# Exploring Scientific Workflow Provenance Using Hybrid Queries over Nested Data and Lineage Graphs

Manish Kumar Anand[1], Shawn Bowers[2,3], Timothy McPhillips[2], and Bertram Ludäscher[1,2]

[1] Department of Computer Science, University of California, Davis
[2] UC Davis Genome Center, University of California, Davis
[3] Department of Computer Science, Gonzaga University
{maanand,sbowers,tmcphillips,ludaesch}@ucdavis.edu

**Abstract.** Existing approaches for representing the provenance of scientific workflow runs largely ignore computation models that work over structured data, including XML. Unlike models based on transformation semantics, these computation models often employ update semantics, in which only a portion of an incoming XML stream is modified by each workflow step. Applying conventional provenance approaches to such models results in provenance information that is either too coarse (e.g., stating that one version of an XML document depends entirely on a prior version) or potentially incorrect (e.g., stating that each element of an XML document depends on every element in a prior version). We describe a generic provenance model that naturally represents workflow runs involving processes that work over nested data collections and that employ update semantics. Moreover, we extend current query approaches to support our model, enabling queries to be posed not only over data lineage relationships, but also over versions of nested data structures produced during a workflow run. We show how hybrid queries can be expressed against our model using high-level query constructs and implemented efficiently over relational provenance storage schemes.

## 1 Introduction

Scientific workflow systems (e.g., [15,7,19]) are increasingly used by scientists to design and execute data analysis pipelines and to perform other tool integration tasks. Workflows in these systems often are represented as directed graphs where nodes denote computation steps (e.g., for data acquisition, integration, analysis, or visualization) and edges represent the required dataflow between steps. Systems execute workflow graphs according to various *models of computation* [15], which generally specify how workflow steps should be scheduled and how data should be passed (and managed) between steps. In addition to automating data analyses, scientific workflow systems can capture the detailed provenance of data produced during workflow runs, often by recording the processing steps used to derive data products and the data provided to and generated by each step. Provenance recording capabilities represent a key advantage of scientific workflow technology over more traditional scripting approaches, enabling scientists to more easily understand, reproduce, and verify scientific results [9,4]. However, effectively representing provenance information is complicated by a number of

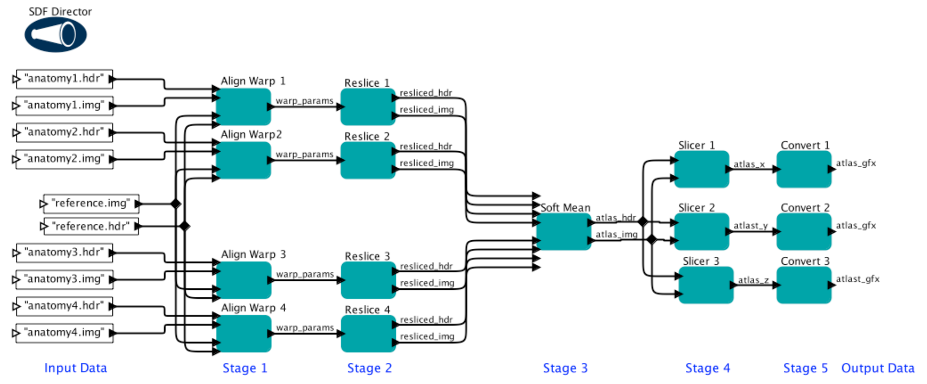


**Fig. 1.** A straightforward Kepler implementation of the first provenance challenge fMRI workflow. Given 3D brain scans (anatomy images) it: (1) compares each image to a reference image to determine "warping" parameters for alignment with the reference; (2) transforms the images according to the parameters; (3) averages the transformed images into an atlas image; (4) produces three different 2D slices of the altas; and (5) converts each 2D slice into a graphical image.

technical challenges as workflows, the models of computation used to enact them, and the structures of the data flowing through them, each become more complex.

**Design Challenges.** Consider the Kepler workflow definition shown in Fig. 1, which is a straightforward implementation of the fMRI image processing pipeline of the first provenance challenge [18] (i.e., directly following the implementation suggested in the challenge). This workflow design hardwires the number of input images (four; see Stages 1–3) and the number of output slices (three; see Stages 4–5). Consequently, performing the same computation on a different number of input images or with additional output slices will require significant modifications to the workflow definition. Simply supplying a different input data set or changing parameter values applied to the workflow is not sufficient. Instead new components (actors) and additional wires must be introduced throughout the workflow. The limitation of this design approach is even more obvious in workflows where the number of data items produced *during* a workflow run is not predetermined. For example, many standard bioinformatics analyses include workflow steps that produce *collections* of output data with indefinite cardinality (e.g., BLAST and maximum parsimony tree inference algorithms [5], among others). Chaining such steps together generally requires data to be grouped into nested collections, and workflow systems often address this need by enabling workflow authors to model data using XML-like data structures. Further, by natively supporting operations over such nested data collections (e.g., [11,20,19,5]), these systems can also yield more generic and reusable workflow designs [16].

**Design via Nested Data Collections.** Fig. 2 illustrates the advantages of employing XML-like data structures in workflows. Shown is a workflow definition with the same intent as that in Fig. 1, but implemented using the *Collection-oriented modeling and*

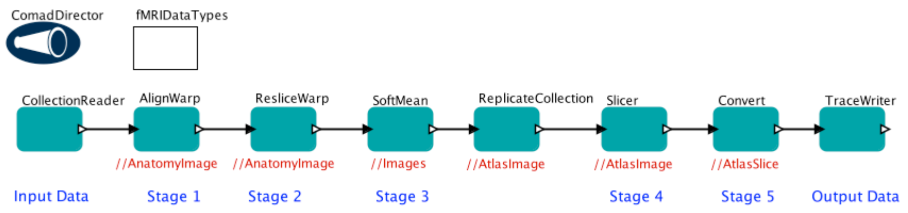


**Fig. 2.** A COMAD design of the fMRI workflow in Kepler, where: (1) CollectionReader is configured with an input XML structure that specifies the input images used; (2) AlignWarp, ResliceWarp, SoftMean, Slicer, and Convert are similar to those of Fig. 1 but work over portions of the XML data stream; and (3) ReplicateCollection is configured to create *n* copies of the resulting SoftMean images, where each copy induces a separate Slicer invocation. COMAD extends Kepler with explicit support for managing data collections, making COMAD actors "collection aware."

*design* (i.e., COMAD) paradigm [16] developed in Kepler.[1] Workflows in COMAD are executed over XML data streams in which actors employ *update semantics* by making modifications to (i.e., updating) portions of the overall XML structure and passing the updated structure to downstream actors. Specifically, actors in COMAD receive fragments of an incoming XML token stream based on their declared *read-scope* parameters (given as XPath expressions in Fig. 2), insert and remove fragments within their matching scopes, and pass the modified stream on to subsequent actors. Actors are executed (or *invoked*) over each corresponding read scope match within an incoming token stream. For instance, the COMAD implementation of the Align Warp actor (Stage 1) is invoked once over each Anatomy Image collection within the workflow input, as shown in Fig. 3a for the first invocation of Align Warp. The workflow of Fig. 2 differs from Fig. 1 in that it can be executed over multiple collections of anatomy images of varying cardinality without requiring modifications to the workflow graph. To have the workflow average five anatomy images, rather than four, the user only needs to add the additional image to the input data set. The COMAD version of the workflow also contains noticeably fewer overall actor occurrences and connections. Other advantages include support for parallel actor execution (where actors are executed concurrently over distinct portions of the overall XML structure) and the ability to easily add and remove actors within a pipeline while minimizing the changes that must be made to the workflow graph [16].

**Provenance Challenges.** While COMAD provides benefits for workflow design and execution, it requires a richer model of provenance than used by systems in which actors correspond to *data transformers* (as in Fig. 1), i.e., where actors (i) treat data as opaque objects (or *tokens*), (ii) produce a set of new output tokens from each set of input tokens they receive, and (iii) assume all input tokens are used to derive all output tokens within a single actor invocation. Provenance management systems tailored to workflows consisting entirely of data transformers (e.g., [2,21,17]) generally store the input and output tokens of each actor invocation, and later use this information to infer (causal) dependencies between data tokens and workflow steps. When applied to

[1] Kepler supports multiple computation models ( *directors*) including standard dataflow models such as Synchronous DataFlow (SDF) and Process Networks (PN) [15].

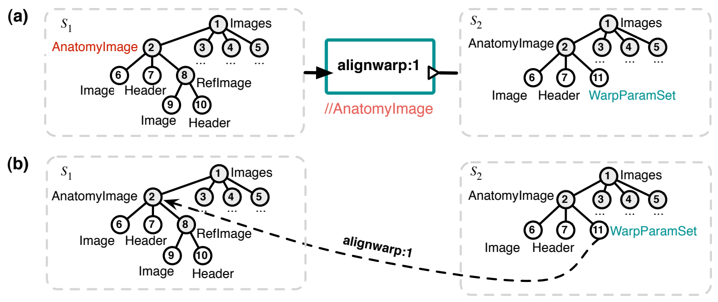


**Fig. 3.** An actor invocation employing *update semantics*: (a) example nested data structures $s_1$ and $s_2$ input to and output by the first invocation of the AlignWarp actor during a run of Fig. 2; and (b) the fine-grain data dependencies of nodes in $s_2$ on nodes in $s_1$ introduced by the invocation

computation models such as COMAD, however, this standard approach to recording and reporting provenance relationships can infer incomplete and even incorrect data and invocation dependencies [3]. For example, Fig. 3b shows the actual data dependencies introduced by the first invocation of the AlignWarp actor; specifically, the object representing the warping parameters (node 11) was derived from each of the image and header objects contained in the AnatomyImage collection matching the read scope of the invocation (node 2). However, if these input and output structures were represented as atomic data objects ($s_1$ and $s_2$), the standard model would infer only a single dependency between the output version ($s_2$) and the input version ($s_1$). Alternatively, if as in COMAD, XML structures are represented as XML token streams, then the standard model would incorrectly infer dependencies between every output and input node.

**Contributions.** We define a model of provenance (based on our prior work [3]) that extends the standard model [17,9] with support for scientific workflows that process XML data and employ update semantics. We also present approaches for querying provenance information based on this model. The model provides a *conceptual* representation of workflow provenance that is used for expressing provenance queries and for interpreting query results. A goal of our approach is to address shortcomings of current approaches for querying provenance information. Specifically, most existing approaches directly expose to users the physical representation of provenance information (e.g., using relational, XML, or RDF schemas) in which users must express provenance queries using associated query languages (e.g., SQL, XQuery, or SPARQL). These approaches are often invconvenient and difficult for users needing to express complex provenance queries, and also limit opportunities for storage and query optimization, since optimization often results in modifications to the underlying provenance representation. As an alternative, we present a *Query Language for Provenance* (QLP; pronounced "clip") that is designed to be independent of any particular physical representation, and that includes constructs tailored specifically for querying scientific workflow provenance. QLP constructs also allow queries to be expressed over both lineage information and different versions of nested data structures produced by workflow runs. We show that QLP can express common provenance queries and that these queries can be answered efficiently based on translations to standard relational database techniques.

**Outline.** Section 2 describes our provenance model for representing COMAD-style workflow runs that supports nested data and update semantics. Section 3 describes shortcomings of current approaches for querying provenance information, presents the QLP query constructs, and shows that these constructs are expressive enough to formulate common provenance queries. We also briefly describe our implementation of QLP and demonstrate the feasibility of our approach in Section 3. Section 4 discusses related work, and Section 5 summarizes our contributions.

## 2 Models of Provenance

Despite efforts to better understand and even standardize provenance models, e.g., in the Open Provenance Model (OPM) initiative [18], a universally accepted model has yet to emerge. This is due in part to the differences in the underlying models of computation employed by workflow systems. In the following, we develop a number of models of provenance, starting with a basic model capturing the conventional view of scientific workflows as simple task dependency graphs over atomic data and atomic (single invocation) processes, similar to the OPM model (as well as others, e.g., [7]). Next, we extend this basic model to handle computations where a workflow step (i.e., task or process) consists of multiple invocations (or *firings* [14]) over a stream of incoming tokens. The resulting model is a variant of *process networks* [13] that is well-suited for stream processing, and comes with "built-in" pipeline parallelism. A second, and for our purposes crucial extension of the basic provenance model, allows us to handle complex data, i.e., nested data collections, represented in XML. We discuss two variants of this XML-based provenance model, with *copy* and *update* semantics, respectively. Finally, our last extension yields a unified provenance model, incorporating the above features and adding fine-grained dependencies such as in Fig. 3b.

**Basic Model.** The conventional (or *basic*) model of provenance consists of a *trace structure* (or simply a *trace*) $T = (V, E)$ forming an acyclic *flow graph*, where each node in $V = S \cup I$ represents either an (atomic) *data structure* $s \in S$ or a *process invocation* $i \in I$. Edges $E = E_{\text{in}} \cup E_{\text{out}}$ are *in-edges* $E_{\text{in}} \subseteq S \times I$ or *out-edges* $E_{\text{out}} \subseteq I \times S$, representing the flow of data during a workflow run. A trace $T$ in this model links atomic data tokens to atomic processes (i.e., having a single process invocation). Data structures are consumed (destroyed) by an invocation via in-edges. Similarly, process invocations create *new* output structures via out-edges. Thus, in this model, processes are viewed as *data transformers*. To avoid write conflicts among multiple invocations, we require that $E_{\text{out}}^{-1} : S \to I$ be a function, associating with each output structure $s \in S$ the unique invocation $i_s \in I$ that created it. A flow graph gives rise to two natural views, a *data dependency graph* $G_d = (S, E_{\text{ddep}})$ and an *invocation dependency graph* $G_i = (I, E_{\text{idep}})$, defined by:

$$E_{\text{ddep}}(s_2, s_1) \longleftarrow E_{\text{in}}(s_1, i),\ E_{\text{out}}(i, s_2)$$
$$E_{\text{idep}}(i_2, i_1) \longleftarrow E_{\text{out}}(i_1, s),\ E_{\text{in}}(s, i_2)$$

Fig. 4a depicts a scientific workflow definition (gray boxes) and Fig. 4b shows a corresponding flow graph in the conventional model. The inferred data and invocation dependency views are shown in Fig. 4c and Fig. 4d, respectively.

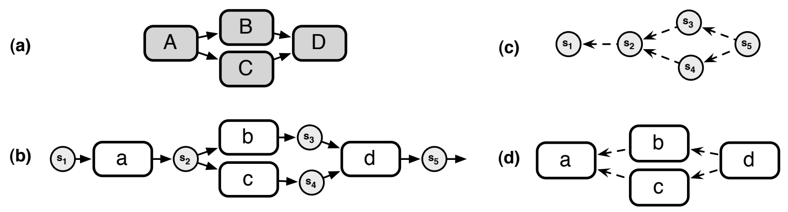


**Fig. 4.** Conventional model: (a) *workflow definition* with steps $\mathtt{A}, \ldots, \mathtt{D}$; (b) example *flow graph* $E = E_{\text{in}} \cup E_{\text{out}}$ with process invocations $\mathtt{a}, \ldots, \mathtt{d}$ and atomic data $\mathtt{s}_1, \ldots, \mathtt{s}_5$; (c) data dependencies $E_{\text{ddep}}$ and (d) invocation dependencies $E_{\text{idep}}$ inferred from the flow graph $E$ of (b)

**Open Provenance Model.** Our basic model closely ressembles the *Open Provenance Model* (OPM) [18]. In OPM, an atomic data structure $s \in S$ is called an *artifact*, an invocation $i \in I$ is called a *process*, an in-edge $s \rightarrow i$ corresponds to a *used* edge $s \overset{\texttt{used}}{\dashleftarrow} i$, and an out-edge $i \rightarrow s$ corresponds to a *wasGeneratedBy* edge $i \overset{\texttt{genBy}}{\dashleftarrow} s$. Similarly, the above dependency views $E_{\text{ddep}}$ and $E_{\text{idep}}$ simply have different names in OPM: For $E_{\text{ddep}}(s_2, s_1)$ we say in OPM that the artifact $s_2$ *was derived from* the artifact $s_1$; and for $E_{\text{idep}}(i_2, i_1)$ we say that the process $i_2$ *was triggered by* the process $i_1$.

**Multi-Invocation Model.** The basic model views processes as *atomic*, i.e., for each task $\mathtt{A}, \mathtt{B}, \ldots$ in the workflow definition, there is a *single* invocation $\mathtt{a}, \mathtt{b}, \ldots$ in the flow graph (see Fig. 4b). Other models of computation, notably process networks [13] and related dataflow variants with *firing* [14] give rise to finer-grained process models for incremental computations over data streams. Fig. 5a shows the execution of process A modeled as two *independent* invocations, a:1 and a:2, which may be executed concurrently over the input stream $\mathtt{s}_1$, $\mathtt{s}_2$, ... (similarly for B and b:1, b:2). Here, the second invocation a:2 of A does not "see" (is independent of) the earlier input $\mathtt{s}_1$ used by a:1. This is the case, e.g., if A is *stateless*, i.e., has no memory between invocations. Fig. 5b is a variant of Fig. 5a in which A is *stateful*, and thus preserves information between invocations a:*i*, resulting in additional dependencies. More formally, in the *multi-invocation model*, a trace $T = (V, E, \alpha)$ includes a function $\alpha : I \rightarrow A$ returning for each invocation $i \in I$ the *actor* $\alpha(i) \in A$ that created $i$. Conversely, for any actor $A$, $\alpha^{-1}(A) = \{i \in I \mid \alpha(i) = A\}$ is the set of invocations created by $A$ during a workflow run.

The underlying *model of computation* (MoC) of a workflow language determines the kinds of traces that can be generated at execution time. One can understand (and formalize) a MoC as a mapping that associates with a workflow definition $W$ and input $s$, a set of possible traces $T(s)$. The basic model, e.g., with its atomic data tokens and atomic processes can create flow graphs as in Fig. 4b, but not those in Fig. 5, which can support *pipeline parallel* execution.

**Nested Model (Copy Semantics).** So far we have considered data structures as atomic *tokens* $s \in S$, i.e., without further access to any internal structure (e.g., $s$ might denote a string, file, or Java object). This model is often too coarse and thus inadequate when dealing with workflows over nested data such as XML. Thus, we refine the multi-invocation model by "drilling down" to the level of data items within structures. In the nested model with copy semantics, a trace $T = (V, E, \alpha, \tau)$ includes a function $\tau : S \rightarrow X$, which maps structures $s \in S$ to XML trees $\tau(s) \in X$ such that the

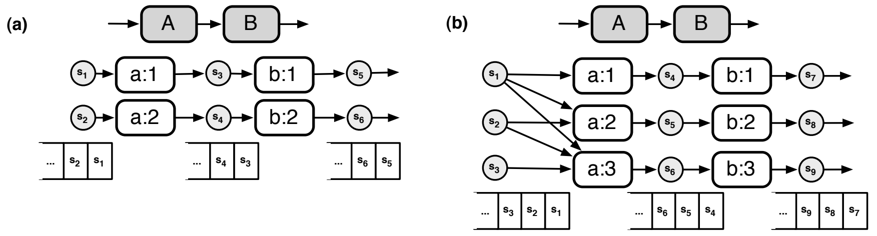


**Fig. 5.** The process network (PN) model supporting *data streaming* and *pipelined execution*: (a) step A of the workflow definition (top) is modeled as two *independent* invocations (a:1, a:2) within the flow graph (bottom), possibly executed concurrently over input stream $s_1, s_2, \ldots$; (b) a variant of (a) where A is *stateful*, preserving information between invocations a:$i$, resulting in additional dependencies (i.e., in edges)

domain $X$ of XML trees is defined as usual. In particular, we assume an underlying space of nodes $N$ from which $X$ is built. As above, we require $E^{-1}_{\text{out}}$ to be a function, i.e., for any $x \in X$ produced, there is a *single* invocation $i_x$ that produced it. This avoids *write conflicts*: no two invocations $i_1$ and $i_2$ can write to the same tree $x$ (like above for $s$). We can think of actors consuming (and destroying) their input and creating "fresh" XML trees for each invocation. Thus, if some data within $x$ must be preserved (e.g., for "pass through"-style processing), a fresh copy $x'$ of $x$ must be created in this model.

**Nested Model (Update Semantics).** Consuming XML objects and recreating parts of them (through fresh copies) can be both inconvenient and inefficient with respect to workflow execution and provenance storage. Under the *update semantics* we assume that different *versions* of trees $\tau(s) \in X$ *share* nodes from $N$ (Fig. 6a).[2] In particular, invocations are modeled as sets of *updates* that produce new versions of their input. Viewing invocations as updates can increase the concurrency of workflow execution for independent invocations. For invocations a:i and a:j of A and inputs $s \in S$, if

$$\Delta_{\mathtt{a:i}}(\Delta_{\mathtt{a:j}}(\mathtt{s})) = \Delta_{\mathtt{a:}j}(\Delta_{\mathtt{a:}i}(s)),$$

then we say that A has *independent invocations*. In Fig. 6a, A has independent invocations a:1 and a:2, i.e., applying $\Delta_{\mathtt{a:1}}$ and $\Delta_{\mathtt{a:2}}$ either in series or in parallel results in $\mathtt{s}_2$. There are different ways to achieve this independence. In COMAD, e.g., one can (i) employ non-overlapping scope expressions, and (ii) require further that actors are stateless across multiple invocations of the same actor.

As shown in Fig. 6b, we relax the earlier constraint that $E^{-1}_{\text{out}}$ be a function. Thus, in the nested model with update semantics, a trace $T = (V, E, \alpha, \tau, \gamma)$ includes two functions denoted by $\gamma$, namely, $\gamma+ : N \to I$ returns the unique invocation that created a node, and $\gamma- : N \to 2^I$ returns the possibly empty set of invocations that deleted a node.[3] This approach avoids write conflicts at the node level, while still allowing parallel updates to the same tree, e.g., as shown in Fig. 6b. Here we restrict the types of updates that can be performed by invocations to *insertions* and *deletions* of nodes, i.e., similar to COMAD, abitrary structural modifications are not considered.

[2] we can view $s$ as the version-id of $\tau(s)$.

[3] Note that for workflows containing branches, multiple invocations can delete the same node.

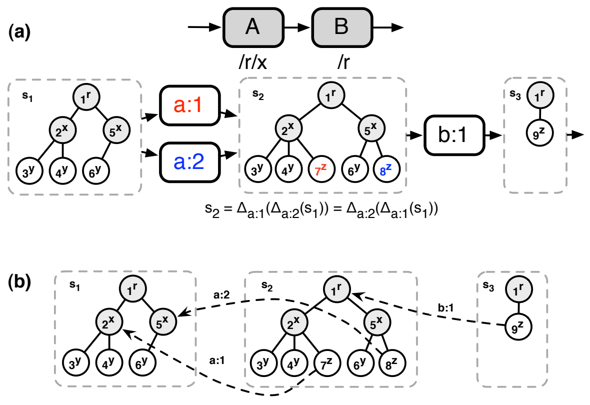


**Fig. 6.** The *unified model*, which supports COMAD and similar models of computation: (a) the complex (i.e., XML) data structure $\mathtt{s}_1$ is *updated* by invocations giving $\mathtt{s}_2 = \Delta_{\mathtt{a}:2}(\Delta_{\mathtt{a}:1}(\mathtt{s}_1))$, and for stateless $\mathtt{A}$, $\mathtt{s}_2 = \Delta_{\mathtt{a}:1}(\Delta_{\mathtt{a}:2}(\mathtt{s}_1))$; and (b) the *labeled dependency (i.e., lineage) edges* representing the correct data dependencies for the nested data structures of (d)

**Unified Model (Fine-Grained Data Dependencies).** Finally, we combine the above models into a unified trace structure supporting both multiple invocations and nested XML with update semantics. Specifically, we extend the previous model with *fine-grained dependencies* for relating nodes according to dependency relationships (as opposed to relating coarse-grained structures as in a flow graph). If a node $n$ is a fine-grained dependency of a node $n'$, then we say that $n$ was directly used in the creation (or derivation) of $n'$. We represent fine-grained dependencies using *lineage relations*, which includes the invocation $i$ that created $n'$, denoted $n \overset{\mathtt{i}}{\dashleftarrow} n'$ in Fig. 6b. Note that if $n'$ depends on a collection node $n$ within the structure $s$, then $n'$ is also assumed to depend on the descendents of $n$ with respect to $s$; and we typically show only the "highest" dependencies of a node, as in Fig. 6b.

A trace in the unified model is of the form $T = (V, E, L, \alpha, \tau, \gamma)$ where fine-grained dependencies are captured by the ternary *lineage relation* $L \subseteq N \times I \times N$. Thus, the unified model consists of three dimensions: (i) *flow relations* among input and output structures (defined by the flow graph), (ii) *parent-child relations* among nodes of structures, and (iii) *lineage relations* defining fine-grained data dependencies. In the following section we consider approaches for querying each dimension separately as well as *hybrid* approaches combining multiple dimensions in a single query.

## 3 Querying Provenance

Our goal is to provide generic support for querying provenance information to enable a wide range of users and applications. Common types of provenance queries we want to support include standard lineage queries [18,6] for determining the data and invocations used to derive other data; queries that allow users to ensure that specific data and invocation dependencies were satisfied within a run; queries for determining the inputs and outputs of invocations (e.g., based on the actors used and their parameters); and queries for determining the structure of data produced by a workflow run. Further, to address shortcomings of current approaches that either do not provide explicit query support or else use query languages not naturally suited for querying provenance, we also seek an approach that satisfies the following desiderata.

**(1) Physical Data Independence.** Many existing approaches for querying provenance information [18] are closely tied to physical data representations, e.g., relational, XML, or RDF schemas, where users express provenance queries using corresponding manipulation languages, i.e., SQL, XQuery, or SPARQL, respectively. Most provenance queries also require computing transitive closures [8,10], and thus expressing provenance queries requires users to have considerable expertise in the underlying schemas and query languages. Some systems also support *multiple* storage technologies (e.g., [2]) in which specific provenance representations are selected by workflow designers. Thus, to query a given workflow run, users must know which representation was used to store the run and be proficient in the associated query language. Instead, a generic approach for querying provenance information should allow multiple representations to be used with the same fundamental query language, and should thus be independent of the underlying physical data model.

**(2) Workflow System Independence.** Many different workflow systems record provenance information. Users should be able to express queries without having to know which workflow system was used to produce provenance information, and without having to use a different language for each system. OPM [17], e.g., is independent of any particular workflow system, whereas the Kepler provenance recorder in [2] requires users to understand the details of Kepler workflow computation models to construct basic lineage information.

**(3) Workflow Definition Independence.** It is often possible to make use of provenance information gathered during a workflow run without accessing or understanding the workflow definition. A generic query approach should allow users to query provenance information without having prior knowledge of the workflow definition, or the types of data input and output by the workflow run. Similarly, a generic query approach should make it possible for users to discover the actors that were invoked during a workflow run, and the types of data used. However, when data types and workflow definitions are known, users should be able to query provenance conveniently via this information.

**(4) Provenance Relationship Preserving.** It is often convenient to visualize provenance information using data and invocation dependency graphs [5,18,8], and these graphs are often constructed from the result of a provenance query. In many provenance approaches (including those based on path expressions [12]), typical queries return only sets of nodes or invocations, requiring additional queries to reconstruct the corresponding lineage graphs. Thus, a general approach for querying provenance information should make it simple for users and applications to construct such graphs by returning paths over lineage information.

**(5) Incremental Queries.** A consequence of preserving provenance relationships within query results is that these results can be further queried. This can allow users to decompose complicated queries into smaller subqueries, and also incrementally refine query results. Thus, a generic query language should be *closed*, i.e., similar to relational algebra, where the result of a provenance query should be queryable using the same language.

**(6) Optimization Transparency.** Because the amount of provenance information produced by a workflow run can be large (e.g., due to the number of workflow steps, and input and output data sets), systems that manage provenance information must provide efficient storage and query approaches [9,8,10,3]. Thus, a generic query approach should be amenable to query optimization techniques, and these optimizations should be independent of the query language itself (i.e., users should not have to modify queries for efficiency).

The first two desiderata are addressed directly through our unified provenance model, which does not depend on any particular workflow system and can be implemented using different underlying provenance storage representations. The rest of this section describes techniques for querying our model, with the goal of address the remaining issues. We first describe how QLP can be used to query the different dimensions of our unified provenance model, and discuss how these dimensions can be combined through *hybrid queries* that mix lineage information with information about the structure of data and the versions of these structures produced by a workflow run. We also give examples of common provenance queries that can be expressed in QLP and discuss our current implementation.

## 3.1 Provenance Queries Using QLP

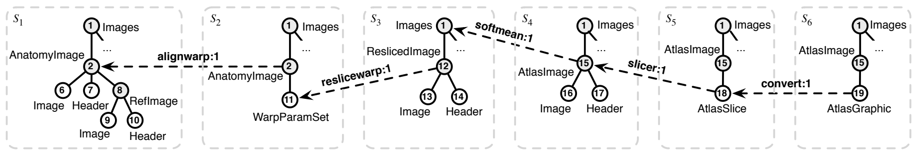
Fig. 7a shows a portion of the trace for a typical run of the workflow of Fig. 2, which we use below in describing the different types of provenance queries supported in QLP.

**Queries over Nested Data.** Because data structures in our provenance model represent XML trees, these structures can be queried using standard XML languages. In QLP, we adopt XPath as our query language for accessing nested data. For each trace, we also define a simple view that merges the different versions of structures within the trace into a *combined data structure*. As an example, Fig. 7b shows the combined data structure $s$ for the trace of Fig. 7 that consists of the structures $s_1$ to $s_6$. Queries over combined structures provide general access to all the nodes used and produced by a run, e.g., to return nodes of a specific type or with specific metadata annotations. The following XPath expressions are valid QLP queries, which are posed against the combined data structure.

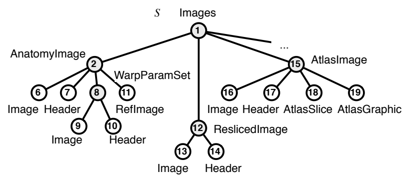
//Image (1)
//AtlasGraphic[modality="speech"]/@* (2)

These queries return (1) the nodes of type Image that were input to or produced by the workflow run, and (2) the metadata annotations (represented as XML attributes) for nodes of type AtlasGraphic with the value "speech" assigned to the (metadata) attribute "modality" [18]. Given a trace $T = (V, E, L, \alpha, \tau, \gamma)$, the combined structure $s$ represents the tree $\tau(s) = \tau(s_1) \cup \cdots \cup \tau(s_n)$ for each $s_i \in V$.[4] XPath queries expressed over combined structures $s$ return ordered subsets of nodes within the tree $\tau(s)$. We note that

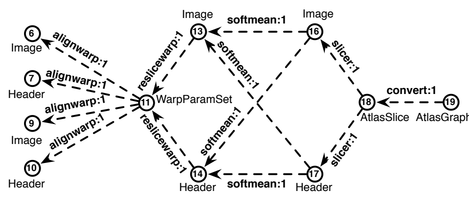
[4] Because invocations can only insert and delete nodes, merging two trees is straightforward.



**(a). Trace versions with fine-grained dependencies**



**(b). Combined data structure**



**(c). Fine-grained dependencies of data nodes**

**Fig. 7.** Example trace of a typical run of the workflow in Fig. 2: (a) versions and dependencies created by the first invocation of each actor corresponding to stages 1 through 5; (b) the corresponding combined structure; and (c) the corresponding fine-grained dependency graph

the combined structure of a trace is distinct from the final versions of structures, and instead, contain all data input to and produced by a run (including deleted nodes).

**Queries over Flow Relations (Versions).** In addition to queries expressed over the combined data structure of a trace, QLP also provides constructs for accessing specific versions of structures produced by a workflow run. As mentioned in Section 2, a node in $N$ may occur within multiple versions. We denote an occurrence of a node $n \in N$ within a structure $s$ as $n@s$. The expression $n@s$ *positions* node $n$ at structure $s$ (i.e., $n$ is *positioned* at $s$), whereas the expression $n$ leaves the node *unpositioned*. The following QLP queries use the **@in** and **@out** constructs (see Table 1) to access the different versions of structures within a trace (according to the flow relations).

```
@in                                         (3)
@out                                        (4)
@out slicer:1                               (5)
18 @out slicer:1                            (6)
```

These queries return (3) the input structure $s_1$ of the run, (4) the output structure $s_6$ of the run, (5) the version of the output structure $s_5$ produced by the first invocation of the Slicer actor, and (6) the version of node 18 in the structure $s_5$ produced by the first invocation of Slicer.

Unlike querying combined structures, the **@in** and **@out** operators return sets of positioned nodes. For example, the set of nodes returned by query (3) when applied to the example in Fig. 7a would include $1@s_1$, i.e., node 1 positioned at $s_1$ (in addition to all of the descendents of node 1 in $s_1$). The **@in** and **@out** operators return the nodes of structures input to and output by the run when no invocations are given, and when invocations are given, the input and output structures of the invocation, respectively. An (unpositioned) node may also be supplied as an argument to these operators, in which case the set of nodes returned contains only a positioned version of the node (i.e., the node without its descendents), or the empty set if no such node exists in the

**Table 1.** Example QLP constructs and short-hand notations

| *Construct Primitive* | *Shorthand* | *Result* |
|---|---|---|
| $N$ **@in** $I$ | | The version of $N$ input to invocation $I$ (if $I$ not given, at run input) |
| $N$ **@out** $I$ | | The version of $N$ output by invocation $I$ (if $I$ not given, at run output) |
| $N_1$ **derived** $N_2$ | $N_1..N_2$ | Lineage edges forming *transitive* paths from $N_1$ to $N_2$ |
| $N_1$ **1_derived** $N_2$ | $N_1.N_2$ | Lineage edges forming *one-step* paths from $N_1$ to $N_2$ |
| $N_1$ **through** $I$ **derived** $N_2$ | $N_1..\#I..N_2$ | Lineage edges forming *transitive* paths from $N_1$ to $N_2$ through invocation $I$ |
| $N_1$ **through** $I$ **1_derived** $N_2$ | $N_1.\#I.N_2$ | Lineage edges forming *one-step* paths from $N_1$ to $N_2$ through invocation $I$ |
| **type** $N$ | | The type (tag name) of a node $N$ |
| **actors** $L$ | | Actors of invocations in lineage edges $L$ |
| **invocations** $L \cup A$ | | Invocations of lineage edges $L$ or of an actor $A$ |
| **nodes** $L$ | | Nodes of lineage edges $L$ |
| **input** $L$ | | Source nodes of lineage edges $L$ |
| **output** $L$ | | Sink nodes of lineage edges $L$ |

corresponding structures. For example, query (6) applied to the run graph of Fig. 7a returns the positioned node $18@s_5$.

**Queries over Lineage (Fine-Grained Dependencies).** The majority of provenance queries are expressed over lineage relations (e.g., see [18,6]). Fig. 7c is an example of a portion of the lineage relations for the trace of Fig. 7a, showing only the fine-grained dependencies of data nodes (i.e., collection nodes are not shown). The QLP operators for querying lineage (see Table 1) act as filters over lineage relations $L \subseteq N \times I \times N$, returning subsets of $L$. Thus, in addition to query results maintaining lineage relations, these results can be further queried via QLP lineage operators. For example, consider the following simple QLP lineage queries.

* **derived** 19 (7)
6 **derived** * (8)
* **through** slicer:1 **derived** * (9)

These queries return (7) lineage relations denoting the set of paths starting at any node and ending at node 19, (8) lineage relations denoting the set of paths starting at node 6 and ending at any node, and (9) lineage relations denoting the set of paths with any start and end node, but that go through the first invocation of the Slicer actor.

Given a set of lineage relations $L$, let $paths(L)$ be the set of paths implied by $L$, where a lineage path takes the form

$$n_1.i_1.n_2 \ldots i_j \ldots n_{k-1}.i_{k-1}.n_k \qquad (k \geq 2)$$

For any set $P$ of such paths, let $edges(P)$ be the set of lineage relations for $P$, such that $L = edges(paths(L))$. Given lineage relations $L$, the query '$n_1$ **derived** $n_k$' returns the subset of $L$ consisting of paths of any length starting at $n_1$ and ending at $n_k$. Similarly, the query '$n_1$ **through** $i_j$ **derived** $n_k$' returns the subset of $L$ consisting of paths of any length starting at $n_1$ and ending at $n_k$ that also pass through invocation $i_j$. If **1_derived** (i.e., one-step or immediately derived) is used in place of **derived**, then only paths of the form $n_1.i_j.n_k$ are returned. That is, in **1_derived** each selected path is defined by a single lineage relation. These operators can be combined to form more complex lineage paths. In this case, each individual expression specifies a segment of a larger lineage path. For instance, a query such as '$n_1$ **derived** $n_j$ **derived** $n_k$' selects lineage relations that form paths from $n_1$ to $n_j$ and from $n_j$ to $n_k$.

Lineage queries may also be expressed using a shorthand notation (see Table 1). Similar to the use of '/' and '//' in XPath expressions for navigating parent-child relationships of nodes, we use '.' to navigate immediate (i.e., one-step) lineage paths and '..' to navigate transitive lineage paths (i.e., paths consisting of one or more steps). When using the shorthand notation, we distinguish actor invocations from structural nodes by prefixing invocations with '#'. For example, queries (7-9) can be expressed as '*..19', '6..*', and '*..#slicer:1..*' (or simply '#slicer:1') using the QLP shorthand notation.

**Hybrid Queries over Multiple Dimensions.** Lineage queries can be combined with queries over data structures. We call these "hybrid" queries since they allow both structural and lineage information to be accessed simultaneously. The following are simple examples of QLP hybrid queries.

* **derived** //AtlasImage//* (10)
//ReslicedImage//* **through** softmean:1 **derived** //AtlasImage (11)
//Image **through** Slicer[x="0.5"] **derived** //AtlasImage (12)
(//* **@in** slicer:1) **derived** //AtlasImage (13)

These queries return lineage relations denoting (10) paths ending at descendents of AtlasImage nodes, (11) paths starting at ReslicedImage descendent nodes and ending at AtlasImage nodes that pass through the first invocation of SoftMean, (12) paths starting at Image nodes and ending at AtlasImage nodes that pass through invocations of Slicer with parameter "x" set to the value 0.5 (i.e., resulting in "Atlas X Images"), and (13) paths starting at (positioned) input nodes of the first invocation of Slicer and ending at AtlasImage nodes. By returning all lineage relations, the result of these queries can be easily visualized as lineage graphs and can be treated as views that can be further queried using similar expressions.

Hybrid 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 lineage queries to the resulting nodes. For example, when applied to Fig. 7, query (10) is first evaluated by executing the XPath query '//AtlasImage//*' over the combined structure $s$, returning nodes 16–19. For each node, a separate lineage query is evaluated, i.e., '* **derived** 16', '* **derived** 17', '* **derived** 18', and '* **derived** 19', such that the answer to query (10) is the unique set of resulting lineage relations.

Hybrid queries may also combine lineage operators, e.g., consider the following query that returns the lineage relations denoting paths through SoftMean, Slicer, and Convert invocations, and ending at nodes of type AtlasGraphic.

* **through** Softmean **derived** * **through** Slicer **derived** * **through** Convert **derived** //AtlasGraphic (14)

For queries that specify complex lineage paths, it is often more convenient to use the shorthand notation, e.g., query (14) can be equivalently written as

#Softmean .. #Slicer .. #Convert .. //AtlasGraphic (short-hand version of 14)

QLP also supports additional operations that can be applied to sets of lineage relations, which are summarized in Table 1 and used in the following examples.

**invocations**(AlignWarp[m="12", dateOfExecution="Monday"]) (15)

**output**(//Header[max="4096"] **derived** //AtlasGraphic) (16)
**output**(#AlignWarp[m="12"] .. #Softmean) (17)
**input**(//Image **derived** //AtlasGraphic @**out**) (18)
**actors**(//Image **derived** //AtlasGraphic @**out**) (19)
(//Image @**in**) − **input**(//Image @**in derived** //AtlasGraphic @**out**) (20)

Query (15) returns the set of invocations of AlignWarp that used the values 12 and "Monday" for the "m" and "dateOfExecution" parameters, respectively. Query (16) returns the output nodes for lineage paths starting at Header nodes having the value "4096" for the global maximum metadata field "max" and ending at AtlasGraphic nodes. Specifically, the **output** operation (similarly, **input**) returns the nodes within lineage relations that do not have outgoing (incoming) lineage edges. Query (17) returns the output nodes produced by invocations of SoftMean that were derived from AlignWarp invocations using the value 12 for parameter "m". This query combines multiple lineage operators and is expressed using the QLP shorthand notation. Query (18) returns the input nodes of paths starting from Image nodes and ending at AtlasGraphic nodes that were part of the output of the run. Query (19) is similar to (18) but returns the actors of invocations used in the corresponding lineage paths. Query (20) finds the Image nodes input to the workflow run that were not used to derive any output Image nodes. This is achieved by first finding all Image nodes input to the workflow run, and then subtracting from this set the input Image nodes used to derive an output AtlasGraphic node. Although not shown here, the **type** operator of Table 1 can be used to return the tag names of XML nodes.

QLP also provides support for constraining the structure of lineage paths using regular expressions. For instance, the query '$n_1$ **through** ($i_1 \mid i_2$) **derived** $n_2$' selects lineage relations denoting paths from $n_1$ to $n_2$ that pass through either $i_1$ or $i_2$. These queries can be used by workflow developers, e.g., to ensure that complex workflows (e.g., involving multiple branches) executed correctly.

## 3.2 Implementation and Evaluation of QLP Query Support

Our current implementation of QLP supports queries expressed using the shorthand notation described in Table 1, and answers these queries using the relational storage strategies described in [3]. In particular, we have implemented a Java application that takes as input a QLP query, transforms the query into an equivalent SQL query, and executes the SQL query over the underlying database. Using our implementation, we were able to express and answer the queries of the first provenance challenge [18] as well as queries similar to those in [6]. A number of variants of these queries are also given in (1–20) above. The QLP versions of these queries are significantly more concise than those typically expressed against underlying storage structures [18], and in general, are easier to formulate and comprehend.

The storage strategies described in [3] apply reduduction techniques for efficiently storing both immediate and transitive provenance dependencies. In particular, a naive approach for storing nodes $N$ and their dependencies $D$ is as tuples $P(N, D)$ in which nodes involved in shared dependencies will be stored multiple times. For example, if nodes $n_4$, $n_5$, and $n_6$ each depend on nodes $n_1$, $n_2$, and $n_3$, nine tuples must be stored

$P(n_4,n_1)$, $P(n_4,n_2)$, $P(n_4,n_3)$, $P(n_5,n_1)$, ..., $P(n_6,n_3)$, where each node is stored multiple times in $P$. Instead, the approach in [3] introduces additional levels of indirection through "pointers" (similar to vertical partitioning) for storing reduced sets of dependencies. Thus, we divide $P(N,D)$ into two relations $P_1(N,X)$ and $P_2(X,D)$ where $X$ denotes a pointer to the set of dependencies $D$ of $N$. For instance, using this approach we store only six tuples $P_1(n_4,\&x)$, $P_1(n_5,\&x)$, $P_1(n_6,\&x)$, $P_2(\&x,n_1)$, $P_2(\&x,n_2)$, and $P_2(\&x,n_3)$ for the above example. Additional levels of indirection are also used to further reduce redundancies within dependency sets based on their common subsets, and similar techniques are used to reduce transitive dependency sets by applying reduction techniques directly to pointers (as described in [3]).

As an initial evaluation of the feasibility and scalability of our approach for executing QLP queries, we describe below the results of executing lineage queries over synthetic traces of increasing numbers of nodes and lineage relations. We compare our *reduced-transitive approach* (R), which transforms QLP queries to SQL queries over our underlying relational schema storing immediate and transitive lineage relations in reduced form, to a *naive approach* (N) in which only immediate dependencies are stored, and corresponding QLP lineage queries are transformed to recursive stored procedures.

Our experiments were performed using a 2.4GHz Intel Core 2 duo PC with 2 GB RAM and 120 GB of disk space. Each approach used MySQL to store provenance information. We compare query response time and storage size using synthetic traces ranging from 100 to 3000 nodes, $10^3$ to $10^4$ immediate dependencies, $10^4$ to $10^6$ transitive dependencies, and lineage paths of length 25 to 150, respectively. The synthetic traces were taken from [3], and represent typical lineage patterns generated by real-world workflows implementing phylogenetic and image-processing pipelines [3,5,18].

We consider the following basic lineage queries for evaluating query response time. In general, queries over lineage relations are considerably more expensive [10,3] than queries over only combined structures (i.e., XPath queries), or queries that select only versions of structures within a trace (e.g., using the **@in** and **@out** QLP operators).

$$* \,..\, n \tag{Q1}$$
$$n \,..\, * \tag{Q2}$$
$$\textbf{exists}\ n_1 \,..\, n_2 \tag{Q3}$$
$$n_1 \,..\, n_2 \tag{Q4}$$
$$n_1 \,..\, n_2 \,..\, n_3 \tag{Q5}$$

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 that start from a node $n$ (i.e., the "progeny" of $n$ [6]); (Q3) true if there is a lineage path from node $n_1$ to node $n_2$ (where '**exists**' denotes a boolean query); (Q4) lineage relations denoting paths starting at node $n_1$ and ending at node $n_2$; and (Q5) lineage relations denoting paths starting at node $n_1$ and ending at node $n_3$ that pass through node $n_2$.

The left side of Fig. 8 shows that the query response time for the reduced-transitive implementation grows linearly with increasing trace size, and is significantly faster than the naive approach (shown in dotted lines). The more expensive query response time of the naive approach is due to the use of recursive stored procedures to select transitive dependency relationships. Moreover, the increased query response time for the reduced-transitive approach, when compared to the naive approach, is not simply based

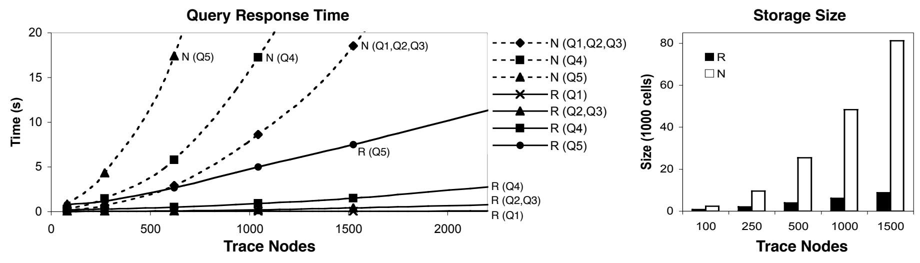


**Fig. 8.** Query response time (left) and storage size (right) for the reduced-transitive and naive approaches. In the reduced-transitive approach, because of the nature of the reduction strategies and the QLP-to-SQL translation, query Q1 takes (slightly) less time than query Q2, and similarly, query Q2 takes the same amount of time as query Q3 for the example traces.

on increases in storage size. In particular, the right side of Fig. 8 shows that when compared to the naive approach, storage size is also significantly smaller using our reduction strategies. Thus, we can improve query response time by materializing transitive lineage relations, while at the same time reducing the provenance storage size using reduction techniques. Fig. 8 also demonstrates that the standard approach of using recursive stored procedures to answer lineage queries does not scale linearly with trace size.

These results highlight the advantages of decoupling provenance query languages from underlying storage schemes. In particular, the same QLP queries can be answered using different storage strategies, in this case using the naive and reduced-transitive approaches, to transparently improve query response time and provenance storage size.

## 4 Related Work

Conventional models of provenance [17,8,2,9] are largely based on the assumption that data structures and processes are atomic. These assumptions do not hold, however, for many computation models employed within current scientific workflow systems [16,15]. Our approach extends the conventional model by supporting workflows composed of actors that can have multiple invocations (e.g., as in process networks [15]) and that employ update semantics over complex data structures (in particular, XML). This paper extends our prior work [3] on efficiently storing provenance information by defining a logical representation of the model and corresponding query approaches (i.e., QLP).

Few query languages have been specifically developed for querying provenance information. Instead, most approaches (e.g., see [18,22]) define their provenance models in terms of physical schemas represented within general-purpose data management frameworks (e.g., for storing relational, XML, or RDF data), and provenance queries are expressed using the corersponding query languages of the framework (e.g., SQL, XQuery, or SPARQL). This approach often leads to complex query expressions, even for answering basic provenance questions. For instance, to query over lineage information stored within a relational database, users must typically specify join operations over multiple tables and compute transitive closures [9,8,10,3]. Our approach differs by separating the logical provenance model from its physical representation and providing provenance-specific query constructs. Thus, users express queries against the logical model using more natural constructs, which are automatically translated to equivalent queries expressed against the underlying physical representation.

Similar to our approach, the VisTrails provenance query language (vtPQL) [21] defines provenance-specific query constructs. However, vtPQL only supports the standard provenance model, provides a limited set of constructs for acessing provenance relations, and does not support queries over data structures. For example, vtPQL defines $upstream(x)$ to return all modules (i.e., actors) that procede $x$ in the workflow definition. The upstream operator is the primary construct for answering lineage queries related to invocations, where a query "upstream(x) − upstream(y)" is used to find dependencies between two invocations of modules $x$ and $y$ within a run. However, vtPQL assumes only a *single* lineage path between two such modules, and thus would return incorrect results in the case of multiple lineage paths. Further, queries related to exclusive data lineage (e.g., $n_1$ .. $n_2$, $n_1$ .. $n_2$ .. $n_3$, $n_1$ .. $\#a$ .. $n_2$) are not supported in vtPQL.

In [12], Lorel [1] is used to represent and query provenance information. Similar to our approach, [12] employs generalized path expressions for querying lineage information. While both vtPQL and our approach provide a closed language over lineage edges, e.g., to support the visual representation of provenance lineage graphs, languages based on path expressions (Lorel, XPath, OQL, etc.) primarily return sets of identifiers (e.g., representing nodes and invocations) that require additional queries for constructing lineage graphs. Further, approaches such as [12] are still closely tied to physical representations of provenance.

## 5 Conclusion

We have described a logical model of provenance for representing scientific workflow runs based on computation models that work over XML structures. Our approach naturally extends the conventional provenance model by adding support for nested data and for accurately capturing detailed lineage information of processes employing update semantics. We also described a general approach for querying provenance using our Query Language for Provenance (QLP), which provides specialized constructs for expressing both structural (i.e., parent-child) and lineage (i.e., data dependency) "hybrid" queries. We have also shown how these constructs can be used to express a wide range of provenance queries (including those of [18]) and that answering QLP queries using relational technology can be feasible when both immediate and transitive dependency edges are stored according to [3]. As future work we plan to extend our current implementation of QLP with additional optimization techniques, with the goal of providing a generic and efficient approach for addressing challenges in managing the provenance of scientific workflow runs.

**Acknowledgements.** This work supported in part through NSF grants IIS-0630033, OCI-0722079, IIS-0612326, DBI-0533368, ATM-0619139, and DOE grant DE-FC02-01ER25486.

## References

1. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel Query Language for Semistructured Data. Intl. J. on Digital Libraries (1997)
2. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance Collection Support in the Kepler Scientific Workflow System. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 118–132. Springer, Heidelberg (2006)
3. Anand, M.K., Bowers, S., McPhillips, T., Ludäscher, B.: Efficient Provenance Storage Over Nested Data Collections. In: EDBT (2009)
4. Buneman, P., Suciu, D.: IEEE Data Engineering Bulletin. Special Issue on Data Provenance 30(4) (2007)
5. Bowers, S., McPhillips, T., Riddle, S., Anand, M., Ludäscher, B.: Kepler/pPOD: Scientific Workflow and Provenance Support for Assembling the Tree of Life. In: IPAW 2008. LNCS, vol. 5272 (2008)
6. Bowers, S., McPhillips, T., Ludäscher, B., Cohen, S., Davidson, S.B.: A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 133–147. Springer, Heidelberg (2006)
7. Callahan, S., Freire, J., Santos, E., Scheidegger, D., Silva, C., Vo, H.: VisTrails: Visualization Meets Data Management. In: SIGMOD (2006)
8. Chapman, S., Jagadish, H.V., Ramanan, P.: Efficient Provenance Storage. In: SIGMOD (2008)
9. Davidson, S.B., Freire, J.: Provenance and Scientific Workflows: Challenges and Opportunities. In: SIGMOD (2008)
10. Heinis, T., Alonso, G.: Efficient Lineage Tracking for Scientific Workflows. In: SIGMOD (2008)
11. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., den Bussche, J.V.: Petri Net + Nested Relational Calculus = Dataflow. In: Meersman, R., Tari, Z. (eds.) OTM 2005. LNCS, vol. 3760, pp. 220–237. Springer, Heidelberg (2005)
12. Holland, D., Braun, U., Maclean, D., Muniswamy-Reddy, K.K., Seltzer, M.: A Data Model and Query Language Suitable for Provenance. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272. Springer, Heidelberg (2008)
13. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: IFIP Congress, vol. 74 (1974)
14. Lee, E.A., Matsikoudis, E.: The Semantics of Dataflow with Firing. In: From Semantics to Computer Science: Essays in memory of Gilles Kahn. Cambridge University Press, Cambridge (2008)
15. Ludäscher, B., et al.: Scientific Workflow Management and the Kepler System. Conc. Comput.: Pract. Exper. 18(10) (2006)
16. McPhillips, T., Bowers, S., Zinn, D., Ludäscher, B.: Scientific Workflow Design for Mere Mortals. Future Generation Computer Systems 25(5) (2009)
17. Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J., Paulson, P.: The Open Provenance Model. Tech. Rep. 14979, ECS, Univ. of Southampton (2007)
18. Moreau, L., et al.: The First Provenance Challenge. Conc. Comput.: Pract. Exper., Special Issue on the First Provenance Challenge 20(5) (2008)
19. Oinn, T., et al.: Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. Conc. Comput.: Pract. Exper. 18(10) (2006)
20. Qin, J., Fahringer, T.: Advanced Data Flow Support for Scientific Grid Workflow Applications. In: ACM/IEEE Conf. on Supercomputing (2007)
21. Scheidegger, C., Koop, D., Santos, E., Vo, H., Callahan, S., Freire, J., Silva, C.: Tackling the Provenance Challenge One Layer at a Time. Conc. Comput.: Pract. Exper. 20(5) (2008)
22. Simmhan, Y.L., Plale, B., Gannon, D.: A survey of data provenance in e-science. SIGMOD Record 34(3) (2005)