Efficient Provenance Storage over Nested Data Collections

Manish Kumar Anand[1], Shawn Bowers[2], Timothy McPhillips[2], Bertram Ludäscher[1,2]
[1]Dept. of Computer Science, University of California, Davis
[2]Genome Center, University of California, Davis
{maanand, sbowers, tmcphillips, ludaesch}@ucdavis.edu

ABSTRACT

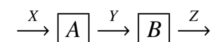
Scientific workflow systems are increasingly used to automate complex data analyses, largely due to their benefits over traditional approaches for workflow design, optimization, and provenance recording. Many workflow systems employ a simple dependency model to represent the provenance of data produced by workflow runs. Although commonly adopted, this model does not capture *explicit* data dependencies introduced by "provenance-aware" processes, and it can lead to inefficient storage when workflow data is complex or structured. We present a provenance model, extending the conventional approach, that supports (i) explicit data dependencies and (ii) nested data collections. Our model adopts techniques from reference-based XML versioning, adding annotations for process and data dependencies. We present strategies and reduction techniques to store immediate and transitive provenance information within our model, and examine trade-offs among *update time*, *storage size*, and *query response time*. We evaluate our approach on real-world and synthetic workflow execution traces, demonstrating significant reductions in storage size, while also reducing the time required to store and query provenance information.

1. INTRODUCTION

The automation of scientific data analyses often requires distinct software programs and services to be combined in complex ways. Traditionally, scientists use batch files and scripting languages (e.g., Perl) to automate the execution of individual programs and the routing of data between them. However, these approaches require significant technical skills and can be cumbersome and time-consuming for scientists, especially in data- and compute-intensive applications. *Scientific workflow systems* try to address these challenges, and promise to better support scientists wishing to automate their computational tasks [19].

Scientific workflow systems are often based on dataflow languages [24] in which a workflow is represented as a directed graph of nodes denoting computational steps implemented by *actors*, and connections representing the desired dataflow between steps. These systems additionally offer support for workflow design and analysis [20, 29], efficient execution and deployment of workflows [17, 21], and automatic recording of data and process dependencies (i.e., *provenance*) introduced during workflow runs [15, 33, 38]. Provenance support in particular has been recognized as an important added value of scientific workflow systems over traditional approaches. However, a number of data-management challenges must be overcome to effectively represent, store, and query the often large amounts of provenance information generated by workflow executions [15, 16, 33, 38].

Many workflow systems (e.g., [37, 4, 2, 40, 41, 35]) and provenance approaches (e.g., [13, 3, 10, 22, 34]) employ a simple provenance model that is data "agnostic" [15]. This model generally can be characterized as recording the inputs and outputs for each actor invocation occuring within a workflow run. Conceptually, a *workflow execution trace* in such a model consists of pairs of records $\mathsf{in}(x, a)$ and $\mathsf{out}(a, y)$, stating that x was an input and y an output of an actor invocation a.[1] This information is then used to infer data and process dependencies. For example, a run of a simple workflow



can be captured by a set of trace facts $\mathsf{in}(x, a)$, $\mathsf{out}(a, y)$, $\mathsf{in}(y, b)$, and $\mathsf{out}(b, z)$, implying that invocation a of actor A directly preceded invocation b (of B), and that output y directly depended on x, while z indirectly depended on x. In most systems, each input and output of an invocation is denoted by one or more *tokens* that encapsulate data values (e.g., DNA sequences) or references to values (e.g., filenames or accession numbers). Tokens are assumed to be immutable (once created their values cannot be modified), are assigned unique identifiers, and may be further organized into records, lists, trees, streams, and so on [34, 29, 32].

This "conventional" model of provenance is useful for representing data and process dependencies of scientific workflows consisting primarily of *black-box transformations* [14], in which actors (1) produce new outputs from their inputs; and (2) use all inputs to derive their outputs, implying that all invocation outputs depended on all invocation inputs.

However, many scientific workflows do not operate under these assumptions, and in such cases directly employing the conventional model can result in inefficient provenance storage and insufficient (or even incorrect) inferences of data dependencies. For instance, many systems (e.g., [27, 32, 36, 31, 23, 39, 29, 30, 18, 26]) support actors that make only small changes or updates to incoming data, passing on some or all of their input to downstream actors. Thus, if invocation a above retains within its output y some unchanged

[1]Additional information may also be captured, e.g., timestamps of invocation reads and writes, and invocation parameters.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.
EDBT'09, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...$5.00

substructure s from its input x, denoted as[2]

$$x = (s \oplus x_0), \quad y = (s \oplus y_0)$$

then s will be stored twice: once in the trace record $\mathsf{in}(x, a)$ (call this occurrence s_x) and once in $\mathsf{out}(a, y)$ (call this occurrence s_y). In the conventional model, all parts of output y are said to depend on all parts of input x, so these records may also *incorrectly* imply that s_y depends on x_0.

Furthermore, because actors often wrap complex external applications and services, various patterns of data dependencies (e.g., see [31, 32, 36, 7]) can arise in which not all parts of the output depend on all parts of the input. Assume, e.g., that invocation a above receives input x and produces output y as follows

$$x = (x_1 \oplus \ldots \oplus x_n), \quad y = (y_1 \oplus \ldots \oplus y_m).$$

Common examples of a with data dependency patterns not supported by the conventional model include:

(a) actors that filter data prior to applying a scientific function, resulting in dependencies where each y_i depends only on *some* of the x_j's

(b) actors that process each input token in turn, resulting in dependencies where each y_i depends on a *single* x_j ($j = i$)

(c) actors that perform running aggregates over their input, resulting in dependencies where each y_i depends on the set $\{x_1, \ldots, x_i\}$

(d) actors that apply functions over their input using sliding windows of a fixed size w, resulting in dependencies where each y_i depends on the window $\{x_{i-w}, \ldots, x_i\}$

Because many possible dependency patterns may occur (e.g., via combinations or variations of the above), in general all data dependencies must be recorded explicitly for each workflow run to accurately represent data lineage.

Contributions. We present a new provenance model that generalizes the conventional approach described above to support (i) actors that employ update and "add-only" semantics ($y = x \pm \Delta$, for input x and output y) and (ii) explicit declarations of data dependencies.[3] The model is an integral part of the *Project Histories* management framework [8] being developed within the Kepler scientific workflow system [27]. A goal of this framework is to allow scientists to easily store, interconnect, organize, and query comprehensive provenance information for multiple workflow runs, together with input and derived data products used within their research projects. In the conventional model, the amount of provenance information that must be stored may be many times larger than the data products generated, even for single runs [10, 22, 15]. The feasability of managing multiple, interconnected runs therefore relies on an efficient representation of provenance information that also minimizes the time needed to store and query traces.

Our model uses nested collections (i.e., XML trees) for representing scientific data—similar tree-structured models also have been employed in Kepler [29] and other approaches [32, 23, 10]. We support actor update semantics and explicit data dependencies on these nested collection structures by employing *embedded* (or "inline") provenance *annotations*, which describe the *changes* (or *deltas* Δ) made to token streams by invocations at each workflow step. These annotations are used to record the tokens that were inserted (added to the token stream) and deleted (removed from the token stream) by invocations, and the token dependencies that were introduced. Thus, instead of separately storing input and output structures of each actor invocation, our traces consist of a single, "condensed" version of the overall structure, and the complete version history including all intermediate "snapshots" can be reconstructed as needed from the provenance annotations of the trace.

In the introductory example, invocation b received input y and produced output z. Let Δ_b denote the changes that b made to y to produce z, i.e., $z = \Delta_b(y)$. Our approach is to store only the most recent version of data (e.g., z), but add provenance annotations (Δ_b) that allow us to "undo" any updates and obtain all earlier versions of data products in the processing history. In this way, we can represent y as the pair (Δ_b^-, z), where the *backward delta* Δ_b^- denotes the inverse of Δ_b, thus

$$y = \Delta_b^-(z).$$

Similarly, the input x of invocation a can be represented by the pair (Δ_a^-, y) such that

$$x = \Delta_a^-(y) \; = \; \Delta_a^-(\Delta_b^-(z)).$$

An advantage of this approach is that provenance information can be represented without introducing unnecessary redundancy. For example, if y and z share a subtree s, then s is stored only once in the resulting trace. Our model can also reduce the number of stored data dependencies by exploiting the nested structure of data collections to "cascade" dependencies within a trace.

Our other main contributions are the development and comparison of different strategies for efficiently storing, updating, and querying provenance traces expressed in our model using a relational database system. We define: (i) inference techniques for *expanding* and *collapsing* the implied data and process dependencies within a trace; (ii) *reduction techniques* for minimizing the size of provenance storage for direct and transitive data and process dependencies (these techniques are generic and applicable to both hierarchical and non-hierarchical data structures); and (iii) *relational views* for recovering all implied and reduced dependencies generated by the inference and reduction techniques. We compare trade-offs in terms of provenance storage size, update time, and query response time, and show using real-world and synthetic provenance traces that our techniques can improve each of these.

Outline. The rest of this paper is organized as follows. Section 2 develops a formal model of our provenance approach. Section 3 describes our overall framework and optimization strategies for efficiently storing workflow traces. Section 4 presents our experimental results, demonstrating the advantages of our reduction techniques presented in Section 3 and examines the trade-offs among the different strategies for storage, udpate, and query. Section 5 describes related work, comparing our approach to other provenance models and provenance optimization approaches. Section 6 concludes by summarizing our results.

2. PROVENANCE MODEL

We use unranked, labeled, ordered trees (similar to XML) to represent workflow data products. Tree nodes are either *collection tokens* or *data tokens*. A collection token may be an internal node (representing a non-empty collection) or a leaf node (empty collection); data tokens occur as leaf nodes only. All tokens are assumed to have unique identifiers (e.g., positive integers), and labels are used to tag (or type) nodes. For example, a data token containing

[2]"$\oplus$" splits complex data into subparts (e.g., into list elements, subtrees, etc.); the details are not important here.

[3]Explicit dependencies may be obtained, e.g., from actors that delcare their dependencies or via inferences made from workflow system "observables" such as timestamps or state information [7].

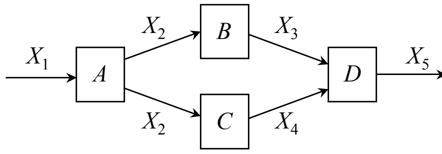


Figure 1: Workflow graph W with actors A, B, C, and D, and dataflow edges X_1 to X_4.

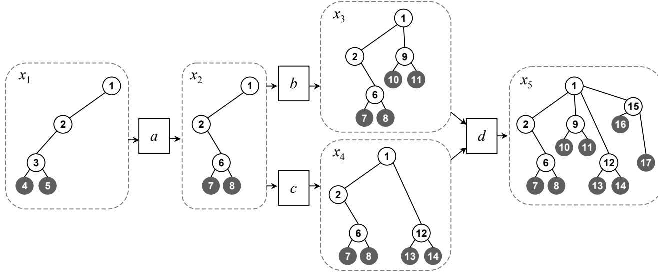


Figure 2: Execution (or *run*) of workflow W, showing versions $x_1, \ldots, x_5$ of intermediate data products received/produced by invocations a, b, c, and d.

a single DNA sequence may have the label `dna_sequence`; a collection token representing a DNA sequence alignment (i.e., a list of aligned sequences) may have the label `dna_alignment`.

Figure 1 shows an example workflow graph; a workflow run is given in Figure 2. In this example, invocation a receives the overall workflow input but only uses the subtree at node 3 as input to generate a new output subtree at node 6 (integers '3' and '6' are node identifiers). We say that node 6 *depended on* node 3;[4] or alternatively, 3 *contributed to* 6. The remaining parts of the input are passed on with node 6 to downstream actor invocations b and c. The changes Δ_a made by a can be represented by the following provenance annotations:

$$\Delta_a \quad = \quad \{\mathsf{ins}(6, a), \mathsf{del}(3, a), \mathsf{dep}(6, 3)\},$$

stating that (i) node 6 was *inserted by* invocation a, (ii) node 3 was *deleted by* invocation a, and (iii) 6 *depended on* 3. Annotations like these are recorded during workflow execution and constitute the provenance of the invocation. The remaining invocations of Figure 2 are represented by the following sets of annotations:

$$\begin{aligned} \Delta_b &= \{\mathsf{ins}(9, b), \mathsf{dep}(9, 2)\} \\ \Delta_c &= \{\mathsf{ins}(12, c), \mathsf{dep}(12, 6)\} \\ \Delta_d &= \{\mathsf{ins}(15, d), \mathsf{dep}(16, 9), \mathsf{dep}(17, 12)\} \end{aligned}$$

In the conventional model described above, the input and output of each invocation (i.e., $x_1, \ldots, x_5$ in Figure 2) would be recorded as part of the workflow trace. This approach would require storing the same sets of nodes multiple times, e.g., nodes 1 and 2 are part of every intermediate data product of the run. Instead, in our approach, only the unique set of nodes across all inputs and outputs are recorded together with the above provenance annotations.

Figure 3 shows this "condensed" trace for the workflow run of Figure 2. We use the following conventions to represent annotations in the figure: "+a" denotes an insertion annotation, here: $\mathsf{ins}(6, a)$; "-a" denotes a deletion annotation, here: $\mathsf{del}(3, a)$; dashed arrows between nodes represent data dependency annotations, e.g.,

[4]We say 'node n' or simply 'n' for 'the subtree rooted at n'.

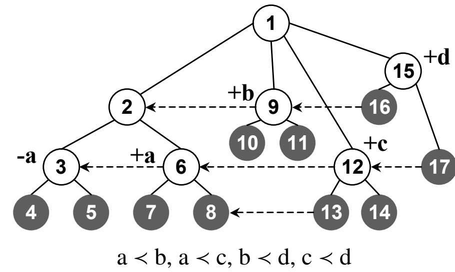


Figure 3: "Condensed" workflow trace of Figure 2, with collection tokens (white), data tokens (gray), insertion ($+x$) and deletion ($-x$) annotations, data-dependency annotations (dashed lines), and partial invocation order ($\prec$-relation at the bottom).

the arrow from node 6 to 3 denotes an annotation $\mathsf{dep}(6, 3)$; and $a \prec b$ denotes that invocation a preceded invocation b.

2.1 The Basic Model

Data collections are represented as (XML) trees; these token streams constitute the dataflow shown in Figure 1. From the perspective of provenance, we assume actor invocations operate by: (1) *inserting* new subtrees into the tree (i.e., data stream); and (2) *deleting* subtrees from the tree, thereby removing nodes from the data stream and not passing them on to downstream invocations.[5]

A workflow execution *trace* $T = (S, I, A, \prec_I)$ consists of a tree structure S, invocations I, node annotations A, and a strict, partial invocation order $\prec_I$ (i.e., representing a DAG). The tree $S = (N, E)$ is unranked, labeled, and ordered; N is a set of nodes, and $E \subseteq N \times N$ is a set of (ordered) parent-child edges. A node annotation A is a (partial) mapping

$$A : N \rightarrow I^{+} \times I^{-} \times 2^{N}$$

associating with a node $n \in N$ the invocation that inserted n (else $\perp$), the invocation that deleted n (else $\perp$), and a subset of nodes from N on which an insertion depended. We consider both node annotations A and invocation order $\prec_I$ as constituting provenance annotations. For a node n such that

$$A(n) = (+a, -b, \{x_1, \ldots, x_n\}),$$

the invocation that inserted n is a, the invocation that deleted n is b, and the insertion of n depended on all x_i ($1 \leq i \leq n$). We use $\mathsf{child}(x, y)$ to denote that the child of node x is y. The following relations express provenance annotations:

(1) $\mathsf{ins}(n, a)$ states that a node n was *inserted* by invocation a. Nodes are either inserted by an invocation or else were part of the overall workflow input.

(2) $\mathsf{del}(n, a)$ states that a node n was *deleted* (i.e., not passed on) by an invocation a. The node is still stored within the trace, but is "marked" deleted. Marked deletions allow us to reconstruct intermediate versions of the trace.

(3) $\mathsf{dep}(n, d)$ states that the *insertion* of a node n *directly depended on* a node d (i.e., d "contributed to" n's insertion).

(4) $a \prec b$ states that invocation *a preceded* invocation b, i.e., b received input which includes (part of) a's output.[6]

[5]Workflow systems typically allow only these operations to better support concurrent execution of actors by avoiding potential race conditions caused by allowing *ad-hoc* structural modifications [29].
[6]$a \prec b$ does *not* imply that a finished executing before b started. Streaming execution models can exploit this flexibility [29].

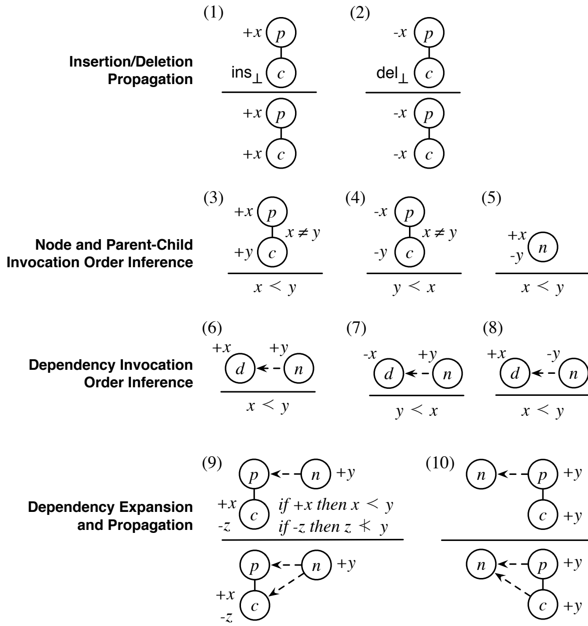


Figure 4: Rules for computing fully annotated traces shown graphically with antecedents drawn above each line and consequents below.

2.2 Minimizing Annotations

An important feature of our model is that not all nodes in a trace require explicit provenance annotations. Instead, these annotations can be inferred from the annotations of other nodes in a trace. This approach allows traces to be stored in a more compact form as compared to an equivalent trace that is fully annotated. The hierarchical structure of a trace is used to minimize the number of provenance annotations recorded. For example, insertion annotations can be recorded at the highest node n in the trace where they apply, and implicitly cascade to descendent nodes (i.e., all subtree nodes of n) that existed at the time of the insertion (similarly for deletion annotations). Nodes that depend on a collection (node) c also depend on all descendents (subtree nodes) of c that were present at the time of insertion of nodes. In Figure 3, e.g., the insertion and dependency annotations for nodes 7 and 8 are the same as for node 6; nodes 6, 7, and 8 have (implicit) dependencies on nodes 4 and 5.

A partially annotated trace is expanded by applying a set of inference rules, shown graphically in Figure 4 and in first-order logic in Figure 5. Rules 1–10 in Figure 5 correspond to those in Figure 4; whereas rules 11–13 place additional constraints on traces. We employ unary relations $\mathsf{ins}_\perp(N)$ and $\mathsf{del}_\perp(N)$ in Figure 5 to mark nodes of a trace without initial insertion or deletion annotations.

In Figure 4, the first set of inference rules (1–2) propagate insertion and deletion annotations from collection nodes to their children nodes. For example, if a collection (or parent) node p has an insertion annotation $\mathsf{ins}(p, x)$, and a child c of p does not have an associated insertion annotation, then applying the inference rule results in the annotation $\mathsf{ins}(c, x)$. The second set of rules (3–5) infer invocation order according to the insertion and deletion annotations of parent-child relationships and individual nodes. The third set of rules (6–8) infer invocation order, but using insertion and deletion annotations on token dependencies. The final set of rules (9–10) expand the set of dependencies of nodes and propagate dependencies to child nodes.

$$\begin{array}{ll}
(1) & \mathsf{child}(p,c) \wedge \mathsf{ins}(p,x) \wedge \mathsf{ins}_\perp(c) \rightarrow \mathsf{ins}(c,y) \\
(2) & \mathsf{child}(p,c) \wedge \mathsf{del}(p,x) \wedge \mathsf{del}_\perp(c) \rightarrow \mathsf{del}(c,y) \\
(3) & \mathsf{child}(p,c) \wedge \mathsf{ins}(p,x) \wedge \mathsf{ins}(c,y) \wedge x \neq y \rightarrow (x \prec y) \\
(4) & \mathsf{child}(p,c) \wedge \mathsf{del}(p,x) \wedge \mathsf{del}(c,y) \wedge x \neq y \rightarrow (y \prec x) \\
(5) & \mathsf{ins}(n,x) \wedge \mathsf{del}(n,y) \rightarrow (x \prec y) \\
(6) & \mathsf{dep}(n,d) \wedge \mathsf{ins}(d,x) \wedge \mathsf{ins}(n,y) \rightarrow (x \prec y) \\
(7) & \mathsf{dep}(n,d) \wedge \mathsf{del}(d,x) \wedge \mathsf{ins}(n,y) \rightarrow (y \prec x) \\
(8) & \mathsf{dep}(n,d) \wedge \mathsf{ins}(d,x) \wedge \mathsf{del}(n,y) \rightarrow (x \prec y) \\
(9) & \mathsf{dep}(n,p) \wedge \mathsf{child}(p,c) \wedge \mathsf{ins}(n,y) \wedge \\
 & \quad ((\mathsf{ins}(c,x) \wedge (x \prec y)) \vee \mathsf{ins}_\perp(c)) \wedge \\
 & \quad ((\mathsf{del}(c,z) \wedge (z \nprec y)) \vee \mathsf{del}_\perp(c)) \rightarrow \mathsf{dep}(n,c) \\
(10) & \mathsf{child}(p,c) \wedge \mathsf{dep}(p,n) \wedge \mathsf{ins}(p,y) \wedge \mathsf{ins}(c,y) \rightarrow \mathsf{dep}(c,n) \\
(11) & \mathsf{ins}(n,a) \wedge \mathsf{ins}(n,b) \rightarrow (a = b) \\
(12) & \mathsf{dep}(n,d) \rightarrow \exists a \; \mathsf{ins}(n,a) \\
(13) & \mathsf{del}(n,a) \wedge \mathsf{del}(n,b) \rightarrow (a \nprec b) \wedge (b \nprec a)
\end{array}$$

Figure 5: First-order logic rules for computing fully annotated traces (1–10), and additional constraints (11–13) for checking well-formedness. Free variables are implicitly ∀-quantified.

The additional rules of Figure 5 define constraints on traces requiring that: (11) each node is inserted by at most one invocation; (12) any node that is dependent on another node is inserted by some invocation; and (13) an invocation receiving a deleted node cannot delete the node again.

Complete Traces. We say that a trace is *complete* (or *fully annotated*) if no new annotations can be inferred using the above inference rules. Let Σ be the set of annotation inference rules (1–10) of Figure 5, and T be a workflow trace. We can complete T by computing the deductive closure DC of T w.r.t. Σ, i.e., complete$(T) = DC(\Sigma, T)$. Note that Σ can be encoded as a safe, negation-free Datalog program, and thus the completion of T is guaranteed to be unique (i.e., resulting in a unique minimal model) [1].

Equivalent, Well-Formed, and Minimal Traces. We can use trace completion to define the *equivalance* of traces, as well as to check if a trace is well-formed: Two traces are *equivalent* if their completions are equivalent. A trace is *well-formed* if its completion consists of a well-formed tree structure S, satisfies all axioms of Figure 5, and $\prec_I$ induces a strict partial order over invocations I. Trace equivalence can be used to define the notion of a *minimal* trace: A trace is *minimal* if there is no other equivalent trace with fewer provenance annotations. While the completion of T is unique, there can be multiple minimal traces for T. For example, Figure 6 shows three equivalent traces in which (a) and (b) are both minimal, and (c) is completed.

Cycle-Free Traces. Token dependencies in a well-formed trace are guaranteed to be acyclic, which follows the intuitive notion of causality that has been identified as an important property of provenance representations [34]. To see that dependencies are always acyclic in our model, first note that because we require a strict partial order over invocations, invocation order is by definition cycle-free.[7] For a cyclic data dependency, there must exist nodes n and d

[7] This does *not* preclude loops in the *workflow graph*: Multiple invocations $a_1, \ldots, a_n$ can arise either by an actor A "firing" multiple times on a data stream or by repeatedly firing within a workflow loop. In either case, the partial order on invocations is strict.

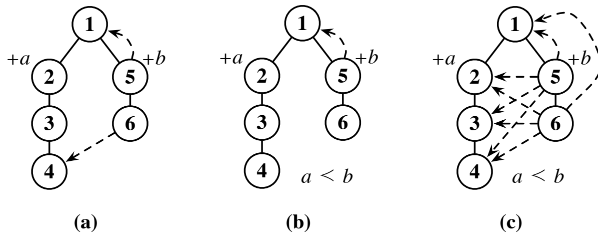


Figure 6: The same trace shown minimally annotated in (a) and (b) and fully annotated in (c).

such that n is directly or indirectly dependent on d, and d is directly dependent on n. In such a case, if $\mathsf{ins}(n, a)$ and $\mathsf{ins}(d, b)$, then by axiom (6) we have $a \prec b$ and $b \prec a$, thereby violating the strict partial ordering restriction on invocations.

Expanding and Collapsing Traces. We use structural recursion to *expand* and *collapse* traces, i.e., to construct equivalent traces with larger or smaller sets of provenance annotations. Specifically, we can complete a trace by traversing its collection and data nodes, and at each step in the traversal apply the annotation rules of Figure 4. In general, it is not possible to fully expand a given trace in a single pass, i.e., a single preorder, inorder, postorder, or breadth-first traversal, in which each node of the trace is visited only once. As a simple counter-example, consider the trace in Figure 6(a). Its completed version cannot be computed using one of the above traversal algorithms, since node 6 must be visited before 2, node 2 must be visited before 3, and node 3 must be visited before 4.

Our approach is to expand traces using three distinct preorder (i.e., top-down, left-to-right) traversals of S. The first pass propagates insertion and deletion annotations according to inference rules (1–2) of Figure 4, followed by applications of rules (3–5) to infer invocation order from nodes and parent-child relationships. The second pass generates the remaining invocation precedence relationships based on the rules (6–8). The third pass expands dependency sets and propagates dependencies to child nodes using the rules (9–10). Insertion and deletion annotations must be propagated before the second pass since both sides of a dependency require an insertion and/or deletion to infer the remaining invocation precendence relationships. Similarly, invocation order must also be known before dependencies can be expanded and propagated.

Expanded traces can also be minimized using a similar approach. Specifically, collapsing a fully annotated trace can be performed using a postorder (i.e, bottom-up, left-to-right) traversal of the trace where the following annotations are removed for each node n: (i) $\mathsf{dep}(n, c)$ if $\mathsf{dep}(n, p)$ and $\mathsf{child}(p, c)$; (ii) $\mathsf{dep}(n, d)$ if $\mathsf{child}(p, n)$ and $\mathsf{dep}(p, d)$; (iii) $\mathsf{ins}(n, x)$ if $\mathsf{child}(p, n)$ and $\mathsf{ins}(p, x)$; and (iv) $\mathsf{del}(n, y)$ if $\mathsf{child}(p, n)$ and $\mathsf{del}(p, y)$. Finally, we remove the invocation order annotations that are implied according to the rules in Figure 4(3–8).

Determining Intermediate Versions. Another important feature of our provenance model is that although each token is stored only once in a trace, the input and output of each invocation can be reconstructed by computing the corresponding "version" of the final workflow output. As described previously, the intermediate versions of the tree structures of Figure 3 are shown in Figure 2. Given a fully annotated trace, we compute the corresponding version of the input structure used by an invocation b as follows. Let $P = \{a \in I \mid a \prec b\}$ be the set of invocations that *preceded b*. The version corresponding to the input of b includes all nodes not deleted by an invocation in P such that either the node (1) was inserted by an invocation in P or (2) has no insertion annotation and thus was an input to the workflow run. The output of b is computed similarly, i.e., by removing from the input of b the nodes deleted by b and adding the nodes inserted by b.

Provenance Recording and Extensions. The model described here is implemented within Kepler for representing the provenance of scientific workflows developed using the COMAD workflow design paradigm [29, 6]. COMAD is one of many dataflow-based computation models supported in Kepler, and is based on the process network model [27] in which each actor runs concurrently over one or more input token streams. The COMAD framework provides built-in support for both nested data collections (analogous to tokenized XML data streams) and actor update semantics. Provenance information in COMAD is recorded during a workflow run by adding special provenance annotation tokens directly into the token stream. These tokens are added based on the insertions and deletions of actor invocations, and invocations explicitly declare dependencies for inserted tokens. Workflow traces generated by COMAD are collapsed, i.e., actors declare dependencies, insertions, and deletions for the "highest" relevant token in the tree, however, there is no guarantee that any given trace is minimal. Traces are serialized into XML, which are used by tools that display, navigate, and query trace files (e.g, see [6, 8]). Our provenance model implementation employed within COMAD also provides a number of extensions beyond the core model described here. These extensions include metadata annotations (name-value pairs) that can be associated with collection and data tokens, parameter annotations for recording invocation parameter values, and deletion dependencies. These additional annotations can also implicitly cascade (or propagate) to descendent nodes [8].

In the following, we describe strategies for efficiently storing and querying trace files that are generated by COMAD workflow runs or other tools supporting the provenance model. We focus on storage strategies for the core model presented here, although our approaches can also support the various extensions described above.

3. PROVENANCE STORAGE

Maintaining provenance information for all nodes in a workflow trace can lead to prohibitively expensive storage costs in terms of database size and update time (i.e., the time needed to load provenance information into the database) [10, 22, 15]. We show that these costs can be decreased using our provenance model by exploiting *trace equivalence* to collapse provenance annotations (Section 4). However, *reducing* the number of annotations in this way can lead to *increases* in query execution time, e.g., queries that need to access collapsed annotations will need to expand these annotations as part of the query.

Moreover, provenance queries often require dependencies to be transitively closed, e.g., to find the nodes or invocations that directly or indirectly contributed to a specific node. Storing dependency closures can further increase storage cost, whereas storing only immediate dependencies requires more expressive query constructs, such as recursion, that often increase query time [22].

We present optimizations for balancing this trade-off between storage size, update time, and query-response time. Our focus is on efficiently storing, loading, and querying individual traces within a relational database system. We also define a set of *strategies* for storing traces (Figure 7), each of which consist of: (i) workflow run identifiers; (ii) nested collection structures; (iii) insertions and deletions; (iv) immediate and transitive node dependencies; and (v) immediate and transitive invocation order. Similar to other approaches [10, 22], we define new techniques for storing direct and indirect node dependencies and invocation order, allowing prove-

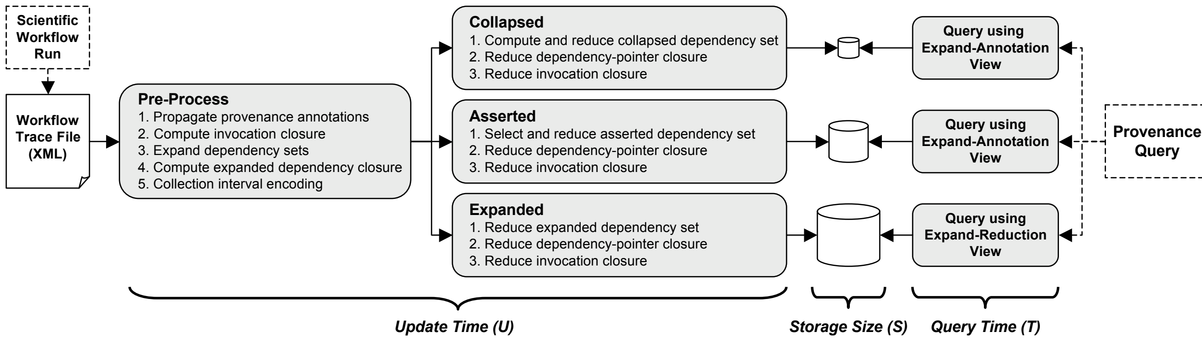


Figure 7: Basic trace storage strategies. Traces are stored in either their collapsed, asserted, or expanded form; reduction techniques are applied to minimize storage size; and relational views are used to rewrite provenance queries expressed against the store.

nance queries to be expressed using purely relational languages instead of the more typical approach of using expressive but less efficient recursive query constructs (e.g., [5, 9]). Section 4 uses these strategies and reduction techniques to compare the effectiveness of approaches for minimizing storage size, update time, and query time on real-world and synthetic traces.

3.1 Storage Strategies

The basic setting we consider for storing workflow traces is summarized in Figure 7. Given an XML file containing a workflow trace, we first perform a number of *pre-processing* tasks before loading the file into a database. Pre-processing includes parsing the trace file, applying trace inference rules, determining interval-encodings [25] for nested data collections, and computing transitive dependency and invocation-order closures. Interval encodings are used to efficiently access the ancestors and descendents of nested collections. Similarly, we store both direct and indirect node dependencies and invocation orders (although in a reduced form) to increase overall query performance and to allow lineage queries to be expressed using relational queries.

Once a trace is pre-processed, one of the following three storage strategies is selected (shown in the middle of Figure 7):

Collapsed Strategy. The first strategy stores a *collapsed*, but not minimal, representation of the trace. In particular, we only collapse the trace with respect to node dependency sets. The trace resulting from the pre-processing step is partially collapsed by removing all implied dependencies of nodes (applying the inverse of the dependency-expansion rule 9 of Figure 5). Note that in this strategy, node insertion and deletion annotations are still propagated to their children. Since a node can be inserted and deleted only once, it is not expensive to store the insertions and deletions for all nodes. However, because nodes can have many other nodes as their immediate dependencies, storing collapsed dependencies reduces the overall number of annotations.

Asserted Strategy. The second strategy stores the *asserted* trace, i.e., the input trace to the system (Figure 7). An asserted trace may contain redundant (i.e., implied) data dependency and invocation order annotations. For both the collapsed and asserted strategies, provenance queries are rewritten (via the *expand-annotation* view of Figure 7; see Section 3.3) to dynamically reconstruct the relevant portion of the expanded trace for the query. In the asserted strategy, we also record during the update process which nodes in the trace are expanded. Although not described further here, this approach allows the expand-annotation view to be applied only when an unexpanded portion of the trace is being queried.

Expanded Strategy. The third strategy stores the *fully-annotated* (i.e., *expanded*) trace, which includes all annotations that are derived from applying the inference rules of Figure 5.

Each of these three strategies employ space-reduction techniques for storing only once the common subsets of node dependency sets, node dependency closures, invocation orders, and invocation order closures. Similar to denormalization, these techniques require splitting dependency and closure relations into multiple tables. Provenance queries for each case are rewritten (via the *expand-reduction* view of Figure 7; see Section 3.3) to reconstruct the relevant portion of the original tables representing dependencies and invocation order. Update time (U) consists of the time required to pre-process the trace; compute interval encodings of nested collections; collapse or expand the trace; reduce common dependency and closure subsets; and load the resulting records into the database.

3.2 Reduction Techniques

Nodes within a trace often have similar sets of immediate dependencies. Using a straightforward representation that stores node-dependency pairs as tuples $R(N, D)$, each shared dependency node in R will be stored multiple times. For example, if n_1 and n_2 both depend on n_3 and n_4, R will contain the tuples $R(n_1, n_3)$, $R(n_1, n_4)$, $R(n_2, n_3)$, and $R(n_2, n_4)$, resulting in n_3 and n_4 being stored twice.

Below we describe four approaches that can be applied sequentially to reduce the redundancy of dependency sets. Each approach requires slight changes to the simple dependency schema $R(N, D)$ to minimize the number of duplicate node occurrences within a provenance database, thereby reducing the overall size of the database. While many possible reduction techniques can be considered, we employ the techniques below because they are simple enough to be applied efficiently at runtime and the reduced provenance store can be queried directly through relational views defined over the reduced schema. Table 1 demonstrates these techniques using four simple dependency sets. The unique dependency nodes in this example are 10, 20, 30, 40, and 50; whereas nodes 100, 200, 300, and 400 are dependent on subsets of these dependency nodes. We use the notation $n \to \{n_1, ..., n_k\}$ in the figure to denote tuples of the form $R(n, n_1), ..., R(n, n_k)$. When no reductions are applied (first row of Table 1), the dependencies given in the example consist of 5 unique dependency nodes requiring 17 total dependency nodes to be stored.

Duplicate-Set Reduction. This reduction technique is a common storage optimization (e.g., [10]) in which nodes indirectly relate to their dependency sets through "pointers" (i.e., integer identifiers x, denoted by $\&x$). Storage size is reduced by using the

Table 1: The result of sequentially applying reduction techniques on example dependency sets

Technique	*Input Dependencies*	*Nodes*	*Output Dependencies*	*Subsequence*	*Subset*	*Total Dep. Nodes*
(0) None	$100 \rightarrow \{10, 20, 30, 40, 50\}$ $200 \rightarrow \{10, 20, 30, 40, 50\}$ $300 \rightarrow \{10, 20, 30, 40\}$ $400 \rightarrow \{10, 30, 50\}$	100 200 300 400	-	-	-	17
(1) Duplicate-Set Reduction	$100 \rightarrow \{10, 20, 30, 40, 50\}$ $200 \rightarrow \{10, 20, 30, 40, 50\}$ $300 \rightarrow \{10, 20, 30, 40\}$ $400 \rightarrow \{10, 30, 50\}$	$100 \rightarrow \&1$ $200 \rightarrow \&1$ $300 \rightarrow \&2$ $400 \rightarrow \&3$	$\&1 \rightarrow \{10, 20, 30, 40, 50\}$ $\&2 \rightarrow \{10, 20, 30, 40\}$ $\&3 \rightarrow \{10, 30, 50\}$	-	-	12
(2a) Subsequence Reduction Applied to result of (1)	$\&1 \rightarrow \{10, 20, 30, 40, 50\}$ $\&2 \rightarrow \{10, 20, 30, 40\}$ $\&3 \rightarrow \{10, 30, 50\}$	$100 \rightarrow \&1$ $200 \rightarrow \&1$ $300 \rightarrow \&2$ $400 \rightarrow \&3$	$\&1 \rightarrow \{10, 20, 30, 40, 50\}$ $\&3 \rightarrow \{10, 30, 50\}$	$\&2 \rightarrow \&1, [10, 40]$	-	10
(2b) Subset Reduction Applied to result of (1)	$\&1 \rightarrow \{10, 20, 30, 40, 50\}$ $\&2 \rightarrow \{10, 20, 30, 40\}$ $\&3 \rightarrow \{10, 30, 50\}$	$100 \rightarrow \&1$ $200 \rightarrow \&1$ $300 \rightarrow \&2$ $400 \rightarrow \&3$	$\&1 \rightarrow \{50\}$ $\&2 \rightarrow \{10, 20, 30, 40\}$ $\&3 \rightarrow \{10, 30, 50\}$	-	$\&1 \rightarrow \&2$	8
(3) Subsequence-Subset Reduction Applied to result of (2a)	$\&1 \rightarrow \{10, 20, 30, 40, 50\}$ $\&3 \rightarrow \{10, 30, 50\}$	$100 \rightarrow \&1$ $200 \rightarrow \&1$ $300 \rightarrow \&2$ $400 \rightarrow \&3$	$\&1 \rightarrow \{20, 40\}$ $\&3 \rightarrow \{10, 30, 50\}$	$\&2 \rightarrow \&1, [10, 40]$	$\&1 \rightarrow \&3$	7

same pointer for each node that depends on the same set of dependencies. In Table 1, applying this technique reduces the total number of nodes stored to 12. Duplicate-set reduction divides the original dependency relation into two distinct relations $R_N(N, \&P)$ and $R_D(\&P, D)$, represented in Table 1 using $n \rightarrow \&p$ and $\&p \rightarrow \{n_1, ..., n_k\}$. Algorithm 1 describes this technique in more detail. The duplicate-set algorithm takes $O(N)$ time, where N is the number of nodes in the trace.

Algorithm 1 *Duplicate-Set Reduction*

Input: Fully-annotated trace T
Output: Dictionaries D and P

1: $D \leftarrow [\,],\ P \leftarrow [\,],\ S \leftarrow [\,],\ i \leftarrow 1$
2: **for** n in nodes of T **do**
3: $\quad D_n \leftarrow \{d \mid \mathsf{dep}(n, d)\}$
4: $\quad$ **if** $S[D_n] = \text{nil}$ **then**
5: $\quad\quad D[n] \leftarrow \&i,\ P[\&i] \leftarrow D_n,\ S[D_n] \leftarrow \&i$
6: $\quad\quad i \leftarrow i + 1$
7: $\quad$ **else**
8: $\quad\quad D[n] \leftarrow S[D_n]$ /* *reduction step* */
9: $\quad$ **end if**
10: **end for**

Subsequence Reduction. This reduction technique identifies dependency sets that represent subsequences of larger dependency sets, and stores only the range of the subsequence relative to the larger set. This technique is applied after duplicate-set reduction. Each unique dependency set is sorted (e.g., using node identifiers) to identify subsequences. The approach stores the largest dependency sequences and creates a new relation $R_{subseq}(\&P_{from}, \&P_{to}, i, j)$ to store subsequences in which i and j denote the (inclusive) range of the subsequence. Only subsequences of length 3 or greater are considered. In Table 1, applying this technique reduces the total number of nodes stored to 10. Subsequence reduction (cf. Section 4) is effective in cases where actors perform a form of running aggregation, either within their invocation or with respect to the overall token stream. Algorithm 2 describes this technique in more detail. Subsequence reduction takes $O(|P|^2)$ time, in which for each unique dependency set (where $|P|$ is the number of unique sets) the largest containing sequence is found.

Subset Reduction. This reduction technique identifies dependency sets that are proper subsets of larger dependency sets. This technique is also applied after duplicate-set reduction. Unlike in subsequence reduction, this approach stores the smaller set and creates a new relation $R_{subset}(\&P_{from}, \&P_{to})$ such that $P_{to} \subseteq P_{from}$, where P denotes the dependency set pointed to by $\&P$. Only subsets of length 2 or greater are considered. We also only permit one level of subset reduction, i.e., a dependency set that is stored as a subset of another set cannot itself be represented through a subset. If the number of nested subset relationships is not held constant (in our case we allow one level of nesting), recursive views would be required in general to reconstruct dependency sets. In Table 1, applying our subset reduction technique reduces the total number of nodes stored to 8. Algorithm 3 describes this technique in more detail. Subset reduction takes $O(|P|^3)$ time, in which the subset that gives the largest reduction is identified in $O(|P|^2)$ time and in the worst case $|P|$ such subsets are found. In each pass, the subset that maximizes the product of its size and the number of sets that contain it is selected, and the set is not considered for reduction in subsequent passes.

Algorithm 2 *Subsequence Reduction*

Input: Dictionary P mapping pointers to *sorted* dependency sets
Output: Dictionaries P and S

1: $S \leftarrow [\,]$
2: **for** p_s in keys of P and $|P[p_s]| > 2$ **do**
3: $\quad \text{found} \leftarrow \mathit{false},\ \text{maxlen} \leftarrow 0,\ p_b \leftarrow \bot,\ D_s \leftarrow P[p_s]$
4: $\quad$ **for** p in keys of P and $p \neq p_s$ **do**
5: $\quad\quad D_b \leftarrow P[p]$
6: $\quad\quad$ **if** D_s is a subsequence of D_b and $|D_b| > \text{maxlen}$ **then**
7: $\quad\quad\quad \text{found} \leftarrow \mathit{true},\ p_b \leftarrow p,\ \text{maxlen} \leftarrow |D_b|$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: $\quad$ **if** *found* **then**
11: $\quad\quad i \leftarrow \text{first}(D_s),\ j \leftarrow \text{last}(D_s)$
12: $\quad\quad S[p_s] \leftarrow (p_b, i, j)$ /* *reduction step* */
13: $\quad\quad P[p_s] \leftarrow \emptyset$ /* *reduction step* */
14: $\quad$ **end if**
15: **end for**

Subseqence-Subset Reduction. This reduction technique first applies the subsequence reduction, and then applies the subset reduction to the result. In Table 1, applying this technique reduces the total number of nodes stored to 7. This approach always provides better reduction than using only subsequence reduction. However, in certain situations applying only the subset technique can provide better overall reduction than applying subsequence followed by subset (cf. Section 4). Thus, our current implementation applies both techniques (during the pre-processing phase) and selects the

Algorithm 3 *Subset Reduction*

Input: Dictionary P mapping pointers to dependency sets
Output: Dictionaries P and S

```
1: S ← [ ], savings ← true, M ← ∅
2: while savings do
3:     savings ← false, r ← 0, p_s ← ⊥, B ← ∅,
4:     for p'_s in keys of P, p'_s not in M, and |P[p'_s]| > 1 do
5:         B' ← ∅
6:         for p'_b in keys of P, p'_b ≠ p'_s, and p'_b not in M do
7:             if P[p'_s] ⊆ P[p'_b] then
8:                 B' ← B' ∪ {p'_b},
9:             end if
10:        end for
11:        if |P[p'_s]| * |B'| > r then
12:            savings ← true, r ← |P[p'_s]| * |B'|, p_s ← p'_s, B ← B'
13:        end if
14:    end for
15:    if savings then
16:        for p_b in B do
17:            P[p_b] ← P[p_b] − P[p_s], S[p_b] ← p_s    /* reduction step */
18:        end for
19:        M ← M ∪ {p_s}
20:    end if
21: end while
```

Table 2: Node and node dependency schemas

Before Reduction	*After Reduction*
node(R, N, I_{ins}, I_{del})	r-node(R, N, I_{ins}, I_{del}, $\&P_{dep}$, $\&P_{depc}$)
dep(R, N, N_{dep})	r-dep(R, $\&P_{dep}$, N_{dep}) r-dep-subseq(R, $\&P^{from}_{dep}$, $\&P^{to}_{dep}$, i, j) r-dep-subset(R, $\&P^{from}_{dep}$, $\&P^{to}_{dep}$)
depc(R, N, N_{depc})	r-depc(R, $\&P_{depc}$, $\&P_{dep}$) r-depc-subseq(R, $\&P^{from}_{depc}$, $\&P^{to}_{depc}$, i, j) r-depc-subset(R, $\&P^{from}_{depc}$, $\&P^{to}_{depc}$)

result with better overall reduction. Similar to subset reduction, this algorithm runs in $O(|P|^3)$ time.

Closure Reduction. The above reduction techniques can be generically applied to any transitive binary relation. We apply the reductions to immediate data dependencies, dependency closures, immediate invocation order, and invocation-order closures. A portion of the schema we use to represent immediate data dependencies is shown in Table 2 (for storing nodes, and immediate and transitive dependencies); similar relations are defined for immediate invocation order and its closure. As shown in Table 2, we employ the reduction techniques over closures of pointers instead of closures of nodes (similarly, for invocations). This approach results in an overall smaller representation compared to directly reducing node closures, since pointer closures do not store nodes and also do not store the immediate-node reduction.

For example, the result of applying the duplicate-set reduction technique to the trace of Figure 3 gives the immediate dependencies $17 \rightarrow \&3$, $12 \rightarrow \&2$, and $6 \rightarrow \&1$ such that $\&3 \rightarrow \{12, 13, 14\}$, $\&2 \rightarrow \{6, 7, 8\}$, and $\&3 \rightarrow \{3, 4, 5\}$. The transitive dependency closure for node 17 is $\{3, 4, 5, 6, 7, 8, 12, 13, 14\}$, which is represented in our approach by the set of immediate dependency pointers $\{\&1, \&2, \&3\}$. We further reduce these pointer-based closures using our reduction techniques, resulting in a representation that is smaller than the corresponding reduction of the node-based closure.

As shown in Table 2, we store multiple workflow traces that are distinguished by a run identifier R. Additionally, each node is stored with its insertion and deletion annotation as well as its immediate-dependency $\&P_{dep}$ and transitive-dependency $\&P_{depc}$ pointers.

```
node(R, N, I_ins, I_del)  :-  r-node(R, N, I_ins, I_del, _, _).
dep(R, N, N_dep)          :-  r-node(R, N, _, _, P, _), v-dep(R, P, N_dep).
v-dep(R, P, N)            :-  v-dep_0(R, P, N).
v-dep(R, P, N)            :-  r-dep-subseq(R, P, P_to, I, J), v-dep_0(R, P_to, N),
                              N ≥ I, N ≤ J.
v-dep_0(R, P, N)          :-  r-dep(R, P, N).
v-dep_0(R, P, N)          :-  r-dep-subset(R, P, P_to), r-dep(R, P_to, N).
```

Figure 8: Expand-reduction view for immediate dependencies.

```
depc(R, N, N_depc)        :-  r-node(R, N, _, _, _, P), v-depc(R, P, P_dep),
                              v-dep(R, P_dep, N_depc).
v-depc(R, P, P_dep)       :-  v-depc_0(R, P, P_dep).
v-depc(R, P, P_dep)       :-  r-depc-subseq(R, P, P_to, I, J), v-depc_0(R, P_to, P_dep),
                              P_dep ≥ I, P_dep ≤ J.
v-depc_0(R, P, P_dep)     :-  r-depc(R, P, P_dep).
v-depc_0(R, P, P_dep)     :-  r-depc-subset(R, P, P_to), r-depc(R, P_to, P_dep).
```

Figure 9: Expand-reduction view for transitive dependencies.

3.3 Rewriting Provenance Queries

For expanded traces, provenance queries over the schema shown on the left-side of Table 2 are answered using the view definition expressed over the schema for storing reductions (Table 2, right). We consider the case of node dependencies, however, invocation-order is handled in a similar way. The views are relational queries written in non-recursive Datalog and can be expressed, e.g., using SQL. The view definitions in Figure 8 are used to recover nodes and immediate dependencies. Similarly, the view definitions in Figure 9 are used to recover dependency closures.

We also define a relational view in Figure 10 for expanding traces stored using the collapsed (or asserted) strategy. A relational query can be used to expand a collapsed trace since: (i) we explicitly store the transitive closure of the invocation order; (ii) we store the insertion and deletion annotations of each node directly within the node relation; (iii) we store the dependency set pointer of each node directly within the node relation; and (iv) we use interval encodings for nested collections. Thus, a traced stored via the collapsed strategy can be expanded by simply applying rule 9 in Figure 5, since (i), (ii), and (iii) already store the provenance information that would be derived from the remaining rules. We use (iv) to access collection descendents instead of children as in rule 9 of Figure 5. The variable X is the insertion invocation of N, and Y and Z are the insertion and deletion invocations of node C, respectively. Node C here is a descendent of N_{dep}, where N_{dep} corresponds to node p in rule 9 of Figure 5. The inv-beforec relation stores the transitive closure of the invocation order. Expanding dependency sets is performed using the expand-reduction view, which accesses the reduced representation of the database. The view in Figure 10 is applied only in the collapsed and asserted strategies of Figure 7, since the expanded strategy stores fully-annotated traces.

4. EXPERIMENTAL EVALUATION

Here we evaluate our provenance model, storage strategies, and reduction techniques on both real and synthetic traces. Real traces were generated from existing workflows implemented within the Kepler system (see [6, 33]). We compare our storage strategies to the conventional provenance model discussed in Section 1 as well as to approaches that do not employ our reduction techniques. Our experiments evaluate the various approaches in terms of storage size, update time, and query time. All experiments were performed

```
exp-dep(R, N, N_dep)   :-   dep(R, N, N_dep).
exp-dep(R, N, N_dep)   :-   dep(R, N, N_dep), descendent(N_dep, C),
                            node(R, N, Y, _), node(R, C, X, Z),
                            inv-beforec(R, X, Y), ¬inv-beforec(R, Z, Y).
exp-dep(R, N, N_dep)   :-   dep(R, N, N_dep), descendent(N_dep, C),
                            node(R, N, Y, _), node(R, C, null, Z),
                            ¬inv-beforec(R, Z, Y).
exp-dep(R, N, N_dep)   :-   dep(R, N, N_dep), descendent(N_dep, C),
                            node(R, N, Y, _),
                            node(R, C, X, null), inv-beforec(R, X, Y).
exp-dep(R, N, N_dep)   :-   dep(R, N, N_dep), descendent(N_dep, C),
                            node(R, N, _, _), node(R, C, null, null).
```

Figure 10: Expand-annotation view for dependencies.

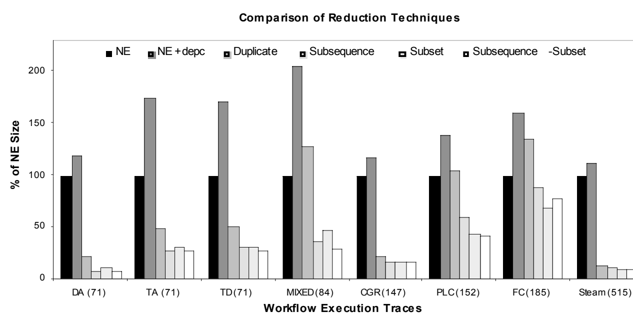


Figure 11: Comparison of reduction techniques.

using a 2.4GHz Intel Core 2 duo PC with 2 GB RAM and 120 GB disk space. Our implementation stores provenance information and base data using MySQL 5. Our algorithms were implemented in Java and use JDBC to communicate with the provenance database.

Table 3 lists the storage strategies we consider. These strategies were compared using a set of *synthetic traces* that consist of actors employing the dependency patterns shown in Table 4. For each pattern, we consider traces of increasing numbers of nodes, ranging from 100 to 6000, and dependency annotations ranging from 10^3 to 10^6 with 10^4 to 10^8 transitive dependency edges. We also evaluated our approaches using the following *real traces* from scientific workflows implemented within Kepler: the *CGR* and *PLC* workflows [6] infer phylogenetic trees from protein and morphological sequence data; the *FPC* workflow was used within the first provenance challenge [33]; and the *Steam* workflow characterizes microbial communities by clustering and identifying DNA sequences of 16S ribosomal RNA. The dependency annotations of these traces range from $2 \cdot 10^3$ to $2 \cdot 10^4$ with $5 \cdot 10^3$ to $5 \cdot 10^4$ transitive dependency edges. Our goal in defining the patterns of Table 4 is to demonstrate the range of reductions possible. In general, actual workflow traces will (including those considered here) contain one or more of these patterns.

Storage Size. Figure 12 shows the storage sizes observed for the TD, DA, and Mixed synthetic traces. Note that we only consider the conventional (Conv) approach for the TD case, since it cannot be used to correctly represent the other dependency patterns. The storage space for SE grows exponentially because it stores provenance and closure records for each node (which ranges from thousands to millions of records). The storage size for NE also grows exponentially, similarly requiring millions of records to store the larger traces. Alternatively, the storage size of RC and RE grows linearly, and is comparable to NC, although RC and RE require slightly less storage space. Thus, storing the closure information for all nodes results in exponential growth in storage space for SE, whereas our reduction techniques reduce the same information such that RE and RC have linear growth in storage space and both require less space than NC. Note that for strategies that collapse provenance annotations, less storage space is used than for their corresponding expanded strategies (e.g., NC versus NE, and RC versus RE). The relative ranking of the strategies in Figure 7 for storage size is RC < RE < NC < NE < SE.

Figure 11 shows the comparative reductions (to NE) achieved when different techniques are applied across each of the synthetic and real traces. Different techniques give minimal reduction depending upon the nature of the trace annotations. For example, for FC the subset reduction technique provides the greatest reduction, whereas in each of the other cases, the subsequence-subset technique provides the best reduction. Figure 11 also shows the additional space required to store pointer-based closures (NE+depc).

Update Time. Parsing trace files, expanding and collapsing annotations, and loading provenance records into the database comprise the update time for the NC, NE, and SE strategies. Update time for RE and RC consists of the time required to pre-process the trace, collapse or expand annotations, apply reduction techniques, and load the resulting records into the provenance database. Figure 15 shows the time taken to upload provenance records for the MIXED pattern traces. Similar to storage size, because SE must upload provenance and closure records for each node, its update time grows exponentially. NE also requires significant update time as the size of the traces increase. The update times for RC and RE are linear in the MIXED pattern and are less than that for NC (which does not store closures), demonstrating than the costs of pre-processing and applying reductions is low. As shown in Figure 15, the cost of applying our reduction algorithms grows linearly with increasing numbers of dependency annotations. While SE and NE do not incur the overhead associated with reduction, as shown in Figure 15 they exhibit exponential growth in update time over increasing numbers of annotations. RE and RC have relatively low update times for the real execution traces as shown in Figure 13(a). Similar to storage size, the relative ranking of these strategies with respect to update time is RC < RE < NC < NE < SE for the MIXED, TA, and TD patterns. For the DA pattern, dependencies grow exponentially resulting in RE and RC having a corresponding exponential increase in update time due to the cost of expanding annotations. For the DA pattern, the relative ranking of these strategies with respect to update time is NC < RC < RE < NE < SE.

Query Time. Because our reduction techniques partition the immediate dependency, transitive dependency, and invocation order tables into multiple tables, we must apply the views defined in Figures 8 and 9 to answer provenance queries. To evaluate the cost of applying these views, we consider three basic provenance queries. Given a specific node, we measure the query time to find: (1) the immediate dependencies of the node; (2) the transitive dependencies of the node; and (3) the sequence of invocations that contributed to the node being inserted. In our experience, (2) is the most widely used of these within provenance queries [7, 33]. Figure 14 shows the query response times for the MIXED pattern traces. Figure 14(a) shows that to find immediate dependencies, the RE strategy performs as good as the other methods (NE and SE). The overhead of applying the view in this case is the cost of joining the reduction tables. RC, however, takes significantly more time to find immediate dependencies, since the query must use the expand-annotation view which involves more join operations (Figure 10).

Figure 14(b) shows that for transitive dependencies, RE does as well as SE (again with only minor overhead). Because NE does not

Table 3: Provenance storage strategies

Strategy	*Label*	*Annotations*	*Dep. Closure*	*Reduced*	*Optimizes*
Simple Expanded	**SE**	Expanded	Node-Based	No	Immediate & Transitive Queries
Naive Expanded	**NE**	Expanded	No	No	Immediate queries
Naive Collapsed	**NC**	Collapsed Dependencies	No	No	Storage size
Reduced Expanded	**RE**	Expanded	Pointer-Based	Yes	Storage Size, Immediate & Transitive Queries
Reduced Collapsed	**RC**	Collapsed Dependencies	Pointer-Based	Yes	Storage Size

Table 4: Workflow execution trace patterns

Pattern	*Label*	*Dependencies*	*Removes Input Nodes from Stream*
Dependency All	**DA**	All output depend on all inputs in token stream at time of insertion	No
Transform Add	**TA**	All invocation outputs depend on all invocation inputs	No
Transform Delete	**TD**	All invocation outputs depend on all invocation inputs	Yes
Mixed Pattern	**MIXED**	Repeatedly perform DA followed by TA followed by TD	Yes (via TD)

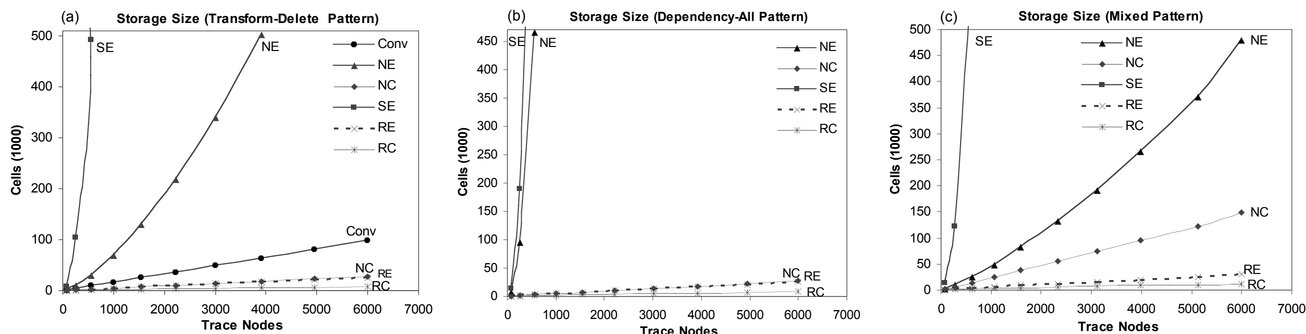


Figure 12: Provenance storage size for different patterns of synthetic traces.

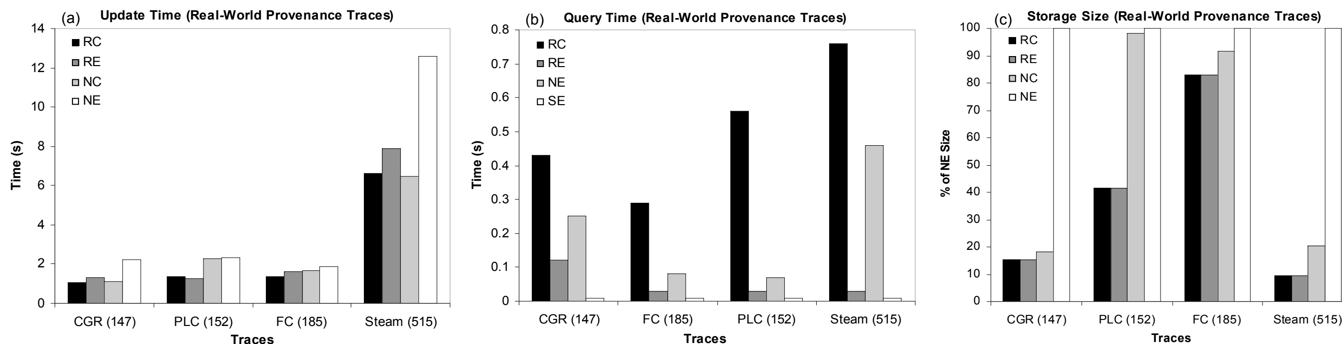


Figure 13: Update time, query response time, and storage size for real execution traces.

store the transitive closure, a recursive stored procedure is used to answer the query. In RC, the expand-annotation view must be used for all nodes in its lineage path. NC will take even more time to answer this query than RC, since each transitive dependency node must be identified recursively and the expand-annotation view must be applied to all such nodes.

In Figure 14(c), determing the sequence of invocations that led to the insertion of a node takes less than 0.1 seconds for NE, SE, RE, and RC even for larger traces. The reason for this low query time is that the number of invocations in a trace typically is only on the order of at most 100's of invocations.

Figure 13(b) shows the total query time across these queries for the real execution traces. As shown, RE and SE have almost the same query response time. The relative ranking of these strategies with respect to query time is SE < RE < NE < RC.

Analysis of Storage Strategies. NC optimizes storage and update time, but requires recursion (e.g., via stored procedures) to answer both immediate and transitive dependency queries, and thus exhibits the worst query response time. NE is better than NC for answering direct provenance annotation queries, but also requires recursion to answer transitive dependency queries. In addition, as the number of provenance annotations increase, NE also leads to exponential growth in storage space and an increasingly high update time. SE optimizes query response time, but incurs exponential storage size and upload time, making it impractical except for small traces. RE optimizes all three metrics, i.e., storage space, update time, query response time. Applying the expand-reduction views for RE incurs only minor overhead for query response time. RC optimizes storage size and update time, but incurs significant overhead by applying the expand-reduction and expand-annotation views. The table below gives a relative ranking of these strategies for storage size, update time, and query response time. In particular, SE is generally impractical with respect to storage size and

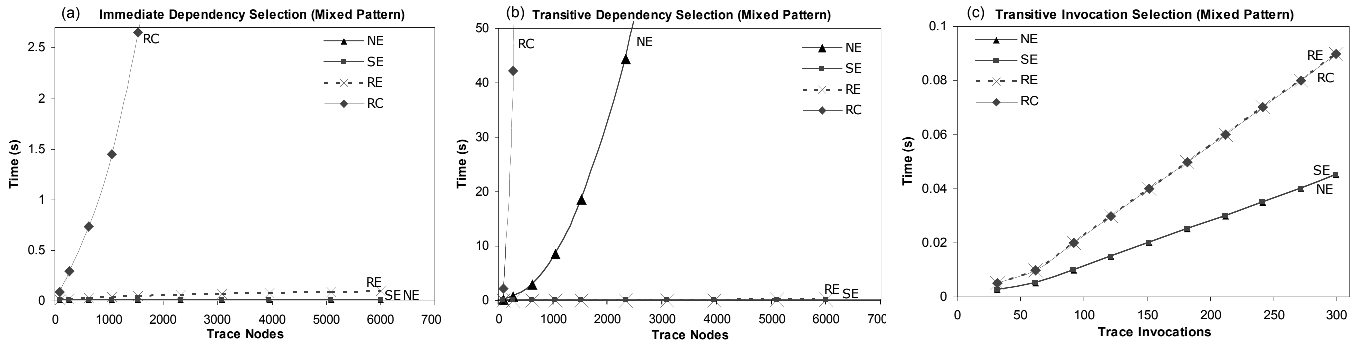


Figure 14: Query response time for MIXED pattern traces.

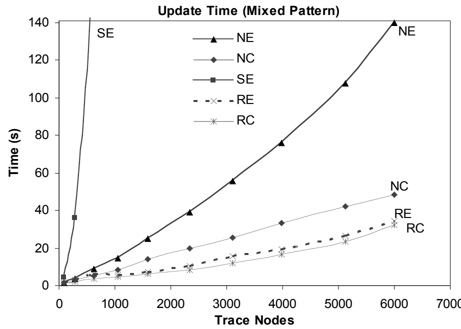


Figure 15: Provenance update time for the Mixed pattern.

update time, and executing queries using RC is expensive and provides only minor storage savings over RE. In addition, RE reduces the exponentially growing storage size of storing provenance and dependency relations to linearly growing storage size by applying reduction techniques. Thus, we consider RE to be the best of these strategies when considering all three criteria.

metric	ranking
storage	RC < RE < NC < NE < SE
update time	RC < RE < NC < NE < SE
query time	SE < RE < NE < RC < NC

5. RELATED WORK

Scientific workflow approaches are used in many science domains; and numerous approaches have recently been proposed to represent provenance information generated from scientific workflows runs (e.g., [38, 2, 33, 15, 16, 37, 4, 40, 41, 35, 13, 3, 7, 34]). Most of these approaches adopt a "conventional" provenance model in which invocation outputs are assumed to depend on *all* associated inputs, and invocations only produce *new* data, i.e., they do not employ an "update" semantics. In [31] and [32], these assumptions are relaxed such that *constraints* are used to describe anticipated dependency patterns of actors. In contrast, our goal is to record *explicit* invocation dependencies—generated, e.g., by "dependency-aware" actors or by inference-based approaches over state information or read/write timestamps [7, 2]—thereby supporting the many possible patterns of data dependencies that can be recorded for actor invocations.

Our provenance model is inspired by approaches for reference-based versioning of XML [12, 28, 9] and semistructured data [11], and employs a versioning model that supports only a small set of operations over XML structures while adding new annotations for data and invocation dependency annotations. While many XML versioning approaches allow *ad hoc* restructuring and updates to data values (attributes and text nodes), our approach only permits operations for *inserting* and *deleting* subtrees. However, this more accurately captures the operations permitted on token streams in dataflow-based computation models [24, 27], and supports concurrent actor execution by allowing for partially-ordered versions.

With the adoption of scientific workflow approaches and the need for recording large amounts of provenance information, techniques have recently been proposed to efficiently store and query provenance metadata [10, 22, 15]. The approach in [10] can reduce the storage size of process (i.e., invocation) lineage, assuming a restricted version of the conventional provenance model is used where process lineage is defined by *sequences* of invocations; whereas most other provenance models support DAG-based lineage structures (e.g., [2, 34, 13, 7]). XML is used to represent data in [10], where nodes are stored with the sequence of processes that led to their creation. These annotations are reduced by collapsing annotations to the highest associated node in a tree (an often-used approach, e.g., see [8, 9] among others). Similarly, "basic factorization" is used in [10] to maintain only a single copy of identical provenance annotations (process sequences). We introduce new algorithms not considered in [10] that exploit dependency and pointer-based closure sets that can be expressed as subsequences and/or subsets of one or more existing sets. Thus, our reduction algorithms are applied to both immediate and transitive data and invocation dependencies (in general, arbitrary binary relations), whereas those of [10] only apply to simple sequences of processes. Also, the goal in [10] is to reduce the size of an existing provenance store, whereas we include semantics to minimize annotations within our provenance model so that reduction techniques can be applied before a trace is loaded into a database, thus allowing both storage size and update time to be optimized.

In [22], a reduction technique is proposed that is generically applicable to transitive, acyclic binary relations. These techniques employ complex graph manipulations (as opposed to our approaches that are based on dependency sets) to reduce the number of duplicate nodes in a dependency graph. When compared with their algorithms, our reduction techniques on the examples given in [22] provide similar reductions. For example, on the fMRI [33, 22] transitive dependency graph, our reduction techniques store only 60 total nodes, whereas their approach stores 107 total nodes [22]. The reduction approaches both store 22 total nodes for the main example used in [22]. Both approaches support lineage queries using standard relational constructs, whereas most others (e.g., [10, 9, 2]) require recursive queries to reconstruct lineage graphs.

6. CONCLUSION

We have proposed a framework to address open issues in the representation and storage of provenance information for scientific workflows. Specifically, we have described a formal provenance model that generalizes the conventional model used for representing data and process provenance from workflow runs, by supporting a wider range of workflow types and workflow systems. Our provenance model does not limit invocations to simple "black-box" transformers, but allows the workflow system to declare and record data dependencies, based on an underlying update semantics. Our approach also simplifies provenance recording by not requiring implied provenance information to be given explicitly, while at the same time allowing detailed provenance to be recorded efficiently.

This paper extends our prior work [8, 6] by: (i) developing a formal description of our provenance model together with a set of first-order inference rules for expanding and collapsing traces; (ii) a set of strategies for efficiently storing and querying instances of the model; (iii) novel reduction techniques for minimizing redundancy in direct and transitive dependency representations; and (iv) a detailed experimental evaluation of our approach. Our reduction techniques can be applied efficiently and are applicable to both hierarchical and non-hierarchical data structures, and our results show that these reduction techniques can decrease provenance storage size, update time, and query-response time.

Acknowledgements. This work was supported through NSF grants IIS-0630033, OCI-0722079, IIS-0612326, DBI-0533368, and DOE grant DE-FC02-07-ER25811.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] I. Altintas, O. Barney, E. Jaeger-Frank. Provenance collection support in the Kepler scientific workflow system. *IPAW*, 2006.

[3] R. S. B. Barga and L. A. Digiampietri. Automatic capture and efficient storage of e-Science experiment provenance. *Concurrency and Computation: Practice and Experience*, 20(5):419–429, 2008.

[4] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, J. Freire. Vistrails: Enabling interactive multiple-view visualizations, *IEEE Visualization*, 2005.

[5] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. *ICDE*, 2008.

[6] S. Bowers, T. McPhillips, S. Riddle, M. Anand, B. Ludäscher. Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. *IPAW*, 2008.

[7] S. Bowers, T. M. McPhillips, B. Ludäscher, S. Cohen, S. B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. *IPAW*, 2006.

[8] S. Bowers, T. M. McPhillips, M. Wu, B. Ludäscher. Project histories: Managing data provenance across collection-oriented scientific workflow runs. *DILS*, LNCS, 2007.

[9] P. Buneman, A. Chapman, J. Cheney. Provenance management in curated databases. In *SIGMOD*, pp. 539–550. ACM, 2006.

[10] A. Chapman, H. V. Jagadish, P. Ramanan. Efficient provenance storage. In *SIGMOD*, pp. 993–1006, 2008.

[11] S. S. Chawathe, S. Abiteboul, J. Widom. Representing and querying changes in semistructured data. In *ICDE*, pp. 4–13, 1998.

[12] S.-Y. Chien, V. J. Tsotras, C. Zaniolo. Efficient schemes for managing multiversion XML documents. *VLDB Journal*, 11(4):332–353, 2002.

[13] S. Cohen, S. C. Boulakia, S. B. Davidson. Towards a model of provenance and user views in scientific workflows. In *DILS*, pages 264–279, 2006.

[14] Y. Cui, J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1):41–58, 2003.

[15] S. B. Davidson, J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pp. 1345–1350, 2008.

[16] E. Deelman, A. L. Chervenak. Data management challenges of data-intensive scientific workflows. *CCGRID*, pp. 687–692, 2008.

[17] R. Duan, R. Prodan, and T. Fahringer. Run-time optimisation of grid workflow applications. In *GRID*, pp. 33–40, 2006.

[18] W. Fan, G. Cong, and P. Bohannon. Querying XML with update syntax. *SIGMOD*, pp. 293–304, 2007.

[19] Y. Gil, E. Deelman, M. A. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. A. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):24–32, 2007.

[20] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *AAAI*, pp. 1767–1774, 2007.

[21] S. Gurmeet, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *J. Grid Comput.*, 3(3–4):201–219, 2005.

[22] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD*, pp. 1007–1018, 2008.

[23] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. Petri net + nested relational calculus = dataflow. In *OTM Conferences*, LNCS 3760, pp. 220–237, 2005.

[24] W. M. Johnston, J. R. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 204.

[25] C. Koch. Processing queries on tree-structured data efficiently. *PODS*, pp. 213–224, 2006.

[26] C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLDB Journal*, 16(3):317–342, 2007.

[27] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience*, pp. 1039–1065, 2006.

[28] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an xml warehouse. In *VLDB*, pp. 581–590, 2001.

[29] T. McPhillips, S. Bowers, D. Zinn, B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 2008.

[30] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Ngoc. Exchanging intensional XML data. *TODS*, 30(1):1–40, 2005.

[31] A. Misra, M. Blount, A. Kementsietsidis, D. Sow, and M. Wang. Advances and challenges for scalable data provenance in stream processing systems. In *IPAW*, LNCS, 2008.

[32] P. Missier, K. Belhajjame, J. Zhao, and C. Goble. Data lineage model for taverna workflows with lightweight annotation requirements. In *IPAW*, LNCS, 2008.

[33] L. Moreau and *et al.* The first provenance challenge. *Concurrency and Computation: Practice and Experience – Special Issue on the First Provenance Challenge*, 20(5), 2008.

[34] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson. The open provenance model. Technical Report 14979, ECS, University of Southampton, 2007.

[35] L. J. Osterweil, L. A. Clarke, R. Podorozhny, A. Wise, E. Boose, A. M. Ellison, and J. Hadley. Experience in using a process language to define scientific workflow and generate dataset provenance. In *Intl. Symp. on Foundations of Software Engineering*, 2008.

[36] J. Qin and T. Fahringer. Advanced data flow support for scientific grid workflow applications. In *ACM/IEEE Conf. on Supercomputing*, 2007.

[37] C. Silva, J. Freire, and S. P. Callahan. Provenance for visualizations: Reproducibility and beyond. *Computing in Science & Engineering*, 9(5):82–89, 2007.

[38] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[39] N. Walsh, A. Milowski, and H. S. T. (editors). XProc: An xml pipeline language. W3C Working Draft, May 2008.

[40] J. Zhao, C. Goble, R. Stevens, and D. Turi. Mining Taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience*, 20(5):463–472, 2008.

[41] Y. Zhao, M. Wilde, and I. Foster. Applying the virtual data provenance model. In *IPAW*, LNCS 4145. Springer, 2006.