On Implementing Provenance-Aware Regular Path Queries with Relational Query Engines

Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler,
Eric Gribkoff, Michael Wang, Bertram Ludäscher

Department of Computer Science
University of California, Davis
{scdey, vcuevasv, svkoehler, eagribko, mlywang, ludaesch}@ucdavis.edu

ABSTRACT
Use of graphs is growing rapidly in social networks, semantic web, biological databases, scientific workflow provenance, and other areas. Regular Path Queries (RPQs) can be seen as a core graph query language to answer pattern-based reachability queries. Unfortunately, the number of freely available systems for querying graphs using RPQs is rather limited, and available implementations do not provide direct support for a number of desirable variants of RPQs, e.g., to return those edges that are contained in some (or all) paths that match the given regular expression R. Thus, by returning not just a pair (x, y) of end points of paths that match R, but also "witness edges" (u, v) inbetween, our RPQ variants can be understood as returning additional provenance information about the answer (x, y), i.e., those edges (u, v) that are in some (or all) paths from x to y matching R. We propose a number of such RPQ variants and show how they can be implemented using either Datalog or a suitable RDBMS. Our initial experimental results indicate that RPQs and our provenance-aware variants (RPQProv), when implemented using conventional relational technologies, yield reasonable performance even for relatively large graphs. On the other hand, the overhead associated with some of these variants also makes efficient handling of provenance-aware graph queries an interesting challenge for future research.

Categories and Subject Descriptors
H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords
Graph Querying, Regular Path Queries, Datalog

1. INTRODUCTION
Given a labeled, directed graph G, a regular path query (RPQ) given by an expression R returns those pairs of nodes (x, y) which are connected by a path in G such that the concatenated labels match R. RPQs are found at the core of many graph query languages, representing a flexible and user-friendly core language to concisely express powerful reachability queries. For example, using an Open Provenance Model (OPM)-style provenance graph [14] with edges labeled $\mathtt{u}$ (used) and $\mathtt{g}$ (generatedBy), the lineage graph of an artifact x can be simply computed using the expression $(\mathtt{g.u})^+$, i.e., an alternating sequence of $\mathtt{g}$ and $\mathtt{u}$ edges. However, traditional RPQs may not be sufficient for certain information requirements.

When querying graph data, often various forms of queries arise. For example, given a regular path expression R one may be interested in knowing if starting from a node x another node y (i) is reachable by a path that matches R, (ii) is *not* reachable via R, (iii) is reachable using *only* the given R, or (iv) is reachable via an alternate path *disjoint* from the one that matches R, etc. It would be useful to consider these and other variants of RPQs that allow us to ask these questions directly. As far as we know, no extensive effort has taken place to develop a set of RPQ variants and their corresponding implementation.

Traditionally, a RPQ provides the compliant pairs of nodes in the answer, however, it is often required to determine why a pair of nodes is in the answer, i.e., provide the provenance of a RPQ. Several systems implement RPQs via extensions of SPARQL 1.0 or the property paths forming part of the SPARQL 1.1 standard, but provenance is not considered. Furthermore, SPARQL 1.1 property paths are specified using bag semantics and thus many implementations have significant drawbacks as pointed out in [3]. We adopt a set semantics and capture all the edges from all the paths between a pair of nodes satisfying the query, thus capturing the provenance. Our implementation and experiments show that this is practical and useful.

In this paper, we describe how effective and efficient RPQ engines can be developed by translating RPQ queries into (a) standard Datalog queries, or (b) SQL queries, extended with iteration or recursion. We describe our initial findings for different variants and extensions of RPQs that we found useful for certain applications. We include experimental results for implementations built on different systems. These experiments demonstrate that both approaches are viable for our various types of RPQs, and that particularly good

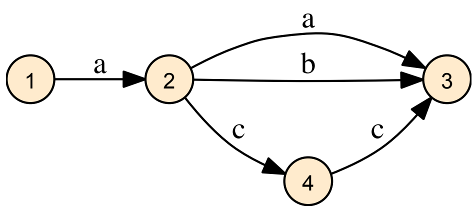

Figure 1: Route map of how cities 1, 2, 3, and 4 are served by airlines a, b, and c.

performance can be achieved by the use of the right techniques. Partial and preliminary aspects of this work have been published earlier in [7] and [4].

2. MOTIVATING EXAMPLE

Let us assume you are planning for a vacation and you want to use the mileage points that you have accumulated thus far. While planning for the vacation, you would like to find connections between cites that are serviced by airlines participating in the mileage program of which you are a member (Query Q_1). Let us also assume that airlines a and b are participating in the mileage program, whereas the airline c is not. Given the route map as shown in Fig. 1, query Q_1 will find that cities (1,2), (2,3), and (1,3) are connected via airlines a or b. The connection (1,2) is in the answer as airline a is servicing it. The connection (2,3) is in the answer as airlines a and b have service between them. The connection (1,3) is in the answer as airline a is servicing the connection (1,2) and airlines a and b are servicing the connection (2,3).

You may want to know the connections with one or more legs serviced by airlines, which are not participating in the mileage program (Query Q_2). This query will answer the connections (1,3), (2,4), and (4,3). The connection (1,3) is in the answer as the legs (2,4) and (4,3) are only serviced by airline c. For the same reason the connections (2,4) and (4,3) are also in the answer.

You may also want to find the connections serviced only by airlines which are participating in the mileage program (Query Q_3). This query will answer the connection (1,2) as only airline a is servicing between them. Note that connections (2,4) and (4,3) are also serviced by only airline c. But, airline c is not participating in the mileage program. In the same same way you may want to find the connections serviced only by airlines which are not participating in the mileage program (Query Q_4). This query will answer the connections (2,4) and (4,3) as only airline c is servicing between them.

It is also useful to find all connections that are connected by at least two disjoint routes serviced by the airlines, which are participating in the mileage program (Query Q_5). This query will answer the connection (2,3) as both airlines a and b are servicing between them.

After reviewing the results so far, you decide to fly from city 1 to city 3 and you want to know all legs you may have to fly (Query Q_6). Based on the route map shown in Fig. 1, to go from city 1 to city 3 flying with the airlines participating in the mileage program, you may fly from city 1 to 2 via airline a and from city 2 to 3 via airline a or b.

You may also be interested to know if there are legs you must fly via some specific airline (i.e., there are no other airlines servicing those legs) while flying from city 1 to city 3 (Query Q_7). To fly from city 1 to 2 there is only one airline a and thus to fly from city 1 to 3, one must fly via airline a, while you can choose between airline a and b to fly from city 2 to 3.

In the following sections, we introduce various variants of RPQ that allow expressing these queries easily.

3. REGULAR PATH QUERIES

3.1 Preliminaries

We consider labeled, directed graphs of the form $G = (N, E, L)$, with N a set of nodes, L a set of labels, and $E \subseteq N \times L \times N$ a set of labeled edges. The graph can be represented by the set E of edges alone, with the understanding that the set N of nodes corresponds to the set of endpoints of the elements of E. We distinguish a special label ϵ representing the empty string such that $(n, \epsilon, n) \in E$ for all $n \in N$.

In our Datalog specification, graphs are represented by a series of edge facts of the form

```
e(u1, l1, v1).
e(u2, l2, v2).
...
e(un, ln, vn).
```

Here, the ϵ edges are implicit, ϵ itself is represented as `eps` in our Datalog implementation.

A standard method to query such graphs is using Regular Path Queries (RPQ). Let $\Sigma = \{l_1, ..., l_n\}$ be a set of base labels (here: $\Sigma = L$). A RPQ is a query that returns pairs of nodes in a graph that satisfy a regular expression R. The expressions R we will employ are defined over the alphabet L of edge labels by the following context-free grammar:

R ::=

	l	// for $l \in L$
|	$R.R$	// concatenation
|	$R \mid R$	// alternation
|	R^*	// transitive closure
|	R^+	// positive trans. closure
|	R^{-1}	// inverse
|	$R^?$	// optional
|	(R)	

The concatenation, alternation, and closure operators have the usual meanings associated with regular expressions (RE). The inverse operator denotes an inversion of the direction of an edge (swapping the source and destination of the edge), while the optional operator indicates that a path can match even if the optional part does not hold.

Let us assume that Σ^* is the set of all strings over Σ. Then the complement (R^c) of a regular expression R is defined as $\Sigma^* - L(R)$, where $L(R)$ is the language of R and is defined as the set of accepted strings over R.

3.2 Regular Path Queries Variants

Here, we describe different variants of regular path queries, including their extensions that take provenance into consideration. The name of each variant corresponds to a relation name in our Datalog implementation. We also show how all the queries described in Section 2 can be answered using these variants.

RPQSome. This relation returns all the pairs of nodes which satisfy the given R and thus can be used to answer Query Q_1 as discussed in Section 2.

Formally, given a graph G, an ordered pair of nodes (x_0, x_n) of G is in the result of an RPQSome query Q_R^{some} specified by a regular expression R, the result being denoted as $Q_R^{some}(G)$, iff $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is a path in G and $l_1 l_2 ... l_n \in L(R)$, where $L(R)$ denotes the language defined by the regular expression R.

RPQNot. This relation returns the pairs of nodes which satisfy R^c, the complement of the given regular expression R. It can be used to answer Query Q_2.

Formally, given a graph G, an ordered pair of nodes (x_0, x_n) of G is in the result of an RPQNot query Q_R^{not} specified by a regular expression R, the result being denoted as $Q_R^{not}(G)$, iff $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is a path in G and $l_1 l_2 ... l_n \in L(R^c)$, where $L(R^c)$ denotes the language defined by the regular expression R^c.

RPQAll. This relation returns a pair of nodes if it satisfies R while it does not satisfy the R^c. Query Q_3 can be answered using this query.

Given a graph G, an ordered pair of nodes (x_0, x_n) of G is in the result of an RPQAll query Q_R^{all} specified by a regular expression R, the result being denoted as $Q_R^{all}(G)$, iff (i) there is a path π such that $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is in G and $l_1 l_2 ... l_n \in L(R)$, where $L(R)$ denotes the language defined by the regular expression R, and (ii) there is no path π such that $\pi' = x_0 \xrightarrow{m_1} x'_1 \xrightarrow{m_2} x'_2 \xrightarrow{m_3} ... \ x'_{n-1} \xrightarrow{m_n} x_n$ is in G and $m_1 m_2 ... m_n \in L(R^c)$, where $L(R^c)$ denotes the language defined by the regular expression R^c.

RPQOther. This relation returns a pair of nodes if it does not satisfy R while it satisfies the R^c. It can answer Query Q_4.

Given a graph G, an ordered pair of nodes (x_0, x_n) of G is in the result of an RPQOther query Q_R^{other} specified by a regular expression R, the result being denoted as $Q_R^{other}(G)$, iff (i) there is no path π such that $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is in G and $l_1 l_2 ... l_n \in L(R)$, where $L(R)$ denotes the language defined by the regular expression R, and (ii) there is at least a path π such that $\pi' = x_0 \xrightarrow{m_1} x'_1 \xrightarrow{m_2} x'_2 \xrightarrow{m_3} ... \ x'_{n-1} \xrightarrow{m_n} x_n$ is in G and $m_1 m_2 ... m_n \in L(R^c)$, where $L(R^c)$ denotes the language defined by the regular expression R^c.

RPQDisjoint. This relation returns a pair of nodes if it satisfies R with some paths between them and all these paths are disjoint, i.e., they do not share any common edges. To answer Query Q_5 this relation can be used.

Given a graph G, an ordered pair of nodes (x_0, x_n) of G is in the result of an RPQDisjoint query $Q_R^{disjoint}$ specified by a regular expression R, the result being denoted as $Q_R^{disjoint}(G)$, iff $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is a path in G and $l_1 l_2 ... l_n \in L(R)$, where $L(R)$ denotes the language defined by the regular expression R, and $\bigcap_{\pi \in path(x_0, x_n)} edge(\pi) = \emptyset$ where the relation $path(x_0, x_n)$ contains all the paths between x_0 and x_n satisfying the given regular expression and the relation $edge(\pi)$ contains all the edges for the path π.

It is important to remark that RPQSome queries can be expressed in a linear Datalog program, i.e. with no rule body having more than one intensional database (inferred) goal and, particularly, by linear programs referred to as linear chain programs. The generation of the linear program is straightforward once the regular expression has been transformed into a regular grammar, for which well-known algorithms exist. The details can be found in [1]. Our experiments, as will be shown later, demonstrate that this can lead to major performance gains. A discussion of how such transformations can fit in our work is presented later in the paper.

3.3 Provenance Aware RPQ

In many cases, it is important and required to know why a pair of nodes appear in the result of a query using any of the above RPQ relations. A pair of nodes appears in a result if there is a path which satisfies the regular expression. To explain why a pair of nodes appear in the result, we can provide the set of edges of all satisfying paths. This makes our RPQ implementation provenance-aware. Next we define the relations we implement to capture the provenance of an RPQ query.

RPQProvUnion. Given a graph G and a regular expression R, an ordered quintuple (x_0, x_n, u, l, v) of nodes x_0, x_n, u, v and a label l of G is in the result of an RPQProvUnion query Q_R^{union} iff $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is a path in G, $l_1 ... l_n \in L(R)$, $u = x_i$, $v = x_{i+1}$ and $l = l_{i+1}$ for some i such that $0 \leq i \leq n-1$. The set $Q_R^{union}(G)_{[x_o, x_n]} = \{(u, l, v) : (u, l, v) \in \bigcup_{\pi \in path(x_0, x_n)} edge(\pi)\}$ where the relation $path(x_0, x_n)$ contains all the paths between x_0 and x_n satisfying the given regular expression and the relation $edge(\pi)$ contains all the edges for the path π. Here, the set $Q_R^{union}(G)_{[x_o, x_n]}$ contains all of the edges of all of the paths that satisfy the regular expression for a given pair of nodes $(x_o, x_n) \in G$. Intuitively, if $Q_R^{union}(G)_{[x_o, x_n]}$ is not empty, it justifies why the pair (x_0, x_n) is in the result. This relation can be used to answer Query Q_6.

RPQProvIntersection. Given a graph G, an ordered quintuple (x_0, x_n, u, l, v) of nodes x_0, x_n, u, v and a label l of G is in the result of an RPQProvIntersection query $Q_R^{intersection}$, specified by a regular expression R iff $\pi = x_0 \xrightarrow{l_1} x_1 \xrightarrow{l_2} x_2 \xrightarrow{l_3} ... \ x_{n-1} \xrightarrow{l_n} x_n$ is a path in G, $l_1 ... l_n \in L(R)$, $u = x_i$, $v = x_{i+1}$ and $l = l_{i+1}$ for some i such that $0 \leq i \leq n-1$. The set $Q_R^{intersection}(G)_{[x_o, x_n]} = \{(u, l, v) : (u, l, v) \in \bigcap_{\pi \in path(x_0, x_n)} edge(\pi)\}$ where the relation $path(x_0, x_n)$ contains all the paths between x_0 and x_n satisfying the given regular expression and the relation $edge(\pi)$ contains all the edges for the path π. Here, the set $Q_R^{intersection}(G)_{[x_o, x_n]}$ contains all the edges presents in all the paths for the pair of nodes (x_0, x_n) satisfying the regular expression. Therefore, these edges must be traversed for any path between the given nodes that satisfies the expression. Intuitively, the partial justifications common to all justifications. This relation can be used to answer Query Q_7.

4. IMPLEMENTATION

4.1 Datalog-based Implementation

To evaluate queries represented by REs using Datalog, we introduce a set of auxiliary predicates `conc`, `or`, `plus`, `star`, `inv`, and `opt`. Here the `conc` and `or` predicates specify how to compose two expressions for the concatenation and alternation operators, respectively. Similarly, the `plus` and `star` predicates enable applying the positive transitive closure and transitive closure operators respectively to single expressions. Finally, the `inv` and `opt` predicates apply the inverse and optional operators. In all cases the new expressions are represented in postfix notation, which makes it simpler to generate the above predicates automatically for a given RE. These predicates can be defined using Skolem functions as shown below:

```
conc(R1,R2,R3) :- R3 = f_conc(R1,R2).
or(R1,R2,R3) :- R3 = f_or(R1,R2).
plus(R1,R2) :- R2 = f_plus(R1).
star(R1,R2) :- R2 = f_star(R1).
```

However, these rules are not safe under the standard bottom-up Datalog evaluation, so instead we create a finite set of facts, consisting only of the sub expressions of a given query R. For example consider the RPQ expression R = $((a.b)^* \ |c)^+$. We will generate the facts as shown below:

```
conc(a, b, "a b .").
star("a b ." , "a b . *").
or("a b . *" , c, "a b . * c |").
plus("a b . * c |" , "a b . * c | +").
```

After creating these facts for a given regular expression, a RPQ can be evaluated using the Datalog engine described below:

```
(1) p(X,L,Y) :- e(X,L,Y).

(2) p(X,R1concR2,Y) :-
      conc(R1,R2,R1concR2),
      p(X,R1,Z), p(Z,R2,Y).

(3) p(X,R1orR2,Y) :-
      or(R1,_,R1orR2), p(X,R1,Y).
(4) p(X,R1orR2,Y) :-
      or(_,R2,R1orR2), p(X,R2,Y).

(5) p(X,Rplus,Y) :-
      plus(R,Rplus), p(X,R,Y).
(6) p(X,Rplus,Y) :-
      plus(R,Rplus),
      p(X,R,Z), p(Z,Rplus,Y).

(7) p(X,Rstar,X) :-
      star(R,Rstar), p(X,_,_).
(8) p(X,Rstar,X) :-
      star(R,Rstar), p(_,_,X).
(9) p(X,Rstar,Y) :-
      star(R,Rstar), p(X,R,Y).
(10) p(X,Rstar,Y) :-
      star(R,Rstar),
      p(X,R,Z), p(Z,Rstar,Y).

(11) p(X,Rinv,Y) :-
      inv(R,Rinv),
      p(Y,R,X).

(12) p(X,Ropt,Y) :-
      opt(R,Ropt),
      p(X,R,Y).
(13) p(X,Ropt,X) :-
      opt(R,Ropt),
      p(X,R,_).
(14) p(X,Ropt,X) :-
      opt(R,Ropt),
      p(_,R,X).

(15) rpqsome(X,R,Y) :- p(X,R,Y), q(R).

(16) q(query expression).
```

The predicate `p(X,R,Y)` states that there is a path from node `X` to node `Y` according to the regular (sub)expression `R`. Rule (1) represents the case when the subexpression is a single label. Rule (2) implements the concatenation operator, in which two nodes are traversed through an intermediate node `Z` and their labels concatenated. In turn, rules (3) and (4) implement the alternation operator, matching either the subexpressions `R1` or `R2`. Rules (5) and (6) implement the positive transitive closure (+) by a variant of the well-known transitive closure (TC) rules. The transitive closure (*) is also implemented by rules similar to traditional TC, namely rules (9) and (10); additionally, rules (7) and (8) address the case of single nodes by extracting them from the `g` predicate. Rule (11) implements the inverse while rules (12) to (14) implement the optional operator. Finally, the result is obtained through the `rpqsome` predicate in rule (15) using the `p` predicate and an additional `q` predicate that specifies the actual RE representing the query in rule (16).

Now, using the `rpqsome` predicate we express all other predicates introduced in Section 3.2. For the $\texttt{rpqsome}^c$ predicate we first compute R^c and create the auxiliary facts (i.e., `conc`, `or`, etc.) based on R^c. We use the same set of rules (1) though (16) just by changing the IDB names. For example, we use $\texttt{g}^c$ instead of `g`, and $\texttt{conc}^c$ instead of `conc`. In rule (20), we define `rpqdisjoint` using the `rpqintersect` predicate, which we will describe in paragraph RPQProvIntersection below.

```
(17) rpqnot(X,R,Y) :-
      rpqsome^c(X,R^c,Y), q(R).

(18) rpqother(X,R,Y) :-
      rpqsome^c(X,R^c,Y),
      not rpqsome(X,R,Y), q(R).

(19) rpqall(X,R,Y) :-
      rpqsome(X,R,Y),
      not rpqsome^c(X,R^c,Y).

(20) rpqdisjoint(X,R,Y) :-
      rpqsome(X,R,Y),
      not rpqintersect(X,R,Y,_,_,_).
```

We now discuss our Datalog implementation of the predicates defined in Section 3.3. These predicates provide the provenance of a query result of any of the predicates defined in Section 3.2. For example, `rpqpunion` provides the explanation for why a tuple is in the result of `rpqsome`.

RPQProvUnion. The following Datalog rules implement RPQProvUnion queries:

```
(1) p(X,L,Y,X,L,Y) :- e(X,L,Y).

(2) p(X, LconcM,Y,U,L2,V) :-
      p(X,L1,Z,U,L2,V), p(Z,M1,Y,_,_,_),
      conc(L1,M1,LconcM).
(3) p(X, LconcM,Y,U,M2,V) :-
      p(X,L1,Z,_,_,_), p(Z,M1,Y,U,M2,V),
      conc(L1,M1,LconcM).

(4) p(X,LorM,Y,U,L2,V) :-
      p(X,L1,Y,U,L2,V), or(L1,_,LorM).
(5) p(X,LorM,Y,U,M2,V) :-
```

```
        p(X,M1,Y,U,M2,V), or(_,M1,LorM).

(6) p(X,Lplus,Y,U,L1,V) :-
        plus(L,Lplus), p(X,L,Y,U,L1,V).
(7) p(X,Lplus,Y,U,L1,V) :-
        p(X,L,Z,U,L1,V), p(Z,Lplus,Y,_,_,_),
        plus(L,Lplus).
(8) p(X,Lplus,Y,U,L1,V) :-
        p(X,L,Z,_,_,_), p(Z,Lplus,Y,U,L1,V),
        plus(L,Lplus).

(9) p(X,Lstar,X,X,eps,X) :-
   star(L,Lstar), p(X,_,_,_,_,_).
(10) p(X,Lstar,X,X,eps,X) :-
   star(L,Lstar), p(_,_,X,_,_,_).
(11) p(X,Lstar,Y,U,L1,V) :-
   p(X,L,Z,U,L1,V), p(Z,Lstar,Y,_,_,_),
   star(L,Lstar).
(12) p(X,Lstar,Y,U,L1,V) :-
   p(X,L,Z,_,_,_), p(Z,Lstar,Y,U,L1,V),
   star(L,Lstar), L1 ≠ eps.

(13) rpqpunion(X,L,Y,U,M,V) :-
   p(X,L,Y,U,M,V), q(L).

(14) q(query expression).
```

These rules are analogous to the rules for RPQSome queries presented in Section 3.2. The main difference is that the `p` predicate is now of the form `p(X,R,Y,U,L,V)` where the `(U,L,V)` components represent the edges that form part of the result according to the previous definition. Accordingly, additional rules are required to derive these edges for the concatenation and transitive closure operators. For the transitive closure in particular, rules are introduced for the recursive case that traverses from a node `X` through an edge to a node `Z`, and then from that node `Z` through a path to the node `Y`. One rule adds the edge from `X` to `Z` and another the edges from `Z` to `Y`. Special care must be taken for the transitive closure (*) in rule (12) to avoid redundant ϵ edges. The result is obtained through the `rpqpunion` predicate in conjunction with the `q` predicate specifying the query. For brevity we omit the rules for the inverse and optional operators.

RPQProvIntersection. The evaluation of RPQProvIntersection queries is closely related to the problem of finding the dominators in flow graphs. A flow graph is a directed graph with a distinguished start node $s \in N$ such that all nodes in N are reachable from s. A node v dominates a node n if every path from the start node s to n must pass through v. By definition, every node dominates itself.

Although dominators are defined for nodes, it is straightforward to apply the concept to edges. It suffices to create an *extended* graph in which an artificial node is created to represent each edge. Thus, an edge $(u, l, v) \in E$ is replaced by two edges (u, l, m) and (m, l, v), where m is the new node representing the edge in the original graph.

Our Datalog implementation starts by creating the extended graph, again taking advantage of the rules (1) to (14) from Section 4.1 to compute the RPQProvUnion query. The rules are as follows:

```
(1)
   ...
(14)

(15) node(X) :- rpqpunion(_,_,_,X,_,_).
(16) node(X) :- rpqpunion(_,_,_,_,_,X).
(17) label(X) :- rpqpunion(_,_,_,_,X,_).

(18) extnode(X) :- node(X).
(19) extedge(U,M,id(U,M,V)) :-
   rpqpunion(_,_,_,U,M,V).
(20) extedge(id(U,M,V),M,V) :-
   rpqpunion(_,_,_,U,M,V).
(21) extnode(id(X,L,Y)) :-
   extedge(X,_,id(X,L,Y)), node(X).
(22) extnode(id(X,L,Y)) :-
   extedge(id(X,L,Y),_,Y), node(Y).
```

Rules (15) to (17) extract the nodes and labels from the result of the RPQProvUnion query, while rules (18) to (22) create the extended graph. Being able to distinguish nodes of the original graph from those introduced in the extended graph becomes important later. In rules (19) to (22) the `id` predicates represent the use of a Skolem function to create the new nodes "on the fly".

The dominators of a node n are given by the maximal solution to the following data-flow equations:

$$Dom(s) = s \; Dom(n) = (\bigcap_{p \in preds(n)} Dom(p)) \cup n$$

where the function $Dom : N \to \mathcal{P}(N)$ gives the dominators of a particular node. The first equation states that the dominator of the start node is the start node itself. The second equation states that the dominators of a given node are represented by the intersection of the corresponding dominators of all of its predecessors, when predecessors are direct and not transitive, in addition to the node itself.

The above equations cannot be specified in Datalog directly because the universal quantification on the predecessors is not supported. However, existential quantification can be employed instead according to the equivalence $\neg\forall x.F \Leftrightarrow \exists x.\neg F$. The corresponding Datalog rules to obtain the dominators for the extended graph are as follows:

```
(23) extdomcomp(S,S,N) :-
   node(S), extnode(N), S ≠ N.
(24) extdomcomp(S,N1,N) :-
   extedge(P,_,N1), extdomcomp(S,P,N), N1 ≠ N.
(25) extdom(S,N1,N2) :-
   not extdomcomp(S,N1,N2), extnode(N1),
   extnode(N2), extnode(S), exttc(S,N1).

(26) exttc(X,Y) :- extedge(X,_,Y).
(27) exttc(X,Y) :- exttc(X,Z), extedge(Z,_,Y).
```

The predicate `extdom(S,N1,N2)` states that, taking node `S` as a start node, node `N2` is a dominator of node `N1`. To be able to use existential quantification as just described, we define the `extdomcomp` predicate that represents the negation of the `extdom` predicate in rules (23) and (24), then in rule (25) we define the actual `extdom` predicate by the negation of the `extdomcomp` predicate. Since we used the set complements for the `S` and `N1` nodes in rules (23) and (24), we restrict the `extdom` predicate to those nodes reachable from `S` with the use of the `exttc` predicate in rule (25), the transitive closure of the extended graph that is computed in rules (26) and (27), to avoid false results. Once the dominators for the extended graph have been computed the result of the query can be obtained with the following additional rules:

```
(28) internal(S,E,N) :- extdom(S,E,N), N ≠ S,
   N ≠ E, node(S), node(E), not node(N).

(29) rpqintersect(S,E,N1,L,N2) :-
   internal(S,E,N), extnode(N), N=id(N1,L,N2),
   node(N1), node(N2), label(L).
```

The dominators can be represented as a tree with the start node at the root. The ancestors of a node correspond to the dominators of that node. If all of the dominators are trivial, i.e. the start node dominates all other nodes only, a two level tree is formed. If the tree has more than two levels then some dominators are not trivial. We find those with the `internal` predicate in rule (28). We are interested only in dominators representing edges in the extended graph, therefore we introduce the `not node(N)` clause in rule (28), since the nodes created in the extended graph to represent edges are not nodes in the original graph, while the converse does not hold. Finally, rule (29) obtains the results by unnesting the `id` predicates representing edges found to be dominators in the original graph.

RPQProvAll. In addition to the rules (1) to (14) presented in Section 4.1 for RPQProvUnion queries, the following Datalog rules enable the evaluation of RPQProvAll queries:

```
(1)
   ...
(14)

(15) tc(X,Y,X,L,Y) :- e(X,L,Y).
(16) tc(X,Y,U,L1,V) :-
   tc(X,Z,U,L1,V), tc(Z,Y,_,_,_).
(17) tc(X,Y,U,L1,V) :-
   tc(X,Z,_,_,_), tc(Z,Y,U,L1,V).

(18) tc_diff_rpqpunion(X,Y,U,L1,V) :-
   tc(X,Y,U,L1,V), not rpqpunion(X,Y,U,L1,V).

(19) exclude(X,Y) :-
   tc_diff_rpqunion(X,Y,U,L1,V).

(20) rpqpall(X,L,Y,U,M,V) :-
   rpqpunion(X,Y,U,M,V), not exclude(X,Y).
```

Our evaluation of RPQProvAll queries is based on a procedure similar to the relational algebra division operator. First we compute in lines (15) to (17) the transitive closure of the original graph including the edges that connect each pair of nodes in the TC, regardless of the strings formed by the labels on the paths. Then in line (18) we compute the set difference of the TC with the result of the RPQProvUnion query under the same RE. This enables us to determine the pairs of nodes that are connected by some path that does not adhere to the RE. These pairs of nodes are then extracted in the `exclude` predicate of rule (19). Finally, in rule (20) the pairs of nodes that should be excluded are removed from the RPQProvUnion result yielding the result of the RPQProvAll query.

4.2 RPQ Evaluation on a RDBMS

We also implemented an evaluator for RPQ and RPQ provenance queries based on the capabilities provided by a relational DBMS, in particular PostgreSQL. Figure 2 shows the architecture of our system, which was developed in the Python programming language. First, the RE representing the query is parsed to generate a parse tree. A parser built using the ANTLR [1] parser generator and based on the grammar presented in Section 3.1 performs this task.

The key in our approach is to cumulatively build the result by traversing the parse tree in a bottom-up manner and evaluating each subexpression by means of programmatically generated SQL queries that operate on the base graph. Thus, our RDBMS-based evaluation has a strong resemblance to our evaluation using Datalog. Concretely, the base graph is represented by a relation g containing tuples of the form $g(x,l,y)$ for RPQ and $g(x,l,y,u,l',v)$ for RPQ provenance queries.

[1] http://www.antlr.org/

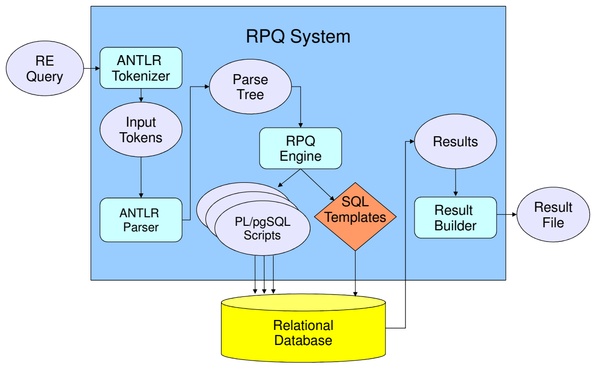

Figure 2: RDBMS-based evaluator architecture

The base graph relation not only serves as input but also as a working space on which the result is built. At any given time during the query evaluation the l attribute holds the subexpression describing the path connecting nodes x and y. The u, l', and v attributes for the RPQ provenance tuples denote edges $u \xrightarrow{l'} v$ that form part of some path between x and y satisfying the RE. For simplicity, the REs are represented using postfix notation.

The `SQLGenerator` generates SQL queries to evaluate subexpressions based on a series of templates, as indicated by the parse tree. The results of the alternation, concatenation, inverse, and optional operators can be computed by standard SQL queries. However, the transitive closure operators, are evaluated by PL/pgSQL scripts using the embedded programming language of PostgreSQL. When the entire parse tree has been evaluated, the desired result can be obtained by performing a selection on the g relation with the label being the full RE representing the query. We next describe the evaluation of the TC operators for RPQ provenance queries, since it is the most computationally expensive.

Listing 1 presents the main SQL query used within the PL/pgSQL script to evaluate both transitive closure operators (* and +). Before executing this query, the tuples of interest from the g relation are stored in a temporary `g2` relation, each tuple identified by an integer id generated by a sequence.

A relation tc is then created with the `RECURSIVE` clause of SQL-99. It consists of the start node, subexpression, end node and an array containing the path formed during the graph traversal. Because only the edges that satisfy the expression under evaluation were inserted in the $g2$ relation, and then in the tc relation in the base case, the `WHERE` clause of the recursive part of the query only needs to verify that the nodes are connected. The `SELECT` clause then projects the `array_union` of the path array with the joined edge, storing only the ids of the edges. The expression of the `tc` tuples corresponds to the concatenation of the expression under evaluation, represented by the parameter `edgelabel`, with the TC operator specified by the `tcsymbol` parameter (representing '*' or '+' since we are using postfix notation).

An aggregation is then applied on the start node and end node of

Listing 1: Recursive SQL query to evaluate RPQProvUnion queries

```
INSERT INTO g(
   WITH RECURSIVE tc(compstart, label1, compend, path) AS (
      SELECT g2.compstart as compstart, edgelabel || tcsymbol as label1, g2.compend as compend,
          ARRAY[g2.id]
      FROM g2
         UNION ALL
      SELECT compstart, edgelabel || tcsymbol, compend, array_union(path) FROM (
         SELECT tc.compstart as compstart, edgelabel || tcsymbol as label1, g2.compend as compend,
             array_union(tc.path, ARRAY[g2.id]) as path
         FROM g2, tc
         WHERE tc.compend = g2.compstart
      ) as foo GROUP BY compstart, compend
   )
   SELECT DISTINCT v.compstart, v.label1, v.compend, g2.basestart, g2.label2, g2.baseend
   FROM (
      SELECT compstart, label1, compend, unnest(path) as path
      FROM (
         SELECT compstart, edgelabel || tcsymbol as label1, compend, array_union(path) as path
         FROM tc
         GROUP BY compstart, compend
      ) as u
   ) as v
   LEFT JOIN g2 ON v.path = g2.id
);
```

`tc`, yielding an array of the edges that connect each two nodes forming part of the RPQProvUnion result. It is important to remark that applying the aggregation inside the recursive query yields significant performance gains over applying it outside. The rest of the query will simply unnest the results taking advantage of the edge ids. Again the use of aggregation results in performance gains. The result tuples of the query are then inserted into the g relation.

For the evaluation of the RPQProvAll and RPQProvIntersection queries we adopted a variant of the strategy just described and is ommitted for brevity. In addition to using the `RECURSIVE` clause of SQL-99, we experimented with scripts implementing the seminaive evaluation strategy employed in Datalog. However, the performance turned out to be between 5 and 10 times slower in many cases. Significant gains were also achieved by the use of indexes on the labels for concatenation intensive queries in particular, without major overhead for the rest of the queries. Additional details of the RDBMS-based evaluation process can be found in [16].

5. EXPERIMENTAL RESULTS

We evaluated the performance of our Datalog and RDBMS-based implementations for most of the query types we identified in Section 3.1, with our focus being the overall execution time of the queries. In the following experiments we employed the Postgres implementation discussed in Section 4.2 (using PostgreSQL 8.1), while in the case of Datalog we used two systems: the DLV [2] system developed to support disjunctive and answer set programming, and the Datalog engine of the commercial LogicBlox [3] platform for enterprise applications in version 3.9. The experiments were carried out on a server with 2 Quad-Core AMD Opteron Processors 2389 (8 cores at 2.9GHz), 32GB RAM memory, 2 1TB HDD as Raid1, Debian Linux kernel 3.2.0-4-amd64 SMP.

5.1 RPQSome and RPQProvUnion

The goals of our experiments regarding these queries were twofold. First, we wanted to measure how expensive it is to compute the queries for the RDBMS and Datalog implementations. Second, we wanted to compare how expensive it is to evaluate RPQProvUnion queries as opposed to RPQSome queries.

With these purposes in mind, we adopted as a benchmark a series of graphs representing binary decision diagrams (BDDs) corresponding to instances of the well-known N-queens problem. BDDs encode boolean functions as directed acyclic graphs consisting of two terminal nodes (1-terminal and 0-terminal for a true and false evaluation, respectively) and a series of decision nodes with edges labeled 1 and 0 whose path to the terminal nodes represent variable assignments. We generated these graphs by means of the JavaBDD [4] library.

Figure 3 presents the results for the queries `0.0.0` and `0+`, which we chose as representative for conventional and transitive closure operators, respectively. The graphs concerned represent problem instances ranging from 7 to 12 queens, and from aproximately 1100 to 430,000 nodes, the number of edges being typically twice the number of nodes.

We can observe that the performance of the DLV and Postgres implementations is very similar, except for the positive transitive closure in RPQSome queries, for which the Postgres implementation is significantly more efficient, a discrepancy that we will address shortly. The performance of LogicBlox 3.9 is by far the best, which we believe is caused by better optimizations for recursion and better use of available main memory. On the other hand, Postgres is significantly constrained by I/O. In turn, DLV was developed to support novel variants of logic programming rather than mainly conventional Datalog.

Although RPQProvUnion queries are significantly more computationally expensive than RPQSome queries, as the experiments show, it is possible to evaluate them for relatively large graphs. The time required can be high, but to some extent this is unavoidable due to the size of the results, which are large but remain feasible.

[2]http://www.dlvsystem.com/dlvsystem/index.php/Home

[3]http://www.logicblox.com/

[4]http://javabdd.sourceforge.net/

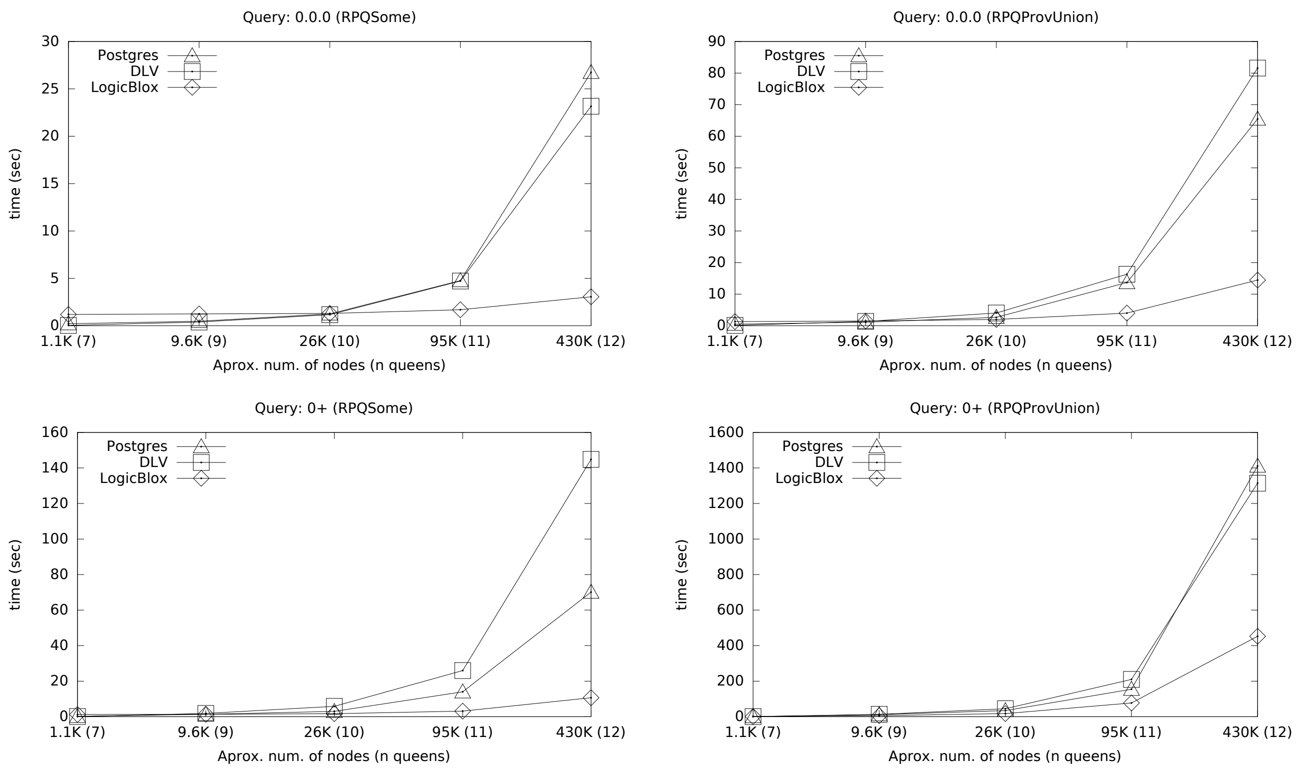

Figure 3: Evaluation of RPQSome and RPQProvUnion queries

5.2 Linear vs. non-linear Datalog

It is to be expected that major performance differences could arise in the evaluation of transitive closure operators when employing non-linear as opposed to linear Datalog programs. The differences and further implications for parallel and cluster computing are studied in [2]. Consequently, we performed an experiment to test the non-linear Datalog programs to evaluate RPQSome queries presented in Section 3.2 along with the linear programs derivable from regular grammars representing the query.

For this experiment, we devised a simple benchmark that would stress-test the implementations by generating a large number of results for a small graph. This benchmark is based on what we call ladder graphs of different levels. Figure 4 (left) shows a ladder graph of 2 levels. The query we employed is $(a|b)+$, which is expected to generate a large number of results for a graph with hundreds or thousands of levels.

The results are presented in Figure 5. It is important to remark that in accordance to SQL-99 Postgres implements linear recursion only. The non-linear DLV program shows the worst performance, being unfeasible to evaluate the query on a graph with 1500 levels. The linear Datalog counterpart for DLV shows better performance, similar but inferior to the non-linear program in LogicBlox 3.9. For a query intensely dominated by transitive closure, the advantages of LogicBlox 3.9 are hindered by the use of a non-linear program. The linear recursion implementation in Postgres shows good performance, but it is significantly surpassed by the linear program in LogicBlox 3.9. The advantageous characteristics of linear Datalog programs, such as their suitability for parallelism [15], can result in very good performance for an adequate implementation.

N links	Postgres [sec]	DLV [sec]	LogicBlox [sec]
50	3.26	91.3	6.053
100	21.93	1376.37	36.878
200	174.79	$\perp$	291.422

Table 1: Experimental results for RPQProvAll query a+

5.3 RPQProvAll and RPQProvIntersection

We developed a simple benchmark to test RPQProvAll and RPQProvIntersection queries. It consists of graphs of the form presented in Figure 4 (right), which we call chain queries. The figure shows one with 2 links. The results are presented in Table 1 for the query `a+`.

Clearly the Postgres implementation significantly outperforms the DLV implementation. Our Datalog program designed for LogicBlox 3.9 is slower than our SQL implementation in Postgres but significantly faster than DLV. We are currently working on an alternative specification of RPQProvAll queries in Datalog. In relation to the equivalent RPQProvIntersection query, the runtimes are significantly lower in both cases given that the results are represented by the bridges connecting the various links in the graph, and are therefore omitted. We are currently working on the development of a more representative benchmark for this type of query.

6. RELATED WORK

The theoretical and complexity characterization of the RPQs on graphs that we categorize as RPQSome is given in [13], where an automata-based algorithm is proposed for their evaluation. We opted alternatively for an implementation based on the declarative capabilities of RDBMSs and Datalog. Although the implementa-

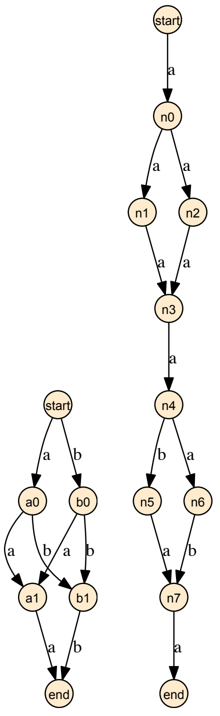

Figure 4: Ladder benchmark graph of 2 levels (left) and chain graph with 2 links (right)

tion of transitive closure in RDBMSs has been extensively studied (e.g. [9]) we are not aware of their previous use to evaluate RPQs, for which we present a successful implementation.

The authors of [10] present a heuristic approach to evaluate RPQs (RPQSome in our categorization) on large graphs more efficiently. This heuristic is based on performing searches starting on nodes of the graph having rare labels. The problem of RPQ evaluation in a distributed setting is studied in [8] along with conventional reachability queries, where a series of performance guarantees based on a technique called partial evaluation is presented. We are considering the use of such techniques for the evaluation of RPQProv queries.

The RPQs corresponding to RPQAll in our categorization are addressed in [11] by extending the work presented in [5], but referred to as universal RPQs. The authors in [11] present algorithms for the evaluation of a variant called parametric RPQs comprising the existential (RPQSome) and universal (RPQAll) categories. The parameters serve to represent program variables, and specific parametric RPQs serve to solve a variety of program verification problems. We adopt a different evaluation approach and consider also RPQs in conjunction with provenance.

7. CONCLUSIONS

We have presented variants of RPQs that adopt the notion of the provenance of RPQ results as well as how to evaluate them. In particular, we presented evaluation techniques based on a RDBMS and on Datalog. Our experiments on both implementations show that with adequate techniques it is feasible to compute RPQProv queries for relatively large graphs, even though they are significantly more computationally intensive.

Additional experiments demonstrated that the use of linear recur-

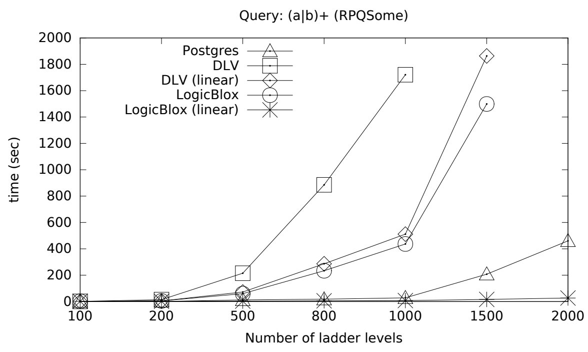

Figure 5: Linear vs. non-linear Datalog evaluation

sion has major benefits in terms of performance. Consequently, future work includes extending the transformation techniques in [1] to RPQProv queries. Another interesting research direction is to develop techniques for RPQProv query evaluation with cluster computing platforms such as MapReduce [6] and Pregel [12].

We believe that our current results, particularly the RDBMS-based implementation, are effective enough to serve as a foundation to develop tools for provenance querying. Ongoing work focuses on the development of a suitable Web-based GUI to interact with our RDBMS RPQ query engine as well as additional components to enable provenance visualization and querying.

8. ACKNOWLEDGMENTS

Work supported in part by NSF award OCI-0830944 (DataONE), and a gift from LogicBlox, Inc.

9. REFERENCES

[1] F. Afrati and F. Toni. Chain queries expressible by linear datalog programs. In *Deductive Databases and Logic Programming (DDLP)*, pages 49–58, 1997.

[2] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 132–143, New York, NY, USA, 2012. ACM.

[3] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 629–638, New York, NY, USA, 2012. ACM.

[4] V. Cuevas-Vicenttín, S. Dey, B. Ludäscher, M. Li Yuan Wang, and T. Song. Modeling and querying scientific workflow provenance in the d-opm. In *7th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, november 2012.

[5] O. De Moor, D. Lacey, and E. Van Wyk. Universal regular path queries. *Higher Order Symbol. Comput.*, 16(1-2):15–35, Mar. 2003.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] S. Dey, S. Köhler, S. Bowers, and B. Ludäscher. Datalog as a lingua franca for provenance querying and reasoning. In

Workshop on the Theory and Practice of Provenance (TaPP), 2012.

[8] W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *Proc. VLDB Endow.*, 5(11):1304–1316, July 2012.

[9] R. Kabler, Y. E. Ioannidis, and M. J. Carey. Performance evaluation of algorithms for transitive closure. *Inf. Syst.*, 17(5):415–441, Sept. 1992.

[10] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proceedings of the 24th international conference on Scientific and Statistical Database Management*, SSDBM'12, pages 177–194, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 219–230, New York, NY, USA, 2004. ACM.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[13] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, Dec. 1995.

[14] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.*, 27(6):743–756, June 2011.

[15] J. D. Ullman and A. Van Gelder. Parallel complexity of logical query programs. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 438–454, Washington, DC, USA, 1986. IEEE Computer Society.

[16] M. L. Y. Wang. Querying Provenance as Regular Path Queries with Relational Databases. Master's thesis, University of California, Davis, 2012.