}$ in an arbitrary graph (edge labels and directions omitted).**

$b$ edges and the beginning of the $d$ edges. These paths must fulfill the regular expression between the two rare labels, in this case $c{+}$. Here, two such paths can be found. For one rare edge, no path could be found, thus it is removed from further consideration in step 3.

In step 4, a one-way backward search is performed, starting at the start nodes of the $b$ edges. The search ends once all paths have been found that fulfill the first part of the regular expression ($a{+}$). In step 5, we search all paths from the end of the last rare label to the end of the regular expression in forward direction. As a last step (not shown in the picture), we enumerate all paths by combining the results of the previous steps. In this case, there are 4 distinct paths. The result subgraph can be gathered by enumerating all nodes and edges of the result paths.

Our approach specifically aims at queries that include labels that do not occur often in the graph. While most queries used in Bioinformatics are interested in these rare labels, there are also queries in which no rare label is present. In such cases, our algorithm automatically switches to a brute force search, starting a search at every node in the graph (respectively at the given start/end nodes, if specified). In Section 6 we show that our implementation is faster than other approaches (in particular, the automata-based one) even for those cases.

## 4.3 Determining Rare Labels

The algorithm described above assumes a fixed value for $m$, the parameter determining which labels are considered rare. The choice of $m$ needs to find a compromise between treating as many labels as rare as possible and keeping the number of occurrences of rare labels as small as possible. If a rare label has many occurrences in the graph, the search space increases because each occurrence needs to be included in the search (forward and backward). On the other hand, multiple different rare labels in a query speed up its execution, because partial paths that need to be searched are shorter.

We know of no simple way to determine the best value for $m$. It depends on the graph as well as on the query, so using a fixed value is not the best approach. We therefore use an adaptive heuristic for determining which labels to consider rare for a given graph and query. The idea is that if, for a given query, there are several possible rare labels, we set the threshold for rare labels higher than if only very few

rare labels can be found. This, on the one hand, produces less queries without any rare labels. On the other hand, for queries with many potential rare labels, only labels with a small number of occurrences are included.

Our proposed heuristic works as follows. We first acquire a list of all potential rare labels for the query. We then reduce this list depending on the overall number of paths that would need to be searched in the current configuration. Labels that produce the most paths are removed first; we can compute the number of paths between any two adjacent rare labels $r_1$, $r_2$ as $|r_1| \cdot |r_2|$. The overall number of paths is the sum of all paths between all adjacent pairs. For the first and last rare labels, we also add their number of occurrences to account for the search to the beginning and to the end of the path. We repeatedly remove the rare label that produces the most paths, until the sum of all paths is below a threshold.

# 5. IMPLEMENTATION

In this section, we give a short overview of the data structures and some details about the implementation of the algorithms. Like all other implementations of RPQ-like queries we are aware of (see Section 2), our algorithms are main-memory based. Our current implementation requires about 2 GB of memory for a graph with 10 million nodes and 20 million edges, including all additional data structures described in the following. Thus, working with graphs even much larger than the ones we use for evaluation (see Section 6.1) would, in the first place, not be a problem of memory.

## 5.1 Data Structures

We use a node based storage schema, which means that the nodes in the graph are represented explicitly, while the edges only exist as attributes of the nodes in the form of adjacency lists. Edges are stored in forward and backward direction, which enables two-way searching but almost doubles memory consumption. Labels are always stored as integers; if the labels are given as strings, a global mapping table is used to map them to numbers. This schema needs 28 bytes per node plus 16 bytes per edge to represent the graph. However, due to the storage overhead in Java objects, our implementation needs about 80 bytes per node plus 16 bytes per edge. To be able to efficiently gather rare labels from the graph, we also use an index on the edge labels also encoding their multiplicity.

Regular path queries are represented as NFAs. Converting a regular expression into an NFA is straight-forward. The automaton is stored as states and transitions, which are labeled with the number representing the label for the transition. Transitions are stored in both directions for two-way search. To speed up query execution, we create a list for every state with all labels that are accepted in that state, and a list showing into which states the automaton may transition for each label. We call these *labeled follow sets* as a reference to follow sets used in compiler construction [3].

## 5.2 Search Algorithms

Answering RPQs involves several algorithms. In this section, we give an overview of the most important ones.

Rare Label Search. The first task in executing a query is to find the rare labels. To this end, we go through the automaton representing the query from start to end. For every state, we check if it is mandatory (i.e., does not have a modifier as * or ?). If it is mandatory, we look up the edge label the state represents in the edge label index (which requires constant time). This index gives us the number of occurrences of the label in the graph. If it is below the given threshold, the current state is added to the list of rare states. We consider alternatives (e.g., $a|b$) as rare if the sum of the number of occurrences of all alternatives is below the threshold. Items in brackets can only be mandatory if the bracket itself is mandatory.

Two-way Search. If two or more rare labels have been found, we use a two-way search algorithm to find paths between each two neighboring rare labels. Since rare labels can occur more than once, this is a many-to-many search, starting at the end nodes of all edges with one rare label, and ending at the start nodes of all edges with the next rare label. The search is performed breadth-first by iterating through the graph and the query automaton at the same time. A search state (one specific point during the search process) consists of the current position in the graph and the current state of the automaton. Different search states can be at the same position in the graph but in different states of the automaton, or vice versa. When traversing an edge in the graph, we check if its label is in the *labeled follow set* of the current state. In that case, new entries are added to the end of the list of search states to be processed. One search state is created for each entry in the follow set.

The aim of the search is to find all paths for each pair of fix start and end nodes. A path is found if a forward and a backward search meet at a node and are in the same state of the query automaton. We keep lists for every node where we store in which states a search passed the node. This is used to find completed paths as well as to prevent cycles in the result paths. The search ends once the list of unprocessed search states is empty.

Start and End Search. Searching for the start and end of the path works much like the two-way search algorithm. The only differences are that we search one-way and that we do not end when finding specific nodes, but when hitting a finish state in the query automaton (or a start state, for the backward search).

## 5.3 Two-Way Search Complexity

Theoretically, all nodes might need to be searched during a two-way search, and every node could be visited in every state of the automaton, and from every start or end node, resulting in a complexity of $O(|V| \cdot |S| \cdot r)$, with $S$ being the states of the automaton and $r$ the number of start plus end nodes, i.e., the occurrences of the rare labels used for that search. In reality, however, the search space is limited because the labels on the path must match the automaton (a query should thus be as small as possible, e.g. not include $a*a*$ instead of $a*$). Also, the number of start and end nodes is much smaller than $|V|$ (depending on which labels are considered rare). Due to two-way search, the search space is reduced further in most cases, since complete paths are only half as long from both directions.

Additional checks have to be made during the search, but they do not add to its complexity. For finish and cycle checks, we need to check for all nodes encountered on the path whether they have already been visited in the same state in the same direction (indicating a circle) or in the other direction (indicating a completed path). This can be done in constant time. However, cycle checks might be performed more than once per node, if a node has multiple in-

coming edges: In the worst case, the check is performed as often as the number of edges in the graph. Thus, the overall worst-case complexity sums up to $O((|V| + |E|) \cdot |S| \cdot r)$.

## 5.4 Parallelization

The search process can be parallelized in different ways. For a query that contains $n$ rare labels, $n+1$ smaller queries have to be answered. This is performed as $n+1$ independent searches which are executed in parallel. Also, the search algorithm is a many-to-many search if the rare labels occur more than once in the graph. This can also run in parallel, with different threads processing different start points.

# 6. EXPERIMENTAL RESULTS

In this section, we present an experimental evaluation of our method. We compare our rare-label based algorithm with other implementations available and show results for different graphs and different kinds of queries. Further experiments are devoted to scalability with regard to the size and density of graphs, to the effects of parallelization, different query types, and different label distributions. All tests were executed on a Quad-Core AMD Opteron machine with 16 GB of main memory. The execution times for queries given in the following were gathered by executing 10,000 queries (see Section 6.2) and building the average.

As threshold for the number of path combinations in the rare label optimization (see Section 4.3), we used a value of 100; higher values did not yield any significant changes in runtimes, while lower values lead to slower queries.

## 6.1 Graphs and Queries

We use real graphs (from biological research) as well as artificially created graphs for the evaluation. We present results for two real-world graphs which we call *AliBaba* and *Extracts*. *AliBaba* is a network of protein-protein-interactions extracted by text mining on all of PubMed [33]. The graph has about 50,000 nodes and 340,000 edges. *Extracts* is a graph of enzymes and their relations, also extracted by text mining from biomedical abstracts, containing about 80,000 nodes and one million edges. Note that these graphs are not toys; such networks today are used regularly in Systems Biology, for instance to improve protein function prediction [19] or disease-gene identification [2]. Their size is roughly comparable or larger to that of the largest databases of biological networks (the KEGG network currently contains approximately 45.000 nodes), but their density is considerably higher. Thus, they represent rather difficult cases for this domain. Results for other biological networks we tested were similar to those on these two graphs.

To systematically study the scalability of the algorithms with regard to various parameters, we created artificial graphs with sizes between 1000 and one million nodes, with different average degrees and different label distributions. All real and all synthetic graphs are scale-free and roughly have a Zipfian distribution of edge label frequencies. The influence of the type of graph and the label distribution is shown in Section 6.6.

## 6.2 Query Sets

For our evaluation, we used both real-life queries from the Bioinformatics domain as well as large, randomly generated sets of queries with similar properties.

For most tests, we used synthetically generated sets of 10,000 queries that were created with the following properties. The length of a query is chosen randomly with certain probabilities. The shortest query consists of three labels and has the highest probability of occurring (15%). The probability decreases to the longest possible query of 10 labels (5%). Queries containing multiple brackets can be up to 20 labels long. The average query length is about 5.

Most labels are connected by concatenation. Brackets are used with a probability of 5% for every label. Alternations only occur in brackets, with a probability of 10%. Modifiers (such as $+$, `*` and `?`) are also distributed randomly on the labels and brackets. Each label or bracket has a 30% probability of having no modifier, 30% each of having a $+$ or `*`, and 10% for `?`.

For query sets that do not use rare labels, labels are chosen randomly. Labels that occur more often in the graph are used more often in the queries, with the same degree. For query sets with rare labels, we ensure that at least one rare label is present in the query. Different rare query sets use different minimum numbers of rare labels, and different numbers of occurrences of the labels to be considered rare. The rare labels do not appear in brackets and have no modifiers except possibly `+`, so that they can be used for our optimizations.

## 6.3 Biological Queries

In addition to these synthetic queries, we evaluated our method using 12 biological queries from a real-life application. Recall that the AliBaba graph has been created by extracting different types of relations (generally called interactions) between proteins from a selection of abstracts available in PubMed. A very important question in Systems Biology is how sets of such single interactions act together to achieve and control a certain biological function, such as formation of protein complexes or concerted gene regulation [39]. Such groups of interactions are called network motifs, and a large class of these motifs can be described using regular path queries. As an example, researchers are interested in finding chains of interacting proteins that result in the acetylation and then up-regulation of other proteins. As an "interaction" can be expressed by many words in a paper, finding such motifs requires both unions of labels (for finding everything that is indicative for an interaction) and their concatenation in a regular expression. Other query examples are methylation that results in up- or down-regulation, receptors that increase or decrease phosphorylation, or fusions of interactions and phosphorylation. We omit the concrete queries here for brevity; they are available on request.

Table 1 shows result sizes and runtimes for the biological queries executed on the AliBaba graph. Eight of these queries contain a rare label and can thus benefit from our optimization, while the others are answered using the brute force approach. Clearly, all queries with rare labels can be answered significantly faster than with the brute force algorithm. In turn, times for answering the queries with the automata-based method are between 2 and 6 times longer than with our method. Note that all queries, even those which generate almost 100,000 different paths, can be answered in less than one second.

## 6.4 Comparing with Other Implementations

We are aware of just two implementations of regular path

| Query | Rare Label | Result Paths | Run Time Rare | Run Time Brute |
|---|---|---|---|---|
| 1 | acetylation | 1710 | 62 ms | 458 ms |
| 2 | acetylation | 20 | 4 ms | 196 ms |
| 3 | methylation | – | 95 ms | 211 ms |
| 4 | methylation | – | 4 ms | 164 ms |
| 5 | fusions | – | 1 ms | 122 ms |
| 6 | fusions | 8 | 3 ms | 417 ms |
| 7 | receptor | – | 9 ms | 330 ms |
| 8 | receptor | – | 2 ms | 100 ms |
| 9 | – | 80905 | – | 796 ms |
| 10 | – | 2118 | – | 170 ms |
| 11 | – | 249 | – | 250 ms |
| 12 | – | 49638 | – | 823 ms |

**Table 1: Rare labels, number of result paths, average execution times with our rare label algorithm and with the brute force baseline for biological queries on the AliBaba graph (using 4 processors).**

queries on graphs and considered both as competitors of our algorithm.

RPL [41] evaluates RPQs on RDF graphs using an automata-based implementation. Using the original code provided by the authors, we observed that the system works well on small RDF graphs. However, it cannot handle graphs of the size we target with our work. For answering queries on a graph with approx. 10,000 nodes and 20,000 edges, the system already required 16 GB of main memory. Queries on this graph required already several seconds per query, while our algorithm answers those queries in less than 100ms. All larger graphs, including our real-world biological graphs, could not be handled anymore. Therefore, we do not include this system in the following systematic evaluation.

GLEEN [10] is implemented as an extension to the ARQ library for SPARQL processing. Thus, a comparison of runtimes would be rather unfair, as the ARQ extension mechanism is based on an iterative load-and-verify of single edges of a graph, which cannot be compared to our approach of loading the entire graph into memory at once.

This leaves the re-implementation of the DataGuides system [15] (called AUT from now on) to compare with. For comparing our RL method with AUT, we use sets of queries with rare labels as well as completely random queries (which may contain an arbitrary number of rare labels or none). As we are not aware of any efficient parallelization scheme for automaton minimization and determinization, we only compare single-threaded versions of both implementations; the additional speed-up that is possible with our algorithm on current (multi-core) hardware will be evaluated in Section 6.8.

Figures 4 and 5 show the average runtime (averaged over 10,000 queries) for answering an RPQ with RL and with AUT. The former is faster in all cases. For queries without rare labels, the runtimes do not differ much for small graphs, but differences get significant for larger graphs. For a graph of 10,000 nodes and 20,000 edges, RL already performs almost two orders of magnitude faster than AUT. For queries that contain at least one rare label, again, RL is always faster, and its superiority increases with graph size; differences for those queries are larger than for queries without rare labels.

Comparing Figures 4 and 5, one can see that rare labels have a considerable influence on the runtime of RL.

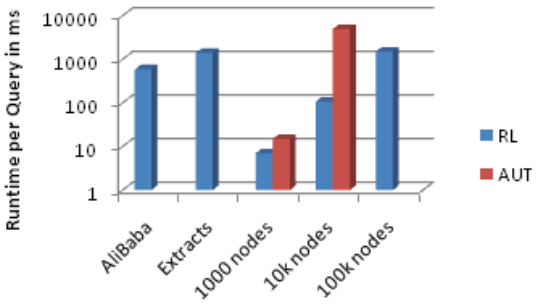


**Figure 4: Average runtime (log scale) to answer one query without rare labels on different graphs. Queries on Extracts as well as on the synthetic graph with 100K nodes could not be executed with AUT.**

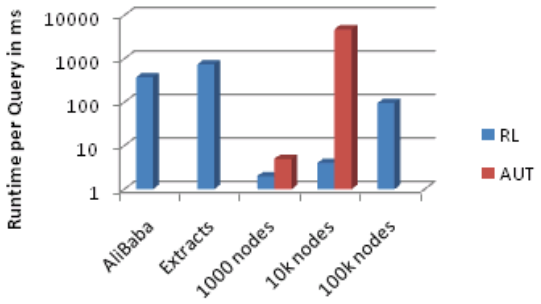


**Figure 5: Same as Figure 4, but for queries containing at least one rare label.**

In contrast, we found that for AUT, the runtime for different queries on the same graph is about equal. However, time is only one problem of this method; the other is space. The NFA-DFA conversion in AUT incurs an exponential increase in the number of nodes. In our implementation, the whole process requires 350 MB for the graph with 1000 nodes and 2000 edges, but already 3.8 GB for the 10 times larger graph – which still is considerably less space than required by RPL (see above). Running AUT on the AliBaba or Extracts real-life graphs or on the 100K synthetic graph failed due to memory overflow.

## 6.5 Scalability: Graph Size and Density

To test different scalability aspects of RL, we used artificially created graphs with varying properties. All queries contain at least one rare label. For this evaluation, we used the multi-threaded implementation with a fixed number of four threads. We evaluated the effect of our rare-label optimization by comparing it to a baseline method, which performs a brute-force search starting at every node (also in parallel) without considering label frequencies.

Figure 6 shows how RL scales with the size of the graph at a fixed node/edge ratio. The smallest graph has 10K nodes and 20K edges, and the largest graph has 1 million nodes and 2 million edges. Clearly, the scaling of the implementation with the rare-label optimization is much better than that of the baseline. Even for the largest graphs we tested, RL can answer a regular path query in few seconds on average.

Figure 7 shows scalability of RL with the average graph

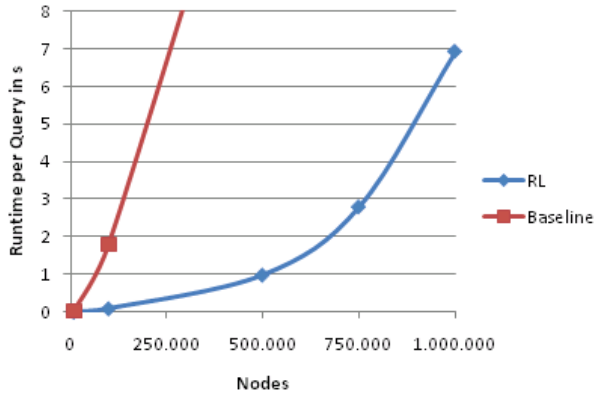


**Figure 6: Average runtime for answering one query on synthetic graphs with different numbers of nodes and a fixed node to edge ratio of 1:2.**

degree. Using multiple artificial graphs with 100,000 nodes, we increased the number of edges (and thus the average degree), leaving all other properties equal. Again, the increase in execution time is favorable compared to the baseline.

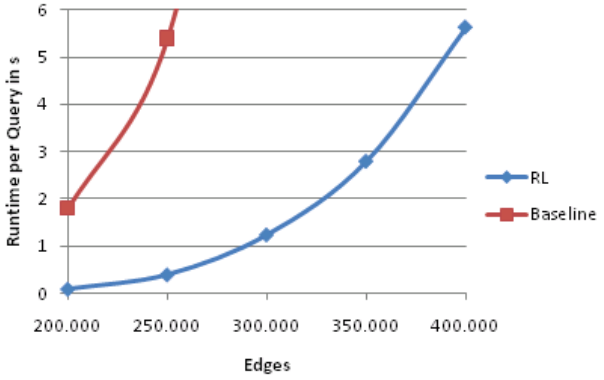


**Figure 7: Average runtime for answering one query on synthetic graphs with 100,000 nodes and different numbers of edges.**

Both experiments also show that execution times grow super-linearly with increasing graph size and graph density. Also, the absolute times cannot be compared to those achieved for answering, for instance, reachability queries on graphs of similar size [40, 37]. But one should not forget that evaluating RPQs on graphs is a NP-hard problem, whereas reachability can be answered in $O(n^3)$.

## 6.6 Skew in Distribution of Label Frequencies

To show the influence of the distribution of label frequencies in the graph, we created several graphs of the same size with differently distributed labels. To this end, we raised the exponent in the power law the labels are created with, and as a result the total number of labels in the graph grows, creating more – and more rare – rare labels. Figure 8 shows the results for different label distributions. As can be seen, the impact of the distribution is comparably small over a wide range of distributions; only queries on graphs with few distinct labels are answered significantly slower. In those graphs, there are hardly any rare labels; furthermore, the number of matching paths on average increases steeply with decreasing numbers of labels, as the probability for a path to match an RPQ increases by pure chance.

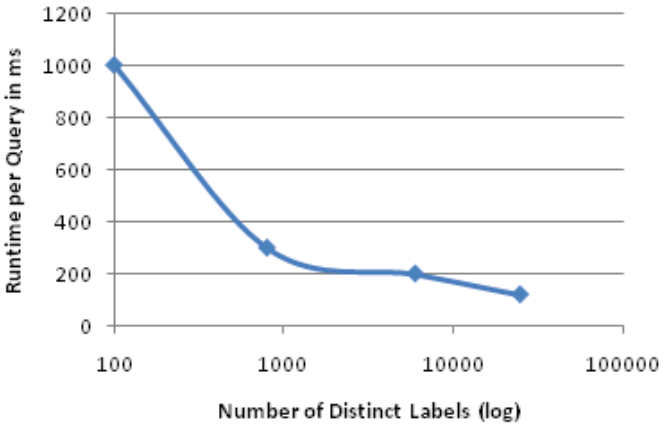


**Figure 8: Average runtime for answering one query on a synthetic graph with 100,000 nodes and 200,000 edges and different edge label distributions.**

## 6.7 Influence of Query Types

To evaluate the influence of our optimizations on different types of queries, we created ten sets of 1000 queries each differing in the number of occurrences of rare labels. One set does not contain any rare labels (set 0). All other sets contain exactly one rare label, but with an increasing number of occurrences in the graph. The rare labels found in query set 1 appear only once in the graph, rare labels from set 2 exactly twice, and so on. The runtimes for the different query sets are shown in Figure 9.

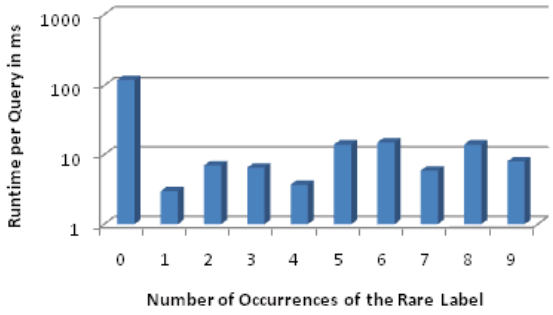


**Figure 9: Average runtime for answering one query from the respective query set. The synthetic graph that was used has 10,000 nodes and 20,000 edges.**

As expected, answering queries without rare labels is considerably slower than for queries containing rare labels. The difference between queries without rare labels and those with one rare label appearing exactly once in the entire graph is almost three orders of magnitude. Also, queries with a rare label that occurs less frequently in the graph generally are executed faster. However, this trend is partly out-weighed by noise generated through the random complexity of the queries in the workload.

To further study how runtimes change for different queries, we categorized the runtimes of all queries from our query set on a given graph. As Figure 10 shows, more than 90% of all queries are answered in less than 10ms, and more than 95% are answered in less than 100ms. But a few queries take exceptionally longer than all others. These pathological queries are queries that contain only very frequent labels, leading to exceptionally large result sets. For instance, the most difficult query from our set took about 4 seconds and generated a result set of 230 million matching paths. However, we believe that such queries rarely occur in any real application.

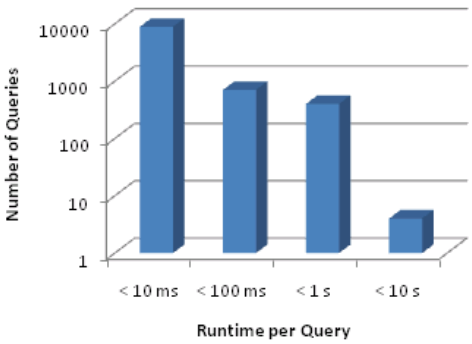


**Figure 10: Number of queries taking up to 10 ms, 100ms, 1s, and 10s to execute (logarithmic scale). The set of 10,000 queries was evaluated on a graph with 10,000 nodes and 20,000 edges.**

We also investigated the influence of query lengths, i.e., the number of lables in the regular expression of the query. Comparing the execution times of queries depending on their lengths, we found that the query length only has a minor influence on execution speed (data not shown). The reason is that, although longer queries are more complicated to answer in general, they often do not have as many matches in the graph which reduces the search space. For execution speed, the number of occurrences of the labels and the number of rare labels are much more important than the sheer number of labels in the query.

## 6.8 Parallelization

Figure 11 shows the effect of using multiple threads for RL. The scale-up is very good for up to four threads, but the additional advantage of adding more threads levels out for more than 4 threads. This behavior can be explained by the fact that our current implementation uses additional threads only for additional rare labels. For example, a query containing two rare labels uses three threads: One for searching the paths between the rare labels, one for searching to the start and one for searching to the end of the query. Since few queries contain more than three rare labels, the scaling diminishes. However, it would be possible to enhance the implementation by also running the searches between two instances of a rare label in parallel. Since most rare labels appear more than once in the graph, finding paths between them is a many-to-many search and could use different threads starting at different nodes.

## 6.9 Search with Fixed Start and End Nodes

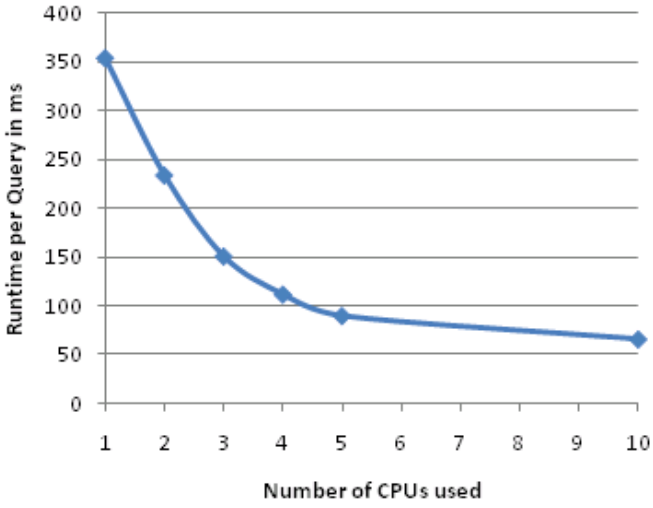


**Figure 11: Average runtime for answering a query containing rare labels on a graph with 10,000 nodes and 20,000 edges against the number of threads used.**

An important subclass of RPQs are those where the start and end node of a query are fixed, i.e., queries that search for all matching paths between two given nodes. As mentioned in Section 2, such queries often appear when RPQs are used as predicates in graph query languages. We performed tests using artificial graphs of different sizes and with different node degrees. We used the same set of 10,000 queries as before, but this time always added randomly chosen start and end nodes. Figure 12 shows the results for graphs of different sizes with a node to edge ratio of 1:2, while Figure 13 shows results for varying graph densities.

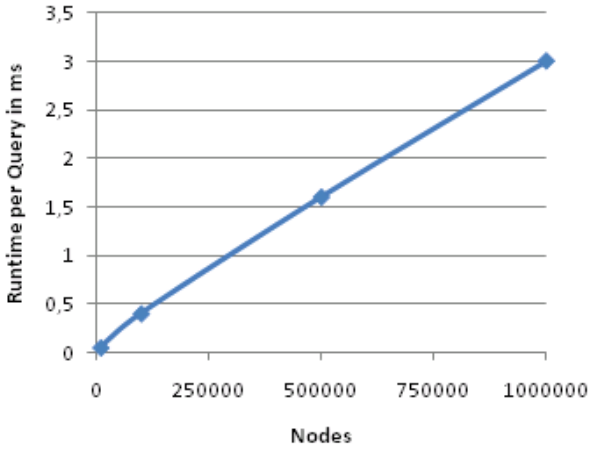


**Figure 12: Average runtime for one query with fixed start and end nodes on graphs with different sizes and a node to edge ratio of 1:2.**

Clearly, specifying start and end nodes has a tremendous effect on runtimes. Queries are answered up to three orders of magnitude faster when compared to the unbound case (see Figure 6), and runtimes grow only linear in the size of the graph. The reason is that fixing start and end nodes

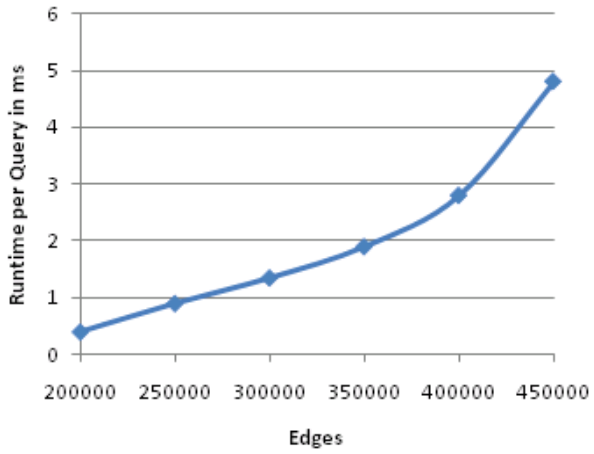


**Figure 13: Average runtime for one query with fixed start and end nodes on graphs with 100,000 nodes and different numbers of edges.**

implies that only a very small part of the graph needs to be searched. The same effect can be observed for graphs with growing density (compare to Figure 7), though the increase in runtime remains super-linear.

Searching paths between two given nodes also is the type of query analyzed in SPARQLer [22], the only other work we know of that gives a quantitative evaluation of RPQs. As our method, SPARQLer is main-memory based. Since no code is available (we contacted the authors without success), we can only compare the numbers as given in their paper. [22] reports runtimes in the range of seconds for graphs of similar size where our algorithm only takes a few milliseconds. However, these numbers are not directly comparable as they were measured with different graphs and machines.

## 7. CONCLUSION

We presented a novel approach for answering regular path queries on large graphs. Our main idea is to structure a graph traversal along those labels from a query that are infrequent in the graph, but guaranteed to occur in any matching path. We use these rare labels as start-, end-, and way-points during traversal, thus essentially breaking up a very large search space into many much smaller ones. We compare our novel method with a traditional approach using automata and find that the former outperforms the latter over a wide range of different graphs and queries; furthermore, it requires only linear preprocessing and is able to handle much larger graphs. We also showed that using the rare-label optimization considerably improves scalability with regard to the size of the graph and to graph density.

## 8. REFERENCES


[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *Int. Journal on Digital Libraries*, 1:68–88, 1997.

[2] S. Aerts, D. Lambrechts, S. Maity, P. Van Loo, et al. Gene prioritization through genomic data fusion. *Nat Biotechnol*, 24(5):537–44, 2006.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1986.

[4] L. Alberghina and H. V. Westerhoff, editors. *Systems Biology: Definitions and Perspectives*. Springer, 2005.

[5] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semant.*, 7(2):57–73, 2009.

[6] K. Anyanwu, A. Maduko, and A. Sheth. Sparq2l: towards support for subgraph extraction queries in rdf databases. In *WWW '07*, pages 797–806, 2007.

[7] P. Barcelo, C. Hurtado, L. Libkin, and P. Wood. Expressive languages for path queries over graph-structured data. PODS '10, pages 3–14, New York, USA, 2010.

[8] C.-Y. Chan, M. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003.

[9] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. *ICDE'02*, 0:0419, 2002.

[10] L. T. Detwiler, D. Suciu, and J. F. Brinkley. Regular paths in sparql: Querying the nci thesaurus. *American Medical Informatics Association*, pages 161–165, 2008.

[11] A. Dries, S. Nijssen, and L. De Raedt. A query language for analyzing networks. In *CIKM '09*, pages 485–494, New York, NY, USA, 2009.

[12] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.

[13] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE '98*, pages 14–23, Washington, DC, USA, 1998. IEEE.

[14] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB '97*, pages 436–445, 1997.

[15] R. Goldman and J. Widom. Approximate dataguides. In *Workshop on Query Processing*, 1999.

[16] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[17] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD '08*, pages 405–418, New York, USA, 2008.

[18] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[19] S. Jaeger, S. Gaudan, U. Leser, and D. Rebholz-Schuhmann. Integrating protein-protein interactions and text mining for protein function prediction. *BMC Bioinformatics*, 9(Suppl8):S2, 2008.

[20] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 123–134, New York, NY, USA, 2010.

[21] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD Conference*, pages 133–144, 2002.

[22] K. J. Kochut and M. Janik. Sparqler: Extended sparql for semantic association discovery. In *ESWC '07*, pages 145–159, Berlin, Heidelberg, 2007.

[23] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1038–1051, 2004.

[24] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, 2005.

[25] L. Li, D. Alderson, R. Tanaka, J. C. Doyle, and W. Willinger. Towards a theory of scale-free graphs: Definition, properties, and implications (ext. version). *Internet Mathematics*, 2(4):431–523, Mar. 2006.

[26] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB 2001*, pages 361–370, Roma, Italy, 2001.

[27] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *PODC '09*, pages 6–6, New York, NY, USA, 2009.

[28] Martínez-Bazan et al. Dex: high-performance exploration on large graphs for information retrieval. In *CIKM '07*, pages 573–582, New York, NY, USA, 2007.

[29] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[30] T. Milo and D. Suciu. Index structures for path expressions. In *International Conference on Database Theory*, pages 277–295, 1999.

[31] P. Mork, R. Shaker, A. Halevy, and P. Tarczy-Hornoch. Pql: a declarative query language over dynamic biological schemata. In *Annual Symp. of the American Medical*, pages 533–537, 2002.

[32] F. Neven. Automata theory for xml researchers. *SIGMOD Rec.*, 31(3):39–46, 2002.

[33] P. Palaga, L. Nguyen, U. Leser, and J. Hakenberg. High-performance information extraction with alibaba. In *EDBT '09*, pages 1140–1143, New York, USA, 2009.

[34] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE 2009*, pages 1595–1602, Shanghai, China, 2009.

[35] M. San Martín and C. Gutierrez. Representing, querying and transforming social networks with RDF/SPARQL. In *ESWC 2009*, pages 293–307, 2009.

[36] P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5(2):100, 2008.

[37] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07*, pages 845–856, New York, NY, USA, 2007.

[38] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD '04*, pages 335–346, New York, NY, USA, 2004.

[39] E. Yeger-Lotem, S. Sattath, N. Kashtan, et al. Network motifs in integrated cellular networks of transcription-regulation and protein-protein interaction. *Proc Natl Acad Sci USA*, 101(16):5934–9, 2004.

[40] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. In *VLDB '2010*. VLDB Endowment, 2010.

[41] H. Zauner, B. Linse, T. Furche, and F. Bry. A rpl through rdf: Expressive navigation in rdf graphs. In *RR 2010*, pages 251–257, 2010. Demo.

[42] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.