ABSTRACT
We demonstrate the use of localized path indexes in generating efficient execution plans for regular path queries. This study is motivated by both the practicality of this class of queries and by the current dearth of scalable solutions for their evaluation. Our proposed solution leverages widely available relational database technology and is often orders of magnitude faster than currently known approaches. We aim in this hands-on demonstration to both highlight the promise of our approach and to stimulate further discussion and study of engineering solutions for this practical yet challenging class of graph queries.

1. INTRODUCTION
Massive graph-structured data collections are ubiquitous in contemporary data management scenarios such as social networks, linked open data, and chemical compound databases. A fundamental paradigm in graph query languages is the so-called regular path queries (RPQs) \[16\]. RPQs specify a regular expression over the edge labels in a graph, and the query answer consists of every path in the graph such that the sequence of edge labels along the path forms a word in the language recognized by the regular expression. Variations and extensions of RPQs are supported in recent query languages such as SPARQL 1.1 \[8\] and the Cypher language of the Neo4j graph database.\[1\]

State of the art. Indexing of paths occurring in data has been shown to be effective for query processing in the context of object-oriented and semi-structured databases \[1\] \[15\]. To our knowledge, however, there has been no investigation of using path indexing for the evaluation of RPQs over graph databases. In particular, three general approaches to RPQ evaluation have been proposed in the literature:

1. Automata- and search-based processing (e.g., \[5\] \[10\] \[13\]), where queries are evaluated by strategies such as breadth-first-search pattern matching of the query graph on the data graph;
2. Datalog-based processing (e.g., \[3\] \[17\]), where the Kleene star operator is translated into recursive Datalog programs or recursive SQL views;
3. Reachability-index-based processing (e.g., \[6\]), where restricted uses of Kleene star are translated into reachability queries, which are then evaluated using off-the-shelf reachability indexes.

Contributions. We present an overview of our ongoing study of the use of path indexes for generating efficient execution plans for RPQs. Our approach supports the evaluation of arbitrary RPQs (unlike approach (3) above), and exhibits significant improvement (often by several orders of magnitude) in query processing times over approaches (1) and (2).

In the next section we briefly define the problem. In Section 2 we introduce the main data structures used in our solution. We then discuss query plan generation and execution in Section 3. We conclude in Section 4 with an overview of our system demonstration.

2. PRELIMINARIES
2.1 Data Model
We consider finite, directed, edge-labeled graphs. A graph vocabulary is a finite non-empty set \( \mathcal{L} \) of edge labels drawn from some universe of labels. Let \( N \) be an infinite universe of atomic data objects. An edge relation is a finite subset of \( N \times N \). A graph over vocabulary \( \mathcal{L} \) is an assignment \( G \) of an edge relation \( \ell^G \) to each \( \ell \in \mathcal{L} \), i.e., there is an edge labeled \( \ell \) from \( m \) to \( n \) if \( (m, n) \in \ell^G \). In the sequel, we will sometimes denote this by \( \ell(m, n) \) or \( m \xrightarrow{\ell} n \). As an example, a graph \( G_{ex} \) over the vocabulary \{supervisor, knows, worksFor\} is shown in Figure 1.

The node set of \( G \) is the collection nodes \( (G) = \{n \mid \exists m \in N, \ell \in \mathcal{L} : (n, m) \in \ell^G \text{ or } (m, n) \in \ell^G \} \). For example, the node set of the example graph contains nine elements: \( \text{nodes}(G_{ex}) = \{ \text{ada}, \text{jan}, \ldots, \text{zoe} \} \).

Let \( k \) be a natural number. If \( k = 0 \), we say there is a \( k \)-path from \( s \) to \( t \), for every \( s \in \text{nodes}(G) \). If \( k > 0 \), for \( s, t \in \text{nodes}(G) \), we say there is a \( k \)-path from \( s \) to \( t \) if there exist \( n_0, \ldots, n_k \in \text{nodes}(G) \) and edge labels \( \ell_1, \ldots, \ell_k \in \mathcal{L} \) such that \( n_0 = s \), \( n_k = t \), and, for \( 0 < i \leq k \), \( (n_{i-1}, n_i) \in \ell_i^G \) or \( (n_i, n_{i-1}) \in \ell_i^G \).
paths is generated by the grammar from regular expression.

2.2 Regular Path Queries

Fix a vocabulary \( \mathcal{L} \). A regular path query (RPQ) is a regular expression \( R \) over the alphabet \( \{\ell, \ell^- | \ell \in \mathcal{L}\} \), i.e., \( R \) is generated by the grammar

\[
R ::= \epsilon | \ell | \ell^- | R \circ R | R \cup R | R^+ | R^* | R^\circ | R^\cup | R^\circ\cup | R^\cup\circ
\]

for \( \ell \in \mathcal{L} \) and natural numbers \( i \) and \( j \) where \( i \leq j \). Intuitively, \( \epsilon \) is the identity transition, \( \ell^- \) is forward navigation along an edge with that label, \( \ell^+ \) is backwards navigation, \( \circ \) is path composition, \( \cup \) is path disjunction (i.e., union of paths), and \( R^+ \) is bounded path recursion.

Given graph \( G \) over \( \mathcal{L} \), the semantics of evaluating an RPQ \( R \) on \( G \) is a set \( R(G) \) consisting of all pairs \( (a,b) \) in \( \text{paths}(G) \) such that there exists a path from node \( a \) to node \( b \) in \( G \) whose label sequence \( \ell_1 \cdots \ell_n \) defines a word in the regular language specified by \( R \).

As examples, we have in the graph \( G_{ex} \) of Figure 1 that:

\[
\text{supervisor} \circ \text{worksFor}^- (G_{ex}) = \{ (\text{kim}, \text{sue}) \}
\]

and

\[
(\text{supervisor} \cup \text{worksFor} \cup \text{worksFor}^-)^4 (G_{ex}) = \{ (\text{kim}, \text{kim}), (\text{kim}, \text{sue}), (\text{sue}, \text{kim}), (\text{sue}, \text{sue}), \}
\]

\[
(\text{ada}, \text{zoe}), (\text{ada}, \text{ada}), (\text{zoe}, \text{ada}) \}.
\]

Note that we deviate from the traditional syntax of regular expressions by replacing Kleene star \( "^*" \) with bounded recursion. This is motivated by the following two observations. First, bounded recursion is supported (and encouraged) in practical graph query languages such as Neo4j’s Cypher\[\footnote{\url{http://neo4j.com/docs/stable/}}\].

Second, since we focus in this work on index construction and use, we are interested in query evaluation on a given graph. It is easy to establish that for any graph \( G \) there exists a natural number \( n(G) \) such that for every RPQ \( R \) it is the case that \( R^n (G) = R^{n+1} (G) \).

3. INDEXES AND HISTOGRAMS

Given a graph \( G \) and a fixed \( k > 0 \), we now present our indexing and selectivity estimation approaches for paths in \( G \) localized to neighborhoods of size \( k \).

3.1 \( k \)-path indexing

A label path is a sequence \( p = \ell_1 \cdots \ell_n \), where \( n > 0 \) is the length of \( p \), and \( \ell_i \in \{\ell, \ell^- | \ell \in \mathcal{L}\} \) for each \( 1 \leq i \leq n \).

Our index on \( G \) is based on an ordered dictionary (which can be implemented, for example, as a B+tree). In particular, we index \( \text{paths}_k (G) \) using an ordered \( k \)-path index \( I_{G,k} \) having search key \( \langle \text{label path}, \text{sourceID}, \text{targetID} \rangle \). Specifically, for each label path \( p \) of length at most \( k \), and for each pair of nodes \( (a,b) \in p(G) \), we insert \( (p,a,b) \) into \( I_{G,k} \).

Given a non-empty prefix \( p \) of a search key, \( I_{G,k} \) returns an ordered list \( I_{G,k}(p) \) of all matching entries.

**Example 3.1.** In the graph of Figure 1, we have

\[
I_{G,k}((\text{knows} \cdot \text{knows} \cdot \text{worksFor})) =
\]

\[
\langle (\text{ada}, \text{tim}), (\text{jan}, \text{ada}), (\text{jan}, \text{jan}), (\text{jan}, \text{kim}), (\text{kim}, \text{ada}), (\text{tim}, \text{jan}), (\text{tim}, \text{kim}), (\text{tim}, \text{tim}) \rangle.
\]

\[
I_{G,k}((\text{knows} \cdot \text{knows} \cdot \text{worksFor} \cdot \text{jan})) =
\]

\[
\langle (\text{ada}, \text{jan}), (\text{kim}) \rangle,
\]

\[
I_{G,k}((\text{knows} \cdot \text{knows} \cdot \text{worksFor} \cdot \text{jan} \cdot \text{ada})) =
\]

\[
\langle \rangle,
\]

\[
I_{G,k}((\text{knows} \cdot \text{knows} \cdot \text{worksFor} \cdot \text{jan} \cdot \text{ada} \cdot \text{joe})) =
\]

\[
\langle \rangle.
\]

We have developed a prototype \( k \)-path index implementation that leverages the B+ tree index support of Postgres\[\footnote{\url{http://www.postgresql.org}}\]. We translate RPQs into equivalent SQL statements over \( I_{G,k} \) implemented as a relational table and backed by a B+tree (see \[\cite{12} \] for full implementation details). In building on mature relational technologies, we are following an emerging trend in this direction \[\cite{4,7,9} \] with practical benefits such as simplicity, ease of integration with and deployment within existing IT ecosystems, and leveraging field-proven technologies.

Notwithstanding the fact that we have described here an implementation of our proposed \( k \)-path indexing technique
using existing RDBMS technologies, other recent work [14] describes an implementation of a B+tree-based k-path index “from scratch”, focusing on issues such as index size, compression and performance, and undertaking a comparative performance study with the Neo4j graph DBMS over several real and synthetic datasets and query workloads. That evaluation too demonstrates the potential of our k-path indexing approach (showing speed-ups in query evaluation times ranging from 2 times to 8,000 times faster compared with Neo4j). Detailed performance comparison between these approaches to path indexing is an area of future work.

3.2 k-path histogram

For query plan generation over \( I_{G,k} \), it is useful to have a data structure \( sel_{G,k} \) which, given a label path \( p \) of length at most \( k \), returns an estimate of the selectivity of \( p \) in \( G \), i.e., the fraction of paths in \( paths_{G}(G) \) which satisfy \( p \).

As an example, we have in the graph \( G_{ex} \) of Figure 1 that \( sel_{G_{ex},2}(\text{supervisor} \circ \text{knows}) \approx 0.02 \), since only one of the 53 paths in \( paths_{2}(G_{ex}) \) is in \( \text{supervisor} \circ \text{knows}(G_{ex}) \).

There is a rich literature on statistics for query optimization [2]. Here, we adopt the well-established histogram data structure, since it is easy to deploy and extremely successful in practice. In particular, we implement \( sel_{G,k} \) as an equi-depth histogram. As with the path indexes, in our prototype implementation we store and access our histogram as a PostgreSQL table; see [12] for full implementation details.

4. QUERY EVALUATION WITH PATH INDEXES

The processing of a RPQ \( R \) proceeds in three steps: The first step is to replace each occurrence of bounded recursion in \( R \) as a union over its expansion. The result is a semantically equivalent query \( R’ \) involving only edge labels or their inverses, compositions, and unions. In the second step, all unions in \( R’ \) are “pulled up” to the top level of the query, resulting in a semantically equivalent query \( R’’ \) consisting of a union of expressions each free of unions and bounded recursion, i.e., \( R’’ = R_{1} \cup \cdots \cup R_{n} \), where each \( R_{i} \) is a label path. In the third step, each disjunct \( R_{i} \) is processed in turn, with the aim of generating a physical execution plan for each \( R_{i} \) in which a merge-join is used whenever possible (to make the best use of the physical sort order of the index) and a hash-join is used otherwise.

As an illustrative example, consider the query \( R = k \circ (k \circ w)^{2} \circ w \), where \( k \) and \( w \) abbreviate \( \text{knows} \) and \( \text{worksfor} \), resp. Query plan generation proceeds as follows, where for clarity of presentation we drop explicit use of the concatenation operation \( \circ \):

1. \((kb)^{2}k\) is expanded, giving
   \[
   R’ = k(kwkwk \cup kwkwkwk \cup kwkwkwkwk)w.
   \]

2. Nested unions are pulled up to the top level, giving
   \[
   R’’ = kwkwkwk \cup kwkwkwkwk \cup kwkwkwkwkwkwkwkwk.
   \]

3. Finally, physical execution plans are generated for each of the disjuncts of \( R’’ \). In particular, suppose that \( k = 3 \); then processing each of the disjuncts proceeds as follows:

- \( kkwkw \) is processed, from left to right, generating the physical plan
  \[
  I_{G,k}(w^{-k}k^{-}) \bowtie I_{G,k}(kwkwk)
  \]
  in which \( \bowtie \) is implemented as a merge join. Note the subexpression \( kwk \) has been inverted to obtain the correct sort order to perform a merge join.
- \( kkwkwkwkwk \) is processed from left to right, generating the physical plan
  \[
  I_{G,k}(w^{-k}k^{-}) \bowtie I_{G,k}(kwk) \bowtie I_{G,k}(ww)
  \]
  in which \( \bowtie \) is implemented as a merge join and \( \bowtie \) as a hash join.
- \( kkwkwkwkwkwk \) is processed from left to right, generating the physical plan
  \[
  [(I_{G,k}(w^{-k}k^{-}) \bowtie I_{G,k}(kwk)) \bowtie I_{G,k}(ww)] \bowtie I_{G,k}(w)
  \]
  in which \( \bowtie \) is implemented as a merge join and \( \bowtie \) and \( \bowtie \) as hash joins.

We term this evaluation strategy semi-naive. The complete physical plan is formed as a union of these three sub-plans.

The third step can be optimized by using the histogram \( sel_{G,k} \), as follows. For each disjunct \( D \) of \( R’’ \):

1. if \( |D| \leq k \), return \( I_{G,k}(D) \).
2. Find the most selective k-path subquery \( D’ \) of \( D \) (i.e., the k-path with smallest \( sel_{G,k} \) value). There are \( n - k + 1 \) such subqueries to consider.
3. Let \( D = D_{\text{left}} \bowtie D’ \bowtie D_{\text{right}} \), and recur on \( D_{\text{left}} \) and \( D_{\text{right}} \), to generate query plans for respective output streams \( \text{LEFT} \) and \( \text{RIGHT} \).
4. Determine the cost of each of the following alternative query plans, and return the cheapest plan:

   - \( \text{[LEFT} \bowtie I_{G,k}(D’)] \bowtie \text{RIGHT} \),
   - \( \text{LEFT} \bowtie [I_{G,k}(D’)] \bowtie \text{RIGHT} \),
   - \( \text{LEFT} \bowtie I_{G,k}(D’^{-}) \bowtie \text{RIGHT} \), or
   - \( \text{LEFT} \bowtie [I_{G,k}(D’^{-})] \bowtie \text{RIGHT} \).

We term this evaluation strategy \text{minSupport}.

As an illustration of this strategy, consider again the first disjunct \( kkwkwkwk \) of our example query. Suppose we find that the subquery \( kwk \) is the most selective amongst all four subqueries of length \( k = 3 \) and that the subquery \( ww \) is more selective than the subquery \( k \). Then the physical plan generated by the \text{minSupport} strategy would be

\[
I_{G,k}(k) \bowtie I_{G,k}(kwk) \bowtie I_{G,k}(w^{-w^{-}})
\]

in which \( \bowtie \) is implemented as a merge join and \( \bowtie \) as a hash join.
5. EMPIRICAL EVALUATION

We refer the reader to [12] for full details of an empirical evaluation of our system with respect to a broad spectrum of RPQs over four different real and synthetic datasets. In addition to the semi-naive and minSupport evaluation methods described above, two other methods are investigated: naive, in which k is fixed at 1 (so indexing is on edge labels, not path labels), which corresponds to automaton-based evaluation (approach 1, discussed in the Introduction); and minJoin, which is similar to minSupport but also aims to minimize the number of joins.

As an indicative subset of the empirical results obtained, the three graphs in Figure 2 show the run-times of 8 queries over the Advogato data set, for each of the four evaluation methods, with values of k ranging from 1 to 3. Advogato is a real-world social network having 6,541 nodes and 51,127 edges with |L| = 3, where edges indicate varying degrees of trust between users in the network [11].

We observe that the naive method always performs worst, that the semi-naive method is generally outperformed by minSupport and minJoin, and that the latter two perform similarly. This demonstrates the value of the lightweight histogram data structure for selectivity estimation. We also see that increasing the value of k generally improves the run-times for all methods (apart from naive, where k is fixed at 1 throughout).

6. DEMONSTRATION OVERVIEW

We give participants a hands-on overview of the life of a regular path query, from its submission to our system, through parsing and optimization, to execution. We further demonstrate the speed-ups achieved by our approach compared with Datalog-based evaluation (approach 2, discussed in the Introduction), where our solution is on average 1200x faster on the Advogato queries [12].

Through these interactive activities, we hope to both demonstrate the promise of our approach to RPQ evaluation and to stimulate further broader discussion and study in the research community of engineering strategies for this challenging practical class of graph queries. A system demonstration is an excellent setting in which to accomplish these goals.

7. REFERENCES


The data set is publicly available at http://konect.uni-koblenz.de/networks/advogato.