Techniques for Efficiently Querying Scientific Workflow Provenance Graphs

Manish Kumar Anand¹, Shawn Bowers², Bertram Ludäscher¹,³
¹Dept. of Computer Science, University of California, Davis
²Dept. of Computer Science, Gonzaga University
³Genome Center, University of California, Davis
{maanand, ludaesch}@ucdavis.edu, bowers@gonzaga.edu

ABSTRACT
A key advantage of scientific workflow systems over traditional scripting is their ability to automatically record data and process dependencies introduced during workflow runs. This information is often represented through provenance graphs, which can be used by scientists to better understand, reproduce, and verify scientific results. However, while most systems record and store data and process dependencies, few provide easy-to-use and efficient approaches for accessing and querying provenance information. Instead, users formulate provenance graph queries directly against physical data representations (e.g., relational, XML, or RDF), leading to queries that are difficult to express and expensive to evaluate. We address these problems through a high-level query language tailored for expressing provenance graph queries. The language is based on a general model of provenance supporting scientific workflows that process XML data and employ update semantics. Query constructs are provided for querying both structure and lineage information. Unlike other languages that return sets of nodes as answers, our query language is closed, i.e., answers to lineage queries are sets of lineage dependencies (edges) allowing answers to be further queried. We provide a formal semantics for the language and present novel techniques for efficiently evaluating lineage queries. Experimental results on real and synthetic provenance traces demonstrate that our lineage based optimizations outperform an in-memory and standard database implementation by orders of magnitude. We also show that our strategies are feasible and can significantly reduce both provenance storage size and query execution time when compared with standard approaches.

1. INTRODUCTION
Scientific results are often based on complex data analysis pipelines that integrate multiple domain-specific applications [14]. Automating the use of these applications is frequently performed using traditional scripting languages, and more recently, through scientific workflow systems (e.g., [20, 27, 9]). An advantage of scientific workflow systems over traditional approaches is their ability to automatically record the provenance of intermediate and final data products generated during workflow execution. This provenance information generally consists of data and process dependencies introduced during a workflow run, and is crucial for enabling scientists to more easily understand, reproduce, and verify scientific results [13].

Scientific workflow provenance is typically represented using data and process dependency (i.e., causal) graphs [19]. The ability to effectively store and query large numbers of dependency graphs remains a significant challenge in managing provenance information [13], since provenance graphs are typically heterogeneous (i.e., they do not share a common “schema”) and each such graph may require considerable storage space. In particular, provenance information may be many times larger than the workflow and input data itself [11], workflows are often executed multiple times over different data sets and parameter settings, and most scientific research projects consist of many distinct workflows [4, 2]. To help address these challenges, a number of approaches have recently been proposed for efficiently representing and storing provenance dependency graphs [11, 16, 2, 5].

Approaches for querying stored provenance information, however, are largely still based on underlying physical data representations [13] (e.g., relational, XML, or RDF schemas), where users express provenance queries through corresponding query languages (i.e., SQL, XQuery, or SPARQL). Typical provenance queries (e.g., see [24]) are exploratory and involve finding some or all of the data and process dependencies that led to the creation of one or more data products. Posed over dependency graphs, these queries require the computation of transitive closures as well as applying selection conditions on corresponding lineage paths. For most users, expressing these queries against the physical schemas used to represent provenance information requires considerable expertise and is cumbersome even for simple queries [13]. The complexity of these queries also presents a number of challenges for efficient query evaluation [15].

In this paper we present a high-level query language for provenance (QLP; pronounced “clip”) and novel evaluation techniques to address these challenges. Our query language provides basic constructs that are tailored to querying provenance information, which can help make common provenance queries easier to express for non-experts. In addition, unlike other standard languages that return sets of nodes (e.g., [10, 17, 29, 12, 6]), QLP is closed under lineage relationships. That is, answers to lineage queries are sets of lineage dependency edges forming provenance subgraphs. This approach simplifies the expression of complex queries, better supports provenance view definitions and graph visualization [13, 7], and leads to query optimizations described in this paper.

QLP employs a general model of provenance to support a wide
range of scientific workflow systems. Most existing approaches for representing provenance do not consider workflow computation models that work over structured data, including XML. These standard provenance representation schemes (e.g., [19, 25, 17, 5, 6] among others) largely assume that workflow models are based on transformation semantics, where each workflow step consumes all input data and produces entirely new output data. Alternatively, workflow models that work over structured data (e.g., [7, 23, 26, 27]) often employ update semantics, where only a portion of an incoming XML structure is modified by each workflow step. Using update semantics can lead to a number of benefits for workflow design and reuse [3]. However, standard provenance approaches often cannot represent dependency relationships correctly or efficiently for XML-structured data [2, 23]. The model of provenance used by QLP extends conventional approaches by explicitly supporting workflow steps that process XML data and employ update semantics [3], and QLP includes query constructs for accessing both the structure and lineage of data items. More generally, QLP can be used in a wide range of situations where data is structured into nested collections and dependencies are defined among data nodes.

**Contributions.** This paper extends our prior work on efficiently storing [2] and querying [3] scientific workflow provenance over standard relational database technology. The main contributions include: (i) a formal semantics for QLP constructs introduced in [3]; (ii) techniques for evaluating QLP constructs; (iii) query optimization strategies that leverage the lineage-graph reduction approaches presented in [2]; (iv) novel algorithms for evaluating lineage queries between sets of nodes of increasing path length; and (v) experimental results verifying the effectiveness of our optimization techniques using real-life and synthetic provenance traces. In addition, we show that it is possible to simultaneously reduce the amount of storage required to represent scientific workflow provenance without negatively affecting query performance using our techniques. Our results show that unlike using standard approaches (which are often based on in-memory implementations, e.g., [29, 15, 21]), answering QLP queries using our optimization techniques over a relational database system is both feasible and scales with the size of provenance information and query complexity. While targeted at provenance applications, our optimizations can also be used in more general settings to efficiently answer a broad range of path queries over labeled, acyclic digraphs.

**Organization.** This paper is organized as follows. Section 2 introduces our provenance model and provides an overview of QLP through examples. Section 3 gives a formal semantics for QLP query constructs. Section 4 then describes different approaches for evaluating QLP lineage queries against various strategies, including our proposed techniques. Section 5 describes our experimental evaluation of the QLP optimization techniques presented in Section 4, considering both real and synthetic scientific workflow provenance traces. Section 6 discusses related work, and we conclude in Section 7.

## 2. PROVENANCE MODEL AND QLP

Consider the workflow in Fig. 1(a) showing a straightforward XML-based implementation of the fMRI image processing pipeline used in the First Provenance Challenge [19]. We refer to steps in the workflow as **actors** that are **invoked** over input data supplied by previous steps. This workflow takes a set of anatomy images representing 3D brain scans and a reference image, and applies the actors in Fig. 1(a) as follows. (1) **AlignWarp** is invoked over each anatomy image to produce a set of “warping” parameters; (2) **Reslice** is invoked over each set of warping parameters to transform the associated anatomy image; (3) **Softmean** averages the transformed images into an atlas image; (4) **Slicer** produces three different 2D slices of the atlas; and (5) **Convert** is invoked over each 2D slice to create a graphical image.

---

1In general, we also consider more complex workflow graphs involving branching, merging, and loops, as in [20, 2, 7].
In this implementation of the workflow, each invocation of an actor receives an XML structure, performs an update on a portion of that structure, and then sends the updated version of the structure to downstream actors (see Fig. 1b). Here we assume that each XML structure denotes an unranked, labeled ordered tree representing workflow data products, each tree node is assigned a unique identifier, and tree nodes represent either collection tokens or data tokens (which wrap complex objects or reference external data, e.g., stored within a file). A collection token may be an internal node (for non-empty collections) or a leaf node (for empty collections), whereas data tokens are leaf nodes only.

Fig. 1(b) shows the first invocation of each actor for a typical run of the workflow. The invocation of the AlignWarp actor (shown as AlignWarp:1) modifies the first AnatomyImage collection (node 2), and replaces its contents with a WarpParamSet data token (node 11). Similarly, the invocation of the Reslice actor uses this WarpParamSet to generate a new Image and Header data token (nodes 13 and 14, respectively). As shown, explicit “fine-grained” data dependencies are represented as part of the provenance of a run for cases when only a portion of an input data structure is modified by an invocation.

For example, the dashed arrow from node 11 to node 2 in Fig. 1(b) states that the WarpParamSet was created from the AnatomyImage collection by the first invocation of AlignWarp. Note that implicitly, node 11 depends on each of the descendents of node 2 (which includes nodes 6-10 in the figure). Similarly, each descendent of a collection implicitly inherits the dependencies of its ancestors (unless otherwise given). In our example, node 13 is a descendent of node 12 (a ReslicedImage collection), and thus implicitly depends on node 11. Taken together, Fig. 1(b) denotes a portion of the trace for a run of Fig. 1(a); in particular, this trace shows only the information associated with the first invocation of each workflow actor.

Traces can be used to derive conventional data and process dependency graphs (as views), as shown in Fig. 1(c) and 1(d), respectively. Here, the data dependency graph only shows dependencies for data nodes in Fig. 1(b), whereas the invocation dependency graph shows the complete set of invocations, assuming that four anatomy images were supplied to the workflow and three slices were created. Note that while the trace can be used to infer the data and invocation dependencies, the full trace cannot be reconstructed from these two graphs alone.

QLP queries are posed against provenance traces using the constructs outlined in Table 1. These constructs were chosen based on the common types of provenance queries identified in the literature [22, 13, 24, 8] (also see [3]). Different QLP constructs are used to query over distinct dimensions of the trace (see Fig. 1) representing: (i) lineage relations among nodes and invocations; (ii) flow relations among input and output data structures of invocations; and (iii) structural relations among nodes within and across data structures. We describe each of these dimensions below, using Fig. 1 as an example.

Queries over Lineage Relations. QLP lineage queries are closed under lineage relations such that a lineage query takes as input a set of lineage relations (edges) \( L \) and returns a subset of these lineage edges as output. A lineage edge in \( L \) is of the form \((n_1, i, n_2)\) stating that a node \( n_1 \) was used to derive a node \( n_2 \) via invocation \( i \). In this way \( L \) denotes the “derived” relation such that each edge in \( L^{-1} \) is a “dependency”. For example, consider a lineage edge \((2, \text{AlignWarp:1, 11})\) of Fig. 1(b) stating that node 2 was used by the first invocation of the AlignWarp actor to produce node 11 (i.e., node 11 depends on node 2). Sets of lineage relations define lineage graphs, and QLP lineage queries act as filters over these lineage relations. For example, consider the following QLP lineage queries over the trace of Fig. 1(b).

\[
\begin{align*}
* \text{ derived } 19 & \quad (1) \\
6 \text{ derived } * & \quad (2) \\
* \text{ through Slicer:1 derived } * & \quad (3)
\end{align*}
\]

These queries return (1) lineage relations denoting the set of paths starting from any node and ending at node 19, (2) lineage relations denoting the set of paths starting at node 6 and ending at any node, and (3) lineage relations denoting the set of paths that go through the first invocation of the Slicer actor. QLP supports both a descriptive form (used above) and a shorthand notation. Using the shorthand notation, the above queries can be equivalently expressed as: “*..19”, “6..*”, and “..\#Slicer:1..*” (or simply “*..\#Slicer:1” or “\#Slicer:1..*”). Nodes and invocations form steps within lineage queries (generalized to XPath expressions and actor names below), and lineage queries consist of two or more steps.

Queries 1-3 select lineage paths of length one or greater (referred to as transitive paths). Although not shown in Table 1, QLP also supports queries that select lineage paths having length one (i.e., immediate paths) using the “*” operator, or in the descriptive form, using the operators 1\_derived (for nodes) and 1\_through (for invocations). Immediate and transitive path operators can be combined in QLP. For example, the following query returns the lineage relations defining dependency paths that start at an input node of the first invocation of the AlignWarp actor.

\[
* 1\_through \text{AlignWarp:1 derived } *
\]

Finally, lineage queries can be chained together, e.g., the following queries

\[
\begin{align*}
2 \text{ derived } 12 \text{ derived } 15 & \quad (5) \\
2 \text{ through Reslice:1 through Slicer:1 1\_derived } * & \quad (6)
\end{align*}
\]

return (5) lineage relations denoting paths that start at node 2 and end at node 15 while passing through node 12, and (6) lineage relations that start at node 2 and end at an output node of the first invocation of Slicer while passing through the first invocation of Reslice.

Queries over Flow Relations. QLP also allows lineage graphs to be filtered based on specific versions (or occurrences) of nodes within a trace using the @in and @out operators. For example, the following queries

\[
\begin{align*}
* \text{ @in derived } 19 & \quad (7) \\
6 \text{ derived } * \text{ @out } & \quad (8) \\
* \text{ @in Slicer:1 derived } * & \quad (9) \\
15 \text{ @in Slicer:1 derived } * & \quad (10) \\
15 \text{ @out Slicer:1 derived } * & \quad (11)
\end{align*}
\]

return (7) lineage relations denoting paths that start at a node in the input data structure provided to the workflow run and end at node 19, (8) lineage relations denoting paths that start at node 6 and end at a node in an output structure of the workflow run, (9) lineage relations denoting paths that start at a node in an input structure of the first invocation of Slicer, (10) lineage relations denoting paths that start at the node at the occurrence of node 15 in the input of the first invocation of Slicer, and (11) lineage relations denoting paths that start at the occurrence of node 15 in the output of the first invocation of Slicer. Note that queries 1 and 7 (and similarly, queries 2 and 8) are not equivalent for all traces. In particular, query 7 only returns lineage graphs with paths starting at an input node of the workflow run, whereas query 1 may contain paths starting at an intermediate node (e.g., a node created by an invocation, but without data depen-
Table 1: Summary of basic QLP constructs and corresponding descriptive notations

<table>
<thead>
<tr>
<th>Construct</th>
<th>Descriptive Form</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n, s, t$</td>
<td>Node expressions $e_n$ as a single node $n$, XPath expression $s$, or set of trace nodes $t$.</td>
<td></td>
</tr>
<tr>
<td>$\theta, a$</td>
<td>Invocation expressions $e_{\theta}$ as an invocation $\theta$ or actor $a$ (denoting the set of $a$ invocations).</td>
<td></td>
</tr>
<tr>
<td>$e_i \text{ @in } e_r$</td>
<td>Nodes of $e_i$ input to invocations of $e_r$. If $e_i$ is not given, then the nodes of $e_r$ input to the workflow run.</td>
<td></td>
</tr>
<tr>
<td>$e_i \text{ @out } e_r$</td>
<td>Nodes of $e_r$ output by invocations of $e_i$. If $e_i$ is not given, then the nodes of $e_r$ output by the workflow run.</td>
<td></td>
</tr>
</tbody>
</table>

Lineage-preserving path queries (examples)

- $\star$ derived $e_r$: The lineage graph for nodes in $e_r$.
- $e_i$ derived $\star$: The lineage graph for all nodes derived from nodes in $e_i$.
- $e_i \text{ derived } \star$: The lineage graph containing all paths from nodes in $e_i$ to nodes in $e_r$.
- $e_i \text{ derived } \star$ through $e_r$: The lineage graph containing all paths from nodes in $e_i$ to nodes in $e_r$ that pass through an invocation in $e_r$.

Functions over path queries

- $\text{exists}(p)$: True if the trace contains a path defined by path query $p$.
- $\text{invocations}(p)$: The invocations of the lineage graph returned by path query $p$.
- $\text{actors}(p)$: The actors of invocations of the lineage graph returned by path query $p$.
- $\text{nodes}(p)$: The nodes of the lineage graph returned by path query $p$.
- $\text{output}(p)$: The sink nodes of the lineage graph returned by path query $p$.

Queries over Structural Relations.

In addition to specific node identifiers, QLP queries can also contain XPath expressions for accessing nodes based on their type (i.e., tag name) and parent-child relationships. For example, consider the following queries.

* derived //AtlasImage/*
/AnatomyImage[@modality="speech"]/* derived * (12)

These queries return (12) lineage relations denoting paths that end at a named or node of an AtlasImage collection, and (13) lineage relations denoting paths that start at a descendant node of an AnatomyImage collection having the value “speech” for the modality metadata attribute. QLP also allows actor names to be used in place of specific invocations. In this case, the actor name denotes the set of invocations of the actor within the trace, returning lineage relations denoting paths that pass through one of the actor’s invocations. Similar to attributes in XPath, we also allow invocation expressions to be selected based on their parameters, e.g., the expression ‘Slicer[@x=’0.5’]’ selects invocations of Slicer in which the parameter $x$ is set to “0.5”.

Queries over Each Dimension. Each of the above dimensions can be combined into a single query. For example, the following query returns the set of lineage relations denoting paths that end at a descendant node of an AtlasImage collection output by a Slicer invocation.

* derived //AtlasImage//[@out] Slicer (14)

Combined queries can be (naively) evaluated by (i) obtaining the structures resulting from @in and @out version operators, (ii) applying XPath expressions to these structures, and (iii) applying invocation queries to the resulting nodes. For example, when applied to the portion of the trace shown in Fig. 1(b), query 14 is evaluated by: (i) obtaining the output structure of the Slicer invocation; (ii) executing the XPath query ‘//AtlasImage//’ over the structure obtained in (i), returning nodes 16–19; and (iii) issuing a separate lineage query for each node, i.e., ‘* derived 16’, ‘* derived 17’, ‘* derived 18’, and ‘* derived 19’, where the answer contains the unique set of resulting lineage relations.

Additional Functions over Lineage Relations. Table 1 gives a number of additional functions that can be applied to the results of QLP lineage queries. The exists function determines whether a trace contains a given path; invocations and actors for a set of lineage relations the invocations and actors, respectively; nodes returns the nodes contained in a set of lineage relations; and input and output return the source and sink nodes in a set of lineage relations, respectively.

The following section provides a more detailed and formal treatment of the constructs presented here, and QLP evaluation techniques are presented in Section 4.

3. QLP SYNTAX, AND SEMANTICS

Provenance model. We consider the following QLP provenance model. A workflow trace $T = (V, F, L)$ consists of:

(i) a set of vertices $V = D \cup I$, where $D$ is a set of XML data structures defined over nodes $N$, and $I$ is a set of invocations over actors $A$;

(ii) a set of flow edges $F = F_{in} \cup F_{out}$, where $F_{in} \subseteq D \times I$ and $F_{out} \subseteq I \times D$ denote invocation inputs and outputs, respectively; and

(iii) a set of lineage edges $L \subseteq N \times I \times N$ denoting an acyclic digraph, where $(n_i, i, n_2) \in L$ states that invocation $i$ used $n_1$ to create $n_2$.

An XML data structure $d \in D$ consists of nodes $n \in N$ such that nodes$(d)$ are the nodes of $d$. A structure $d$ is similar to a snapshot (or version) of an overall XML document denoting the output and/or input of each workflow step, e.g., Fig. 1(b) contains six such data structures $d_1$–$d_6$. We write $A(i)$ to denote the actor associated with an invocation $i \in I$ and $A(a)$ to denote the set of invocations associated with an actor $a \in A$. The input and output structures of a trace $T$ (i.e., of the corresponding workflow run) are defined as:

\[
\text{in}(T) = \{ d \in D | \neg \exists i \in I : \langle i, d \rangle \in F_{in} \} \\
\text{out}(T) = \{ d \in D | \neg \exists i \in I : \langle d, i \rangle \in F_{out} \}.
\]

QLP Syntax. QLP (shorthand) path expressions $p$ are built from the following grammar.

\[
p ::= s \cdot s \mid s \cdot s \cdot s \cdot s \cdot s \cdot p \mid s \cdot p \\
s ::= e_n \mid e_n q \mid e_i \\
e_n ::= n \mid x \mid * \\
e_i ::= #a \mid #i \\
q ::= @in \mid @out \mid @in e_i \mid @out e_i
\]

A path $p$ is composed of two or more steps $s$. A step is either a node expression $e_n$ or an invocation expression $e_i$. A node expression
can be either an XPath x, a wildcard * or a single node identifier n. An
invocation expression specifies either an actor #a denoting all invocations of
a, or an invocation identifier #t. Node expressions are optionally qualified with
either an @in or @out operator. A simple path consists of exactly two steps in which
s1, s2 is referred to as an immediate simple path and s1 s2 a transitive simple path.

QLP Semantics. Let T = (V, F, L) be a trace defined over nodes
N, invocations I, and actors A. Each individual step s is defined as a
collection of edges T to sets of vertices such that s(T) returns a subset of V, as shown in Fig. 2. For XPath expressions x and XML structures d, in the normal way, x(d) returns those nodes in d satisfying the expression x.

We define QLP path expressions p as functions that take traces T = (V, F, L) and lineage graphs L ⊆ L to and return the given trace together with a new lineage graph Lp. Specifically, if p is a path expression, T = (V, F, Lp) is a trace, and L is a lineage graph, p(T, L) = (T, Lp) such that Lp ⊆ L. Simple immediate path expressions of the form s1 s2 for traces T and lineage graphs L are defined in Fig. 3. For convenience we generally omit T when writing steps (as in Fig. 2) and paths (as in Figs. 3, 4, and 5).

Before defining more complex path expressions, we first consider the exists operator in Fig. 4, which returns true if a given lineage graph contains a path p and false otherwise. Checking whether a path s1...s2 exists in L (i.e., the second definition in Fig. 4) employs recursion: a path exists if an immediate path exists between s1 and s2 or else if there is a node n such that an immediate path exists from s1 to n and a transitive path exists from n to s2.

The last two definitions in Fig. 4 consider the general case of paths of length m > 2. As shown, invocation expressions are treated as a special case since in general it is difficult to replace an invocation with a corresponding node (without introducing a number of additional conditions). For example, consider a lineage graph comprising of two edges (n1, i1, n2) and (n2, i2, n3), and the 3-step query ∃(n1...i1...n2). Replacing i1 in the query with its input n1 results in the two subqueries ∃(n1,...,n1) and ∃(n1,...,n2).

As a convention, invocations are written as I, denoting in this case the first invocation of actor a.
the following Datalog program can be used to compute the set of lineage relations \((x,y,z)\) between any two nodes \(N_1\) and \(N_2\).

\[
\begin{align*}
Q(N_1,N_2,N_1,I,N_2) & :- \text{Edge}(N_1,I,N_2). \\
Q(N_1,N_2,N_1,I,N) & :- \text{Edge}(N_1,I,N), \text{Path}(N,N_2). \\
Q(N_1,N_2,X,I,Y) & :- \text{Edge}(N_1,I,N), \text{Q}(N,N_2,X,I,Y). \\
\text{Path}(N_1,N_2) & :- \text{Edge}(N_1,I,N_2). \\
\text{Path}(N_1,N_2) & :- \text{Path}(N_1,N), \text{Edge}(N,N_1,N_2).
\end{align*}
\]

In the above query, Path is the transitive closure of the Edge relation. Query \(Q\) computes lineage relations as path nodes are traversed by recursively performing a breadth-first search (using both \(\text{Path}\) and \(\text{Q}\)).

Here, we propose a more efficient approach in which we (i) compute the set of nodes on paths starting at node \(N_1\) and ending at node \(N_2\), and then (ii) join the resulting path nodes with the Edge table to return the corresponding lineage edges. The following Datalog program implements this approach, which returns the same set of edges as the previous program. In general, we adopt this query approach when considering the immediate (edge) storage strategy, which we denote as storage strategy \(I\).

\[
\begin{align*}
Q_I(N_1,N_2,X,I,Y) & :- \text{PathNode}(N_1,N_2,X), \\
\text{PathNode}(N_1,N_2,Y) & :- \text{Edge}(X,I,Y).
\end{align*}
\]

Note that this approach has a single recursive rule (\(\text{Path}\)) whereas the previous approach has two recursive rules (\(\text{Path}\) and \(\text{Q}\)).

**Immediate Edges and Dependency Closure (IC).** The recursion in the previous approach can be removed by materializing the Path relation. Thus, this strategy (denoted IC) stores both the immediate edges in an edge table \(\text{Edge}(N_1,I,N_2)\) and the transitive closure over dependency nodes \(\text{Path}(N_1,N_2)\). To compute the set of nodes on lineage paths (PathNode), this approach must perform an expensive self-join on a potentially large Path table. In addition, to return a set of lineage relations, a join must still be performed between PathNode and the Edge table. Note that this approach increases storage size by materializing Path (at a cost of \(O(n^2)\)) to reduce query time by eliminating recursion.

**Immediate Edges and Closure via Pointers (ICP).** To address the issue of performing a self-join on large, materialized Path tables, we employ a “pointer-based” strategy (denoted here as ICP) that partitions the Path table into three smaller tables (based on reduction techniques) where the original Path table can be obtained by joining these smaller tables together. This pointer-based storage strategy is discussed in our earlier work [2]. In particular, the Path table is partitioned into the following three tables (see Fig. 6)

- \(\text{Node}(N_2,I,P_{\text{dep}},P_{\text{depc}})\)
- \(\text{DepV}(P_{\text{dep}},N_1)\)
- \(\text{DepcV}(P_{\text{depc}},P_{\text{dep}})\)

where \(P_{\text{dep}}\) is a pointer to the set of immediate dependencies of \(N_2\), and \(P_{\text{depc}}\) is a pointer to a set of immediate dependency pointers \(P_{\text{dep}}\) representing the transitive dependency closure of \(N_2\). Using this approach, the lineage graph shown in Fig. 6(a) is reduced to the graph in Fig. 6(b).

The basic idea of the approach is to use pointers to store a single copy of shared sets of dependency nodes. For instance, in Fig. 6(a), the dependency set for node 16 is \{13, 14\} and the dependency set for node 17 is also \{13, 14\}. Instead of storing multiple copies of the same dependency set, a pointer &9 is used to refer to the dependency set \{13, 14\} in the DepV relation, and nodes 16 and 17 refer to pointer &9 in the Node relation. In addition, instead of storing the transitive closure between dependency nodes, DepcV stores the transitive closure of the immediate pointers, e.g., as shown by the dashed lines in Fig. 6(b). For instance, the transitive dependency closure of node 16 is the set of nodes \{6, 7, 9, 10, 11, 13, 14\}, whereas in the pointer-based representation, node 16 has the transitive dependency pointer &9 that refers to the transitive dependency (pointer) set \{&1, &5, &9\} in the DepcV relation.

The following rules can be used to reconstruct a node-based lineage graph from a pointer-based (reduced) lineage graph.

\[
\begin{align*}
\text{Edge}(N_1,I,N_2) & :- \text{Node}(N_2,I,P_{\text{dep}},P_{\text{depc}}), \text{DepV}(P_{\text{dep}},N_1). \\
\text{Path}(N_1,N_2) & :- \text{Node}(N_2,I,P_{\text{dep}},P_{\text{depc}}), \text{DepcV}(P_{\text{depc}},P_{\text{dep}}), \text{DepV}(P_{\text{dep}},N_1).
\end{align*}
\]

As described in [2], both DepV and DepcV are further reduced by factoring out common subsets of dependency sets. In this way, DepV and DepcV are simple views defined over the further reduced tables.

Note that these Edge and Path queries can be used with PathNode defined above to answer QLP path expressions of the form...
Algorithm 1 Evaluating single node-based path expressions (ICP)

Input: \( p = n_1 \ldots n_2 \ldots n_m \)

Output: Set of lineage relations

1: for \( i \leq m - 1 \) do
2: \( N_p = \emptyset \)
3: \( N_p = N_p \cup \{ n \mid \text{PathNode}(n_i, n_{i+1}, n) \} \)
4: \( \) "no lineage path for \( p^* \)"
5: end if
6: \( (p) \)
7: for \( i \leq m - 1 \) do
8: \( N_p = N_p \cup \{ n \mid \text{Path}(n_i, n_{i+1}, n) \} \)
9: \( \) "no lineage path for \( p^* \)"
10: end if
11: \( N_p \)
12: \( \text{Edge}(x, y) \)
13: \( \) "lineage edge \( \star \)"
14: return \( \{ (x, y) \mid x < N_p, y < N_p, \text{Edge}(x, y) \} \)

Evaluating Simple Reachability Queries. To answer reachability queries between two nodes \( n_1 \) and \( n_2 \), i.e., \( \exists(n_1 \ldots n_2) \), we simply evaluate the query \( \text{Path}(n_1, n_2) \).

Evaluating Single Node-Based Path Expressions. We evaluate general path expressions \( p = n_1 \ldots n_2 \ldots n_m \) using ICP by transforming the expression into a set of simple path expressions \( n_1 \ldots n_{i+1} \) for \( 1 \leq i \leq m \). If each simple path is reachable, then we find the set of edges for each simple path, and compute the set of edges over these nodes (see Algorithm 1).

Naive Evaluation of Set-Based Path Expressions (ICP(N)). A straightforward approach for evaluating set-based path expressions \( p = N_1 \ldots N_2 \ldots N_m \) is to rewrite \( p \) into multiple node-based path expressions \( n_1 \ldots n_m \) for each \( n_i \in N_i \). Each of these node-based path expressions is evaluated in a similar way as in Algorithm 1, i.e., for each node-based path expression we add the corresponding set of nodes returned to \( N_p \) and then compute the set of edges as in step 10 (i.e., computing edges is deferred until each of the node-based expressions is computed). Note, however, that in this approach the number of node-based path expressions that must be computed from \( p \) is \(|N_1| \times |N_2| \times \ldots \times |N_m| \). This node-based approach for evaluating path expressions is referred to as ICP(N).

Algorithm 2 Efficiently evaluating set-based path expressions (ICP(S))

Input: \( p = N_1 \ldots N_2 \ldots N_3 \ldots N_m \)

Output: Set of lineage relations

1: for \( i \leq m \) do
2: \( N_p = \{ n \mid \text{Path}(n_i, n_{i+1}, n) \} \)
3: \( N_p = \{ n \mid \text{Path}(n_i, n_{i+1}, n) \} \)
4: \( \) "forward lineage nodes \( \star \)"
5: end if
6: \( \) "backward lineage nodes \( \star \)"
7: for \( j \leq i \) do
8: \( P_{N_j} = P_{N_i} \cap N_1 \)
9: \( \) "pruned set \( \star \)"
10: for \( i + 1 \leq j \leq m \) do
11: \( P_{N_j} = P_{N_i} \cap N_1 \)
12: end if
13: if \( P_{N_k} = \emptyset \) then
14: \( \) "no lineage path for \( p^* \)"
15: end if
16: for \( \) "forward lineage edge \( \star \)"
17: for \( i \leq m \) do
18: \( P_{N_j} = \{ n \mid \text{Path}(n_i, n_{i+1}, n) \} \)
19: \( P_{N_j} = \{ n \mid \text{Path}(n_i, n_{i+1}, n) \} \)
20: end if
21: \( N_p = \emptyset \)
22: for \( i \leq m \) do
23: \( P_{N_j} = P_{N_i} \cap N_1 \)
24: end for
25: return \( \{ (x, y) \mid x < N_p, y < N_p, \text{Edge}(x, y) \} \)

Efficient Evaluation of Set-Based Path Expressions (ICP(S)). To avoid the cost of computing a potentially large number of node-based path expressions, we propose a more efficient approach that directly computes set-based path expressions (referred to as ICP(S)). For simple paths \( N_1 \ldots N_2 \), we first evulate \( N_1 \ldots N_2 \) as follows: (i) compute the set of pointers \( P_{dep} \) for the set of nodes \( N_1 \) from the DepV relation; (ii) from the set \( P_{dep} \) we select the set of transitive pointers \( P_{dep} \) from \( P_{dep} \); (iii) from Node relation we select all nodes \( N_p \) (i.e., the “forward” lineage nodes) that have \( P_{dep} \in P_{dep} \); and (iv) we add only those nodes from \( N_1 \) to the nodes obtained in (iii) that have a \( P_{dep} \) associated with them. We then use a similar approach to evaluate the expression \( * \ldots N_2 \) resulting in a set of nodes \( N_p \) (the “backward” lineage nodes). The set of nodes \( N_1 \) on the lineage path from \( N_1 \) to \( N_2 \) is \( N_1 \cap N_2 \). Finally, the set of lineage edges is computed from \( N_1 \).

Given a path expression of the form \( p = N_1 \ldots N_2 \ldots N_m \) it may be the case that not all the nodes in the set \( N_1 \) share a lineage relationship with all the nodes in the other set. In ICP(S), we prune each of the sets \( N_i \) giving a new set \( P_{set} \) such that each node \( n_i \in N_i \) shares a dependency relationship with at least one node in all other sets, and vice versa. We rewrite \( p \) to \( P_{set} \ldots N_2 \ldots N_m \) and evaluate each simple path \( P_{set} \ldots N_2 \ldots N_m \) as in Algorithm 2. Lines 1–4 compute “forward” and “backward” lineage nodes for each set \( N_i \). Lines 5–16 prune each set \( N_i \) giving \( P_{set} = \cap_{j=1}^{i-1} N_j \cap \cap_{i}^{m} N_j \) such that all nodes in the pruned sets \( P_{set} \) are reachable from at least one node in each of the other sets. Lines 17–20 compute “forward” and “backward” lineage nodes for pruned sets \( P_{set} \). Lines 21–24 evaluate lineage nodes \( N_p \) for path expressions over pruned sets.

Figure 6: Lineage graph with corresponding pointer-based representations.
Finally, line 25 computes lineage edges from the lineage nodes.

To help see how this approach works, consider a QLP path expression query \( p = N_1 \ldots N_3 \) for the lineage graph shown in Fig. 7(a), where \( N_1 = \{1, 2, 3\} \) (red color), \( N_2 = \{4, 5, 6\} \) (blue color), and \( N_3 = \{7, 8, 9\} \) (green color). Fig. 7(c–g) shows the various stages of Algorithm 2 for evaluating set-based path expressions. Fig. 7(c) shows all those nodes that are reachable from the nodes in each sets \( N_i \). Fig. 7(d) selects those nodes from sets \( N_i \) that all those nodes are reachable from at least one node in all other sets. Fig. 7(e) shows the pruned sets \( P N_i \), where \( P N_1 = \{1, 2\} \), \( P N_2 = \{6\} \), and \( P N_3 = \{8, 9\} \), such that each node in \( P N_i \) is reachable from at least one node in other sets. Fig. 7(f) shows the nodes that lie on lineage path over the pruned sets. Finally, Fig. 7(g) shows the lineage edges \((2, \ldots, 6)\) and \((6, \ldots, 9)\) returned as a result of the lineage nodes \([2, 6, 9]\).

5. EXPERIMENTAL RESULTS

Here we evaluate the efficiency and scalability of answering QLP queries over the storage strategies of Section 4 on both real and synthetic traces. Real traces were generated from existing workflows implemented within the Kepler scientific workflow system (using the extensions described in [7, 3]). Our experiments were performed using a 2.4GHz Intel Core 2 duo PC with 2 GB RAM and 120 GB of disk space. Each storage strategy used PostgreSQL to store provenance information, and our QLP parser was implemented in Java using JDBC to communicate with the provenance database.

We compare query response time and storage size using synthetic traces ranging from 100 to 6000 nodes, \( 5 \times 10^2 \) to \( 10^8 \) immediate dependencies, \( 10^3 \) to \( 10^8 \) transitive dependencies, and lineage paths of length 10 to 100, respectively. The synthetic traces were taken from [2], and represent typical lineage patterns of common scientific workflows [2, 7, 24]. Fig. 8(a) shows the complexity of dependencies (immediate and transitive) as the nodes in synthetic traces increase, and Fig. 8(b) shows the storage size of these traces under the different storage strategies. As shown, the pointer-based approach used to store immediate and transitive dependencies in a reduced form leads to considerably smaller storage size [2].

We also evaluated our approaches using the following real traces from scientific workflows implemented within Kepler: the PTP, GBL, CGR, and PLC workflows [7] use different approaches to infer phylogenetic trees from protein and morphological sequence data; the PCI workflow was used in the first provenance challenge [24] (and is similar to the example shown in Fig. 1); the STP, STM, and CYC workflows are used in characterizing microbial communities by clustering and identifying DNA sequences of 16S ribosomal RNA; the WAT workflow characterizes microbial populations by producing phylogenetic trees from a list of sequence libraries; and the PC3 workflow was used within the third provenance challenge\(^4\). The dependency annotations of these traces range from \( 10^2 \) to \( 10^5 \) with \( 2 \times 10^2 \) to \( 2 \times 10^4 \) transitive dependency edges.

We consider the following QLP queries for evaluating query response time.

\[
\begin{align*}
& n_1 \ldots n_m \\
& n_1 \ldots n_2 \\
& n_1 \ldots n_2 \ldots n_3 \\
& n_1 \ldots N \\
& N_1 \ldots N_2 \\
& N_1 \ldots N_2 \ldots N_3 \ldots N_m
\end{align*}
\]

These queries return \((Q1)\) lineage relations denoting paths that lead to a node \( n \) (e.g., to return the full lineage of \( n \)); \((Q2)\) lineage relations denoting paths starting at node \( n_1 \) and ending at node \( n_2 \); \((Q3)\) lineage relations denoting paths starting at node \( n_1 \) and ending at node \( n_3 \) that pass through node \( n_2 \); \((Q4)\) lineage relations denoting paths that lead to nodes in a set of nodes \( N \); \((Q5)\) lineage relations denoting paths starting at nodes in a set \( N \) and ending at nodes in a set \( N_2 \); and \((Q6)\) lineage relations denoting paths that satisfying path expression of varying length \( m \) (e.g., in our experiments, \( m \)

\(^4\)See [http://twiki.ipaw.info/bin/view/Challenge/](http://twiki.ipaw.info/bin/view/Challenge/)
Figure 8: (a) shows dependency complexity of sample traces. (b) shows size for storing the sample traces in storage strategies.

Figure 9: Evaluating $n \times n$ over storage strategies.

Figure 10: Evaluating $n_1 \times n_2$ over storage strategies.

Figure 11: Evaluating $n_1 \times n_2 \times n_3$ over storage strategies.
These queries are evaluated over various storage strategies (I, IC, and ICP, described in Section 4). We implemented strategy I using XSB (a logic programming and deductive database system that supports in-memory queries) whereas the implementation of IC, ICP(N), and ICP(S) uses PostgreSQL for storing and querying traces.

**Paths with Single Node Identifiers.** In queries Q1–Q3 each step evaluates to a single node identifier. Figs. 9–11 show the time to evaluate Q1, Q2, and Q3, respectively, over I, IC, and ICP. In each case, we show: (a) the total query execution time for computing lineage relations, (b) the portion of the total time used to compute the nodes that lie on the lineage path, and (c) the portion of the total time used to compute the lineage edges from the nodes returned in (b).

Fig. 9(a) shows that Q1 scales for ICP(N) but does not scale for IC and I. I scales when returning set of nodes (Fig. 9(b)), but I does not scale for computing edges from these set of nodes (Fig. 9(c)). Similar results were observed for Q2 and Q3, which have more complex path expressions. We also observe that as path expressions increase in length, query time scales for ICP(N) but not for IC and ICP. For example, for a trace containing 6000 nodes, Fig. 9 shows that ICP(N) takes 500 ms, 600 ms, and 700 ms to evaluate Q1, Q2, and Q3, respectively, representing only a marginal increase in query response time as the length of paths increases.

** Paths with Sets of Node Identifiers.** In query Q5, each step evaluates to a set of node identifiers. Fig. 12 shows the time to evaluate Q5 over the storage strategies. Fig. 12 shows the time to evaluate Q5 over various storage strategies (trace nodes ranging from 1000 to 10,000). Fig. 14(a) and (b) show the storage size for representing lineage graphs and transitive dependencies for the real traces under our storage strategies. Fig. 14(a) and (b) show the storage size for representing lineage graphs and transitive dependencies for the real traces under our storage strategies. Fig. 14(a) shows that Q1 scales for IC and ICP, which is considerably less than that of I and IC. Note that IC stores only lineage edges, while ICP stores both lineage edges and transitive closure of dependency nodes using our reduction techniques, and in most of the cases (except FTP, PC1, and PC3), the storage size for ICP is less than that of I (implying that many nodes have similar dependencies). Also, although ICP stores the same information as IC, the storage size for ICP is less than that for IC.

Storing and Querying Real Traces. Fig. 14(a) and (b) show the space requirement for storing lineage relations in ICP is considerably less than in I and IC. To consider complex queries, each step in the path expressions evaluate to all data nodes of the trace. As shown, path queries are evaluated using ICP(S) in less than $10^2$ ms, which is significantly less than using IC which can take up to $10^6$ ms. This is because IC has to perform a self-join over the transitive closure table together with a join over the dependency table, both containing a large number of tuples, whereas ICP(S) performs three joins over the reduced tables. If we...
Figure 14: (a) and (b) shows size for storing real traces in various storage strategies. (c) and (d) shows time for evaluating constructs $*..N$ and $N_1..N_2$ over these storage strategies.

compare this result with the results for the synthetic traces, we see that as the number of dependencies increases, IC will perform even worse.

**Analysis.** Our detailed experimental evaluation demonstrates that ICP not only requires less storage, but evaluating QLP queries over ICP is faster and scales with increasing complexity of traces. In particular, our optimized set-based approach $ICP(S)$ for evaluating QLP path expressions is both efficient and scalable.

6. RELATED WORK

Conventional provenance approaches (e.g., [19, 17, 5]) largely assume workflow models are based on *transformation semantics*, whereas workflow systems that work over structured data (e.g., [7, 26, 23, 27]) often employ *update semantics*, i.e., where only a portion of incoming data is modified by each workflow step. Applying conventional provenance approaches to such models results in provenance information that is either too coarse or potentially incomplete and even incorrect, which can lead to incorrect analysis of scientific results [3]. Our provenance model subsumes conventional approaches for representing workflow provenance by supporting workflow computation models that permit multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics. In prior work, we show how existing models (including OPM [19]) can be mapped to our provenance model [3].

Approaches for querying provenance are largely based on physical data representations [13] (e.g., relational, XML, or RDF schemas), where users express provenance queries through corresponding query languages (i.e., SQL, XQuery, or SPARQL). Provenance queries often require computing transitive closures over dependency relations, and expressing such queries using standard approaches is typically done using recursion or stored procedures [16, 11, 2]. Expressing such queries is both cumbersome and error-prone, and requires considerable user expertise. Instead, high-level languages such as QLP provide a separation between the logical provenance model and its underlying physical representation, which allows for the use of different representation schemes and additional optimization techniques. Our approach, in particular, automatically translates QLP queries to equivalent relational queries expressed against the provenance storage schemes described in [2]. Standard approaches for querying provenance information (e.g., [10, 17, 29, 12, 6]) return sets of nodes (either sets of data items or process invocations) as the query result. This approach requires additional steps (queries) to reconstruct causal relations among nodes within a query answer. Instead, QLP is closed under lineage relations, where answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs, and thus query results are “provenance preserving.” This approach has a number of advantages, e.g., for supporting provenance views, incremental querying, and for supporting visualization applications [13, 7].

Our approach is similar to a number of approaches for querying graph structures. In [15], a general query language based on graph grammars is presented in which queries return subgraphs. However, whereas our implementation uses a relational database system to store and query provenance graphs, the approach in [15] requires entire graph structures to be stored in main memory, and recursive patterns over graphs (e.g., similar to our ‘..’) are not supported. Similarly, semi-structured query languages (e.g., Lorel [1] and Florid [21]) often include so-called “path variables”, which can be used to query over and return paths (i.e., the nodes) between nodes. However, while a part of the query language of these systems, path variables are generally not implemented (e.g., see [1, 21]). Additionally, the approach used in QLP differs from that of path variables in that query results are sets of lineage relations that implicitly denote paths between nodes as opposed to directly returning the paths themselves. This in turn offers advantages for query composition and efficient evaluation (as described in Section 4).

Similar to QLP, vtPQL (used in VisTrails [10]) defines provenance-specific query constructs. The primary construct in vtPQL for answering lineage queries related to invocations is
upstream\((x)\), which return all modules (actors) that procede \(x\) in the workflow definition. However, vPQL assumes only a single lineage path between two such modules, and thus would return incorrect results in the case where multiple paths exist between nodes.

The QLP construct \(\exists s(1\ldots s_2)\) corresponds to a reachability query \([18, 28]\) over lineage graphs. Besides simple paths, our implementation can efficiently answer more complex reachability queries for paths \(p = s_1\ldots s_2\ldots s_m\) of length \(m > 2\) in which each step \(s\) can evaluate to a set of nodes. Thus, our implementation can also efficiently determine whether a lineage graph satisfies relatively complex path expressions.

7. CONCLUSION

We have presented a general approach for querying provenance using a novel query language (QLP) that provides specialized constructs for querying over lineage and structural relations introduced during scientific workflow runs. Answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs, which simplifies the expression of complex queries and allows query results to be easily queried and visualized. This work extends our prior work on efficiently storing [2] and querying [3] scientific workflow provenance by presenting (i) a formal semantics for QLP constructs; (ii) a set of novel query optimization techniques that leverage the lineage-graph reduction approaches presented in [2]; and (iii) a detailed experimental evaluation based on our QLP implementation that demonstrates the efficiency and scalability of our approach. We have shown that our implementation can simultaneously reduce the amount of storage required to represent scientific workflow provenance without negatively affecting query performance. We have also shown that unlike using standard approaches (often based on in-memory implementations, e.g., [29, 15, 21]), answering QLP queries using our optimization techniques over a relational database system is both feasible and scales with the size of provenance information and query complexity. Our optimizations can also be used in more general settings to efficiently answer a broad range of path queries over labeled, acyclic digraphs.

Acknowledgments. This work supported in part by NSF grants IIS-0630033, OCI-0722079, IIS-0612326, and DOE grant DE-FCO2-07ER25811.

8. REFERENCES


