# Database Support for Exploring Scientific Workflow Provenance Graphs

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

[1] Microsoft Corporation, Redmond, WA, USA
[2] Dept. of Computer Science, Gonzaga University, Spokane, WA, USA
[3] Dept. of Computer Science, University of California, Davis, CA, USA

**Abstract.** Provenance graphs generated from real-world scientific workflows often contain large numbers of nodes and edges denoting various types of provenance information. A standard approach used by workflow systems is to visually present provenance information by displaying an entire (static) provenance graph. This approach makes it difficult for users to find relevant information and to explore and analyze data and process dependencies. We address these issues through a set of abstractions that allow users to construct specialized views of provenance graphs. Our model provides operations that allow users to expand, collapse, filter, group, and summarize all or portions of provenance graphs to construct tailored provenance views. A unique feature of the model is that it can be implemented using standard relational database technology, which has a number of advantages in terms of supporting existing provenance frameworks and efficiency and scalability of the model. We present and formalize the operations within the model as a set of relational queries expressed against an underlying provenance schema. We also present a detailed experimental evaluation that demonstrates the feasibility and efficiency of our approach against provenance graphs generated from a number of scientific workflows.

## 1 Introduction

Most scientific workflow systems record provenance information, i.e., the details of a workflow run that includes data and process dependencies [11,20]. Provenance information is often displayed to users as a (static) dependency graph [16,13,7]. However, many real-world scientific workflows result in provenance graphs that are large (e.g., with upwards of thousands of nodes and edges) and complex due to the nature of the workflows, the number of input data sets, and the number of intermediate data sets produced during a workflow run [10,11], making them inconvenient to explore visually.

The goal of the work described here is to help users more easily explore and analyze provenance information by allowing them to specify and navigate between different abstractions (or *views*) of complex provenance graphs. Specifically, we describe a set of abstraction mechanisms and operators for scientific workflow provenance graphs that allow users to create, refine, and navigate between different views of the same underlying provenance information. We consider the following levels of granularity: (1) A workflow *run* represents the highest level of abstraction; (2) An *actor dependency graph* consists of the types of processes (actors) used in a workflow run and the general

A. Ailamaki and S. Bowers (Eds.): SSDBM 2012, LNCS 7338, pp. 343–360, 2012.

flow of data between them; (3) An *invocation dependency graph* consists of individual processes (invocations) within a run and their implicit data dependencies; (4) A *structure flow graph* consists of the actual data structures input to and output by each invocation; and (5) A *data dependency graph* consists of the detailed data dependencies of individual data items. In addition, we consider navigation operations that allow all or a portion of each provenance view to be *expanded* or *collapsed*, *grouped* into composite structures, *filtered* using a high-level provenance query language, and *summarized* through aggregation operators.

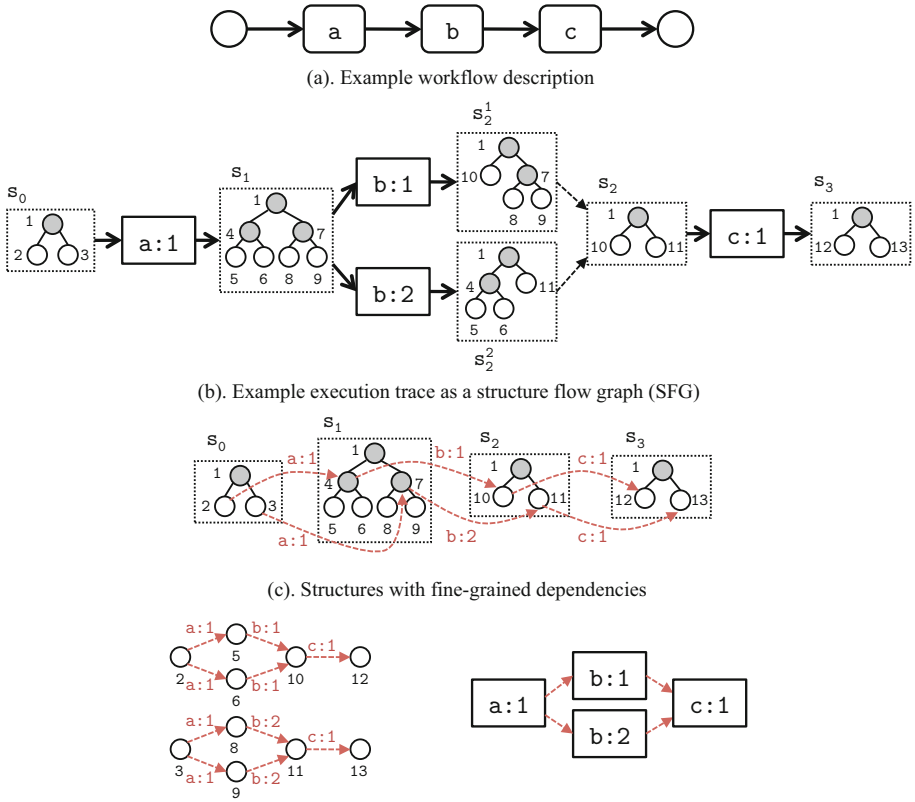
**Contributions.** We present a general provenance model that enables users to explore and analyze provenance information through novel graph-based summarization, navigation, and query operators. We also show how this model can be implemented using standard relational database technology. We adopt a relational implementation for two reasons. First, many existing workflow systems that record provenance store this information within relational databases [11], making it relatively straightforward to adopt the implementation described here. Second, the use of standard database technology can provide advantages in terms of efficiency and scalability over other general-purpose graph-based approaches (e.g., [12]) that provide only in-memory implementations. Finally, we demonstrate the feasibility of our implementation through an experimental evaluation of the operators over real and synthetic scientific-workflow traces.

**Organization.** The basic provenance model and query language that our view abstractions and navigation operators are based on is presented in Section 2. The different levels of abstraction and navigation operators supported by our model is defined in Section 3. We show how our model can be implemented within a relational framework in Section 4, which describes the relational schemas used to store provenance information and the queries used to execute each of the navigation operations and views. Our experimental results are presented in Section 5, which demonstrates the feasibility and scalability of the implementation. Related work is discussed in Section 6 and we conclude in Section 7.

## 2 Preliminaries: Provenance Model and Query Language

In our provenance model, we assume that workflows execute according to standard dataflow-based computation models (e.g., [15,17]). In addition, the provenance model supports processes that can be executed (i.e., *invoked*) multiple times in a workflow run and can receive and produce data products that are *structured* according to labeled, nested data collections (e.g., as XML). To help illustrate, consider the simple workflow definition shown in Fig. 1a. This workflow consists of three actors `a`, `b`, and `c`; distinguished input and output nodes; and four data-flow channels. The channels constrain how data is passed between actors within a workflow run. In paticular, the input to the run is passed to invocations of `a`, the results of `a`'s invocations are passed to invocations of `b`, and so on.

Fig. 1b shows a high-level view of an example run of this workflow, called a *structure flow graph* (SFG). As shown, each actor invocation receives a nested-collection data structure, performs an update on a portion of the structure (by either adding or




(a). Example workflow description

(b). Example execution trace as a structure flow graph (SFG)

(c). Structures with fine-grained dependencies

(d). Data dependency graph (DDG) (e). Invocation dependency graph (IDG)

**Fig. 1.** An example workflow graph (a) with a corresponding structure flow (b) and fine-grained dependency (c) graph together with associated data (d) and invocation (e) dependency graphs

removing items), and then passes the updated version to downstream actors. An SFG consists of the intermediate data structures $s_i$ that were input to and output by actor invocations, where edges denote "coarse-grained" dependencies [2]. In this example, the first invocation a:1 of actor a takes as input $s_0$ and produces the updated version $s_1$. The actor b is invoked twice to produce the modified structure $s_2$. The first invocation b:1 removes item 6 in $s_1$ and adds item 10, and similarly, the second invocation b:2 removes item 7 and adds item 11. The structure $s_2$ is the result (union) of the two independent modifications $s_2^1$ and $s_2^2$. Finally, invocation c:1 modifies $s_2$ to produce the output of the run $s_3$. This use of nested data collections within scientific workflows is supported within both Kepler [7] and Taverna [18] as well as more recent approaches such as [2]. These systems also often support independent invocations as in Fig. 1, where actor b is "mapped" over its input structure such that each invocation of b is applied independently to a specific sub-collection of b's input. Workflow systems that support these and other types of iterative operations (e.g., [17,5]) typically require each of the independent invocations to process a non-overlapping portion of the input to avoid downstream structure conflicts.

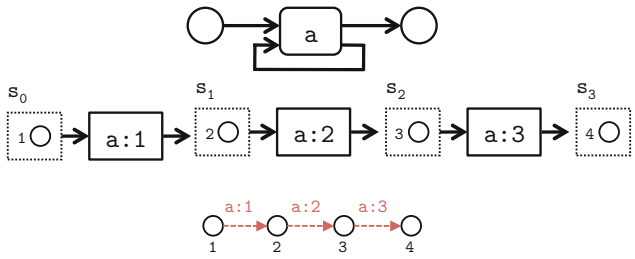



**Fig. 2.** A cyclic workflow graph (a) with a corresponding SFG (b) and data dependency graph (c)

In addition to coarse-grained dependency information, many applications of provenance [2,17,4] also require "fine-grained" dependencies. For instance, invocation `a:1` in Fig. 1b resulted in two new collections (items 4 and 7). However, without also capturing fine-grained dependencies, it is unclear which items in the input of `a:1` were used to derive these new collections. Fig. 1c shows the fine-grained (or explicit) dependencies for each structure in our example run. For example, the arrow from data item 2 in $s_0$ to collection item 4 in $s_1$ states that 4 (including its containing items) was created from (i.e., depended on) 2 via invocation `a:1`, whereas item 7 was introduced by `a:1` using item 3. As another example, the dependency from item 4 in $s_1$ to item 10 in $s_2$ states that item 10 was created by invocation `b:1` using item 4 and its containing items 5 and 6. Note that from Fig. 1c it is possible to recreate the SFG in Fig. 1b as well as the other views including the standard data and invocation dependency graphs of Fig. 1d and 1e.

A workflow execution *trace* is represented as a fine-grained dependency graph, i.e., the trace stores the information shown in Fig. 1c. Each trace stores the data structures input to and produced by the workflow run and the corresponding fine-grained dependencies among structures. The parameter values supplied for each invocation are also stored within a trace (not shown in Fig. 1). A trace can be represented in a more condensed form by only storing data and collection items shared by intermediate structures once together with special provenance annotations for item insertions (including dependency information) and deletions [5]. This approach is similar to those for XML-based version management (based on storing "diffs"). The provenance model is able to represent a large number of workflow patterns and constructs, including iteration and looping. Fig. 2 gives a simple example of a trivial iterative calculation in which an actor `a` is invoked repeatedly until it reaches a fixed point (the output value computed is the same as the input value).

The provenance model supports queries expressed using the *Query Language for Provenance* (QLP) [4]. In QLP, queries are used to filter traces based on specific data or collection items, fine-grained dependencies, and the inputs and outputs of invocations. QLP is similar in spirit to tree and graph-based languages such as XPath and generalized path expressions [1]. Unlike these approaches, however, QLP queries are closed under dependency edges. That is, given a set of dependency edges (defining a fine-grained dependency graph), a QLP query selects and returns a subset of dependencies denoting

the corresponding subgraph. This approach provides advantages in terms of supporting incremental querying (e.g., by treating queries as views) and for query optimization [4].

As an example of QLP, the query "$*$ .. 10" returns the set of dependency edges within a fine-grained dependency graph that define paths starting from any item in a data structure and ending at item 10.[1] Expressed over the trace graph in Fig. 1c, this query returns the dependencies $(2, \texttt{a:1}, 4)$, $(4, \texttt{b:1}, 10)$, $(5, \texttt{b:1}, 10)$, and $(6, \texttt{b:1}, 10)$. Similarly, the query "$*$ .. 4 .. $*$" returns all dependencies defining paths that start at any data-structure item, pass through item 4, and end at any item in the trace. For Fig. 1, this query returns the same dependencies as the previous query plus the additional dependency $(10, \texttt{c:1}, 12)$. In addition to items in data structures, QLP allows paths to be filtered by invocations. For example, the query "#a .. #b:1 ..$*$" returns dependencies that define paths starting at input items of any invocation of actor a, contain a dependency edge labeled by invocation b:1, and end at any data-structure item. Applied to the example in Fig. 1, this query returns the same set of dependencies as the previous query. QLP uses "@in" and "@out" to obtain the inputs and outputs, respectively, of invocations or runs. For example, the query "$*$ ..@in b:1" returns dependencies defining paths from any item in a structure to an input item of the first invocation of actor b. Similarly, the query "@in .. 10" returns the dependencies defining paths that start at any item within an input data structure of the run and that end at item 10.

## 3 Operators for Exploring Workflow Provenance Graphs

While languages such as QLP can help users quickly access and view relevant parts of large provenance graphs, doing so requires knowledge of the graph prior to issuing a query. When a user does not know ahead of time the parts of the graph that are of interest, or would like to create summarized views of only certain parts of a provenance graph to place the relevant portions in context, additional techniques beyond basic query languages are required. Here we describe extensions to the provenance model of the previous section to help support users as they explore provenance graphs. We consider both specific operations for transforming a provenance graph as well as a set of default views. Using these extensions, a user can switch (or *navigate*) to different views of all or a portion of the provenance graph by applying the transformation operators, or by navigating directly to any of the default views (bringing the current view to the same level of granularity). Given a transformation operator, a new view is constructed using the navigate function. If $v_i$ is the current provenance view, $t$ is the underlying trace, and $op$ is a transformation operator, $\texttt{navigate}(t, v_i, op) = v_{i+1}$ returns the new provenance view $v_{i+1}$ that results from applying $op$ to $v_i$ under $t$.

Fig. 3 shows the default views and their relationship to the transformation operators. In addition to the operators shown, we also consider operations for filtering views using QLP queries and for accessing summary data on current views. The rest of this section defines the default views of Fig. 3 and the various operations supported by the provenance model.

[1] In QLP, ".." specifies a fine-grained dependency path of one or more edges, "$*$" specifies any item in the trace, "#" specifies invocations, and "@" specifies data structures.

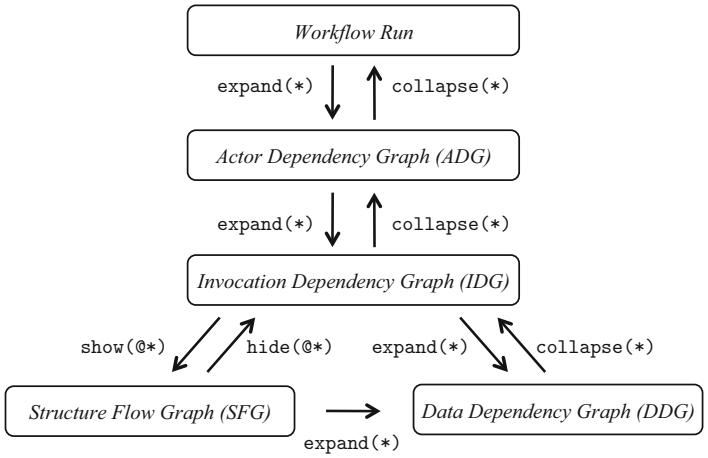



**Fig. 3.** Default views supported by the provenance model and corresponding navigation operators

**Default Provenance Views.** Navigation begins at a workflow run (e.g., see the top of Fig. 4). Expanding the run gives an *actor dependency graph* (ADG), which is similar in structure to a workflow graph, but where only actors that were invoked within the run are shown. Expanding an ADG produces an invocation dependency graph (IDG). An IDG consists of invocations and the implicit data dependencies between them, e.g., as in Fig. 1e. A structure flow graph (SFG) can be obtained from an IDG by showing all input and output structures of the trace, e.g., Fig. 1b. Expanding either an SFG or IDG results in a data dependency graph (DDG), e.g., Fig. 1d. It is also possible to navigate from a DDG to an IDG, an IDG to an ADG, and so on, by collapsing the current view (or in the case of an SFG, by hiding all structures). We provide transformation operations that allow a user to directly navigate to any of the graphs in Fig. 3 from any other graph. In particular, the operators `ADG`, `IDG`, `SFG`, and `DDG` can be used to go directly to the ADG, IDG, SFG, or DDG view, respectively.

**Expand and Collapse.** In general, the `expand` and `collapse` operators allow users to explore specific portions of a provenance view at different levels of detail. We consider three versions of the `expand` operator based on the type of entity being expanded. For a run $r$, $\texttt{expand}(r) = \{a_1, a_2, \dots\}$ returns the set of actors $a_1, a_2, \dots$ that were invoked as part of the run. Similarly, given an actor $a$, $\texttt{expand}(a) = \{i_1, i_2, \dots\}$ returns the set of invocations $i_1, i_2, \dots$ of $a$. For an invocation $i$, $\texttt{expand}(i) = \{d_1, d_2, \dots\}$ returns the set of fine-grain dependencies $d_1, d_2, \dots$ introduced by $i$, where each $d_j$ is a dependency edge of the form $(x, i, y)$ for data items $x, y$. The `collapse` operator acts as the inverse of `expand`. Given a set of of dependencies $\{d_1, d_2, \dots\}$ generated by an invocation $i$, $\texttt{collapse}(\{d_1, d_2, \dots\}) = i$. Note that a user may select a single dependency to collapse, which will result in all such dependencies of the same invocation to also collapse. Given a set of invocations $\{i_1, i_2, \dots\}$ of an actor $a$, $\texttt{collapse}(\{i_1, i_2, \dots\}) = a$. In a similar way, if a user selects only a single invocation to collapse, this operation will cause all invocations of the corresponding actor to also collapse. Finally, for a set

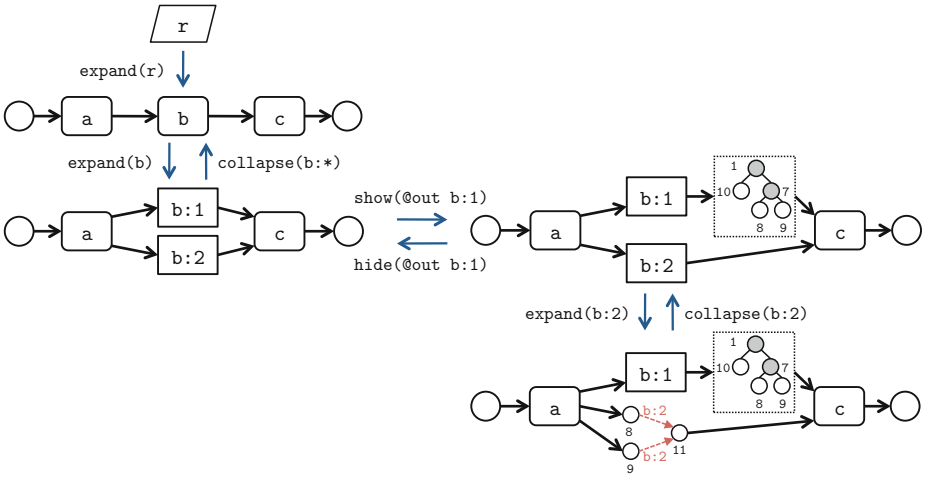


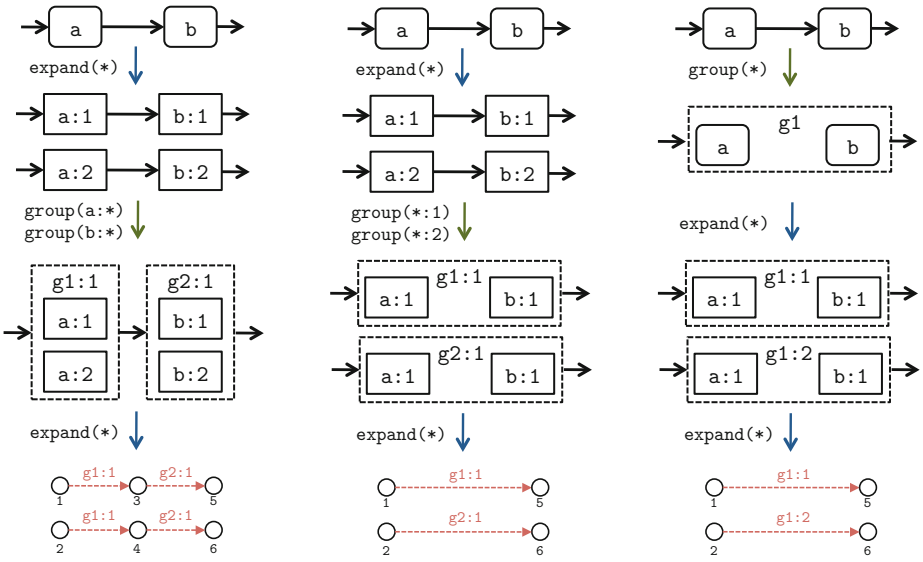

**Fig. 4.** Applying `expand`, `collapse`, `show`, and `hide` operators to only a part of each view

of actors $\{a_1, a_2, \ldots\}$ of run $r$, $\texttt{collapse}(\{a_1, a_2, \ldots\}) = r$. Again, collapsing any one actor will result in setting the current view to the run view.

*Example 1.* The top left of Fig. 4 shows an initial expand step to the corresponding actor dependency graph for the example run of Fig. 1 (labeled `r` in the figure). The second navigation step expands only actor `b` in the ADG. Similarly, the right of the figure shows invocation `b:2` being expanded, resulting in the portion of the data-dependency graph associated with invocation `b:2` (i.e., with dependency edges labeled by `b:2`). The final view shown contains each level of granularity, namely, actors, invocations, data structures, and fine-grain dependencies. Fig. 4 also shows each of the corresponding `collapse` operations. Note that the actor dependency graph is reconstructed by calling `collapse` on the expression `b:`$*$ which denotes all invocations of `b` in the current view.

**Show and Hide.** The `show` operator displays data structures within the current view. The structures displayed depend on the type of entity selected (either a run, actor, or invocation) and whether the input or output of the entity is chosen. We use the QLP "`@`" construct within `show` to select both the entity and whether the input or output is desired. The expression "`@`$*$" denotes all inputs and outputs of each entity in the view. The `hide` operator acts as the inverse of `show` by removing the specified structures. As an example, the `show` operator is used in the third navigation step of Fig. 4 to display the output structure of invocation `b:1` ($\texttt{s}_2^1$ in Fig. 1b). Note that in this example, `show(@in c:1)` would additionally display structure $\texttt{s}_2$ from Fig. 1b.

**Group and Ungroup.** The `group` and `ungroup` operators allow actors and invocations to be combined into composite structures. The `group` operator explicitly allows users to control which items should be grouped and supports both actor and invocation granularity. We consider two versions of `group` and `ungroup`. Given a set of actors $\{a_1, a_2, \ldots\}$, $\texttt{group}(\{a_1, a_2, \ldots\}) = g_{\{a_1, a_2, \ldots\}}$ returns a composite actor $g_{\{a_1, a_2, \ldots\}}$ over the given set.




(a). Invocations grouped by actor (b). Invocations grouped by order (c). Grouping by actor

**Fig. 5.** Actors and invocations combined into composite structures

For a composite actor $g_{\{a_1,a_2,\dots\}}$, $\texttt{ungroup}(g_{\{a_1,a_2,\dots\}}) = \{a_1, a_2, \dots\}$ simply returns the set of actors corresponding to the group (i.e., ungroup is the inverse of group). Similarly, for a set of invocations $\{i_1, i_2, \dots\}$, $\texttt{group}(\{i_1, i_2, \dots\}) = g_{\{i_1,i_2,\dots\}}$ returns a composite invocation $g_{\{i_1,i_2,\dots\}}$, and $\texttt{ungroup}(g_{\{i_1,i_2,\dots\}}) = \{i_1, i_2, \dots\}$ returns the original set.

*Example 2.* Fig. 5 shows three examples of using the `group` operator. Here we consider only a portion of a workflow showing two actors `a` and `b` such that both were invoked twice resulting in the actor dependency graph shown in the second navigation step of Fig. 5a. Each invocation of `a` and `b` consume and produce one data item, where the output of invocation $\texttt{a} : i$ is used by invocation $\texttt{b} : i$. In Fig. 5a, invocations of the same actor are grouped such that `a:1` and `a:2` form one group and `b:1` and `b:2` form a different group. The result of the grouping is shown in the third view of Fig. 5a, and the corresponding data dependency graph is shown as the fourth view. In Fig. 5b, invocations with the same invocation number are grouped such that `a:1` and `b:1` form one group, and `a:2` and `b:2` form a different group. Finally, in Fig. 5c, actors `a` and `b` are first grouped, resulting in a composite actor with two distinct invocations. Unlike in Fig. 5b, these invocations are of the same actor group and have different invocation numbers, whereas in Fig. 5b two distinct groups are created. In general, forming invocation groups explicitly, as opposed to first forming actor groups and then expanding actor groups, supports grouping at a finer-level of granularity by allowing various patterns of composite invocations that are not possible to express at the actor level.

As shown in Fig. 5, composites created by the `group` operator are assigned special identifiers. In addition, the inputs, outputs, and dependencies associated with grouped items

are inferred from the underlying inputs, outputs, and dependencies of the invocations of the groups. For showing dependencies in particular, this often requires computing the transitive dependency closure associated with invocations of the group, e.g., as in Fig. 5b and 5c (since items 3 and 4 are "hidden" by the grouped invocation). When a group is created at the actor level, expanding the group results in a correspondingly grouped set of invocations (as in Fig. 5c). These invocations are constructed based on the invocation dependency graph. In particular, each invocation group of the actor group contains a set of connected invocations, and no invocation within an invocation group is connected to any other invocation in a different invocation group. Thus, the portion of the invocation graph associated with the actor group is partitioned into connected subgraphs, and each such subgraph forms a distinct invocation group of the actor group. When an invocation group is expanded, this composite invocation is used in structure flow and data dependency graphs, resulting in provenance views where dependencies are established between input and output items, without intermediate data in between. This approach allows scientists to continue to explore dependencies for grouped invocations (since fine-grain dependencies are *maintained* through groups, unlike, e.g., the approach in [11]).

**Filter.** The `filter` operation allows provenance views to be refined using QLP query expressions. Issuing a query using `filter` results in only the portion of the current provenance view corresponding to the query answer to be displayed. Given a QLP query $q$, $\texttt{filter}(q) = \{d_1, d_2, \ldots\}$ returns the set of fine-grain dependencies that result from applying the query to the trace graph. These dependencies are used to remove or add entities to the current view. In general, items can be added to a view when the current view is based on a more selective query.

**Aggregation.** It is often convenient to see summary information about entities (actors, invocations, etc.) when exploring provenance information. We provide standard aggregate operators (`count`, `min`, `max`, `avg`) to obtain statistics for a user's current provenance view. The "`count` *entity_type* `of` *scope*" operator returns the number of entities of *entity_type* within a given *scope* expression. Entity types are either `actors`, `invocations`, or `data`, which count the number of actors, invocations, or data items, respectively. For the `actors` entity type, the scope is either $*$, denoting the entire view, or a group identifier. For example, "`count actors of` $*$" returns two for each view in Fig. 5, whereas "`count actors of g1:1` returns one for the third view in Fig. 5. For the `invocations` entity type, the scope is either $*$ (the entire view) or an actor or group identifier. For instance, in the first view of Fig. 5a "`count invocations of` $*$" returns four whereas "`count invocations of a`" returns two. The `count` operation is also useful for exploring workflow loops, e.g., the expression "`count invocations of a`" can be used to obtain the number of iterations of `a` in Fig. 2. For the `data` entity type, the QLP "`@`" syntax is used to define the scope. For instance, for the invocation dependency graph of Fig. 1e, "`count data of @in`" returns two (since two data items are input to the workflow), and "`count data of @out a:1`" returns four (the number of data items output by invocation `a:1`). The `min`, `max`, and `avg` operations return the minimum, maximum, and average number of entity types within a view, respectively. These operations can be used to compute the number of actors by group

(e.g., "`min actor by group`"), the number of invocations by actors or groups (e.g., "`min invocations by actor`"), and the number of input or output data items by actor, group, or invocation (e.g., "`min input data by actor`"). Both the `min` and `max` operators return the entity with the minimum or maxiximum count value, respectively, as well as the number of corresponding entities. Finally, the `params` operation returns the set of parameter values used in invocations of the current view. This operation can also be restricted to specific actors, invocations, or groups, e.g., `params(a:1)` returns the parameter settings for invocation `a:1` whereas `params`$(*)$ returns the parameter settings for all invocations in the current view.

## 4 Implementation

In this section we describe an implementation using standard relational database technology for the operators presented in Section 3. As mentioned in the introduction, this approach has benefits for efficiency and scalability (see Section 5). We first describe a set of relational schemas for representing traces and default views, and then show how the navigation operators can be implemented as relational queries over these schemas.

We consider three distinct schemas: (1) a *trace schema* **T** for representing the trace information corresponding to a workflow run; (2) a *dependency schema* **D** for representing the result of executing QLP queries expressed through `filter` operations; and (3) a *view schema* **V** for representing the user's current provenance view. The trace schema **T** consists of the following relations: $\texttt{Run}(r,w)$ denotes that $r$ was a run of workflow $w$; $\texttt{Invoc}(r,i,a,j)$ denotes that invocation $i$ was the $j$-th invocation of actor $a$ in run $r$; $\texttt{Node}(r,n,p,t,l,i_{ins},i_{del})$ denotes that $n$ was an item (node) in run $r$ such that $p$ is the parent collection of $n$, $t$ is the type of the item (either `data` or `collection`), $l$ is the label of $n$ (e.g., XML tag name), and that $n$ was inserted by invocation $i_{ins}$ and deleted by $i_{del}$; $\texttt{DDep}(r,n,n_{dep})$ denotes a dependency from item $n_{dep}$ to item $n$ in run $r$; $\texttt{DDepc}(r,n,n_{dep})$ stores the transitive closure of `DDep`; $\texttt{IDep}(r,i,i_{dep})$ denotes an invocation dependency from invocation $i$ to $i_{dep}$; and $\texttt{IDepc}(r,i,i_{dep})$ stores the transitive closure of `IDepc`. In general, storing the transitive closure improves query time for more complex path-expression queries and simplifies a number of the operations presented here. We show in [5] an approach for efficiently compressing the transitive closure relations that only marginally affect the response time of basic queries (adding only an additional join in many cases), and adopt this approach in Section 5 when describing our experimental results. In [4] we extend this approach for efficiently answering QLP queries, which we assume here for implementing `filter` operations. We also assume the following views (expressed using Datalog) as part of $T$.

$$\texttt{ADep}(r,a,a_{dep}) \texttt{ :- } \texttt{IDep}(r,i,i_{dep}), \texttt{Invoc}(r,i,a,j_1), \texttt{Invoc}(r,i_{dep},a_{dep},j_2).$$
$$\texttt{ADepc}(r,a,a_{dep}) \texttt{ :- } \texttt{IDepc}(r,i,i_{dep}), \texttt{Invoc}(r,i,a,j_1), \texttt{Invoc}(r,i_{dep},a_{dep},j_2).$$

The dependency graph schema **D** consists of the single relation $\texttt{DepView}(n_{from},i,n_{to})$ denoting a dependency edge from item $n_{from}$ to $n_{to}$ labelled by invocation $i$. Similarly, the view schema **V** consists of the single relation $\texttt{CurrView}(e_{from},t_{from},l,e_{to},t_{to})$ for $e_{from}$ and $e_{to}$ entites connected via an edge label $l$ such that each entity's type

is denoted by $t_{to}$ and $t_{from}$, respectively. The entity type can be either a `run`, `actor`, `invoc` (invocation), `data`, `coll` (collection), or `struct` (structure). A structure is denoted by a QLP expression of the form `@in` $[i \mid a]$ or `@out` $[i \mid a]$ where invocation $i$ and actor $a$ are optional. The following are examples of possible tuples stored within the current view relation: $\texttt{CurrView}(r, \texttt{run}, \perp, \perp, \perp)$ stores a run view ($\perp$ denotes a null value); $\texttt{CurrView}(a_1, \texttt{actor}, \perp, a_2, \texttt{actor})$ stores an edge in an actor dependency graph; $\texttt{CurrView}(i_1, \texttt{invoc}, \perp, i_2, \texttt{invoc})$ stores an edge in an invocation dependency graph; $\texttt{CurrView}(\texttt{@out}\ i_1, \texttt{struct}, \perp, i_2, \texttt{invoc})$ stores an edge in a structure flow graph; and $\texttt{CurrView}(n_1, \texttt{data}, \texttt{a:1}, n_2, \texttt{data})$ stores an edge in a data dependency graph.

When a user begins navigating a trace, the instance $D$ of the dependency schema $\mathbf{D}$ consists of the entire set of dependencies. After applying a navigation operation, the instance $D$ of the dependency schema $\mathbf{D}$ and $V$ of the current view schema $\mathbf{V}$ are updated as needed. For example, after applying an initial `filter` operation, the dependencies of $D$ are updated (denoted as $D_1$) to store the result of the given QLP query over the instances $T$ of the trace schema $\mathbf{T}$. The instance $V$ of the view schema $\mathbf{V}$ is also updated (denoted as $V_1$) based on $D_1$. Similarly, after applying an `expand`, `collapse`, `show`, or `hide` operator, a new view $V_2$ is created from $D_1$, $V_1$, and $T$. This process continues for each navigation step performed by the user, where only the current view and dependency graph is stored together with the initial trace.

**Queries to Generate Default Views.** We can implement the default view operators as queries over a trace instance $T$ and dependency instance $D$ as follows. First, we define the following notation as shorthand for filtering relations in $T$ by $D$. Given a trace relation $R$, we write $R^{(D)}$ to denote the filtered version of $R$ with respect to the dependencies in $D$. For instance, $\texttt{IDep}^{(D)}$ is the invocation dependency relation (`IDep`) containing only invocations that participate in dependency edges within $D$. Given a dependency relation $D$, we define the following:

$$\texttt{IDep}^{(D)}(r,i,i_{dep}) \texttt{:-} \texttt{IDep}(r,i,i_{dep}), \texttt{DepView}(n_1,i,n_2), \texttt{DepView}(n_2,i_{dep},n_3).$$
$$\texttt{ADep}^{(D)}(r,a,a_{dep}) \texttt{:-} \texttt{IDep}^{(D)}(r,i,i_{dep}), \texttt{Invoc}(r,i,a,j_1), \texttt{Invoc}(r,i_{dep},a_{dep},j_2).$$
$$\texttt{Node}^{(D)}(r,n,p,t,l,i_{ins},i_{del}) \texttt{:-} \texttt{Node}(r,n,p,t,l,i_{ins},i_{del}), \texttt{DepView}(n,i,n_2).$$
$$\texttt{Node}^{(D)}(r,n,p,t,l,i_{ins},i_{del}) \texttt{:-} \texttt{Node}(r,n,p,t,l,i_{ins},i_{del}), \texttt{DepView}(n_1,i,n).$$

These relations are used to compute the default view operators for a run `r` (each of which follow the same relation structure as `CurrView`):

$$\texttt{ADG}(a,\texttt{actor},\perp,a_{dep},\texttt{actor}) \texttt{ :- } \texttt{ADep}^{(D)}(\texttt{r},a,a_{dep}).$$
$$\texttt{IDG}(i,\texttt{invoc},\perp,i_{dep},\texttt{invoc}) \texttt{ :- } \texttt{IDep}^{(D)}(\texttt{r},i,i_{dep}).$$
$$\texttt{DDG}(n_1,t_1,i,n_2,t_2) \texttt{ :- } \texttt{DepView}(n_1,i,n_2), \texttt{Node}(\texttt{r},n_1,p_1,t_1,i_{ins_1},i_{del_1}), \texttt{Node}(\texttt{r},n_2,p_2,t_2,i_{ins_2},i_{del_2}).$$

Each of these operators returns a new view that replaces the current view. In a similar way, we can define the structure flow graph for run `r`, which is similar to computing the invocation dependency graph.

$$\texttt{SFG}(i,\texttt{invoc},\perp,\texttt{@out}\ i,\texttt{struct}) \texttt{ :- } \texttt{IDep}^{(D)}(\texttt{r},i,i_{dep}).$$
$$\texttt{SFG}(\texttt{@out}\ i,\texttt{struct},\perp,\texttt{@in}\ i_{dep},\texttt{struct}) \texttt{ :- } \texttt{IDep}^{(D)}\texttt{r},i,i_{dep}).$$
$$\texttt{SFG}(\texttt{@in}\ i_{dep},\texttt{struct},\perp,i_{dep},\texttt{invoc}) \texttt{ :- } \texttt{IDep}^{(D)}(r,i,i_{dep}).$$

**Queries to Generate Input-Output Structures.** Given a trace $T$ and a dependency $D$, we compute the input and output structures of runs $r$, actors $a$, and invocations $i$ as follows. The input of a run r includes all data and collection items in the trace that were not inserted by any invocation:

$$\texttt{RunInput}(n,p) \texttt{ :- } \texttt{Node}^{(D)}(\texttt{r},n,p,t,l,\perp,i_{del}).$$

The output items of $r$ include those that (1) were either input to the run or inserted by an invocation, and (2) were not deleted by an invocation:

$$\texttt{RunOutput}(n,p) \texttt{ :- } \texttt{Node}^{(D)}(\texttt{r},n,p,t,l,i_{ins},\perp).$$

The input of an invocation $i$ includes all nodes not deleted by an invocation that $i$ depends on such that either the item (1) was inserted by an invocation that $i$ depended on or (2) was not inserted by an invocation and thus was an input to the run. The output of $i$ is computed similarly, i.e., by removing from the input of $i$ the nodes deleted by $i$ and adding the nodes inserted by $i$. The following rules compute the input and output of a given invocation i and a run r.

$$\begin{aligned}
&\texttt{InvocInput}(n,p)\texttt{:-}\texttt{Node}^{(D)}(\texttt{r},n,p,t,l,\perp,\perp).\\
&\texttt{InvocInput}(n,p)\texttt{:-}\texttt{Node}^{(D)}(\texttt{r},n,p,t,l,i_{ins},\texttt{i}).\\
&\texttt{InvocInput}(n,p)\texttt{:-}\texttt{Node}^{(D)}(\texttt{r},n,p,t,l,\perp,i_{del}),\texttt{Invoc}(\texttt{r},i_{del},a,j),\neg\texttt{IDepc}(\texttt{r},i_{del},\texttt{i}).\\
&\texttt{InvocInput}(n,p)\texttt{:-}\texttt{Node}^{(D)}(\texttt{r},n,p,t,l,i_{ins},i_{del}),\texttt{IDepc}(\texttt{r},i_{ins},\texttt{i}),\texttt{Invoc}(\texttt{r},i_{del},a,j),\\
&\qquad\qquad\neg\texttt{IDepc}(\texttt{r},i_{del},\texttt{i}).
\end{aligned}$$

Note that the first rule selects input items that were neither inserted or deleted within a run, and the second rule selects input items that were deleted by the given invocation i. The last two rules ensure the item was not deleted by an invocation that i depended on. The output of invocations are defined similarly:

$$\begin{aligned}
&\texttt{InvocOutput}(n,p) \texttt{ :- } \texttt{Node}^{(D)}(\texttt{r},n,p,t,l,\perp,\perp).\\
&\texttt{InvocOutput}(n,p) \texttt{ :- } \texttt{Node}^{(D)}(\texttt{r},n,p,t,l,\texttt{i},i_{del}).\\
&\texttt{InvocOutput}(n,p) \texttt{ :- } \texttt{Node}^{(D)}(\texttt{r},n,p,t,l,\perp,i_{del}),\texttt{Invoc}(\texttt{r},i_{del},a,j),\\
&\qquad\qquad\neg\texttt{IDepc}(\texttt{r},i_{del},\texttt{i}),i_{del}\neq\texttt{i}.\\
&\texttt{InvocOutput}(n,p) \texttt{ :- } \texttt{Node}^{(D)}(\texttt{r},n,p,t,l,i_{ins},i_{del}),\texttt{IDepc}(\texttt{r},i_{ins},\texttt{i}),\\
&\qquad\qquad\texttt{Invoc}(\texttt{r},i_{del},a,j),\neg\texttt{IDepc}(\texttt{r},i_{del},\texttt{i}),i_{del}\neq\texttt{i}.
\end{aligned}$$

The input and output structures of actors (as opposed to invocations) are computed by first retrieving the invocations that are in the current view, and then for each such invocation, unioning the corresponding structures.

**Queries to Group and Ungroup Actors and Invocations.** To implement the group and ungroup operators, we store the set of entities supplied to the group operator in a temporary relation $\texttt{Group}(e,t,n,g)$ where $e$ is one of the entities being grouped, $t$ is the entity type (either actor or invoc), $n$ is the group identifier, and $g$ is the grouping type. For example, to group invocations b : 1 and b : 2 into a group g1 : 1 we store the tuples $\texttt{Group}(\texttt{b}:1,\texttt{invoc},\texttt{g1}:1,\texttt{invoc\_group})$ and $\texttt{Group}(\texttt{b}:2,\texttt{invoc},\texttt{g1}:1,$

invoc_group). The new view with grouped entities is generated as follows: (1) retrieve those tuples from the current view relation $\texttt{CurrView}_i$ where $e_{from}$ and $e_{to}$ are not entities to be grouped and store these in the new view $\texttt{CurrView}_{i+1}$; (2) if $e_{from}$ is an entity to be grouped, then retrieve the tuple from $\texttt{CurrView}_i$ and modify $e_{from}$ and $t_{from}$ with the group identifier $n$ and group type $g$, respectively; and similarly (3) if $e_{to}$ is an entity to be grouped, perform a similar operation as in (2). These steps are performed by the following queries. Note that we assume for each entity $e$ of type $t$ that is not involved in a group within the current view, there exists a tuple $\texttt{Group}(e,t,\perp,\perp)$.

$$\begin{aligned}
\texttt{CurrView}_{i+1}(e_{from},t_{from},l,e_{to},t_{to}) \texttt{ :- } & \texttt{CurrView}_i(e_{from},t_{from},l,e_{to},t_{to}),\\
& \texttt{Group}(e_{from},t_{from},\perp,\perp),\\
& \texttt{Group}(e_{to},t_{to},\perp,\perp),\\
\texttt{CurrView}_{i+1}(n,g,l,e_{to},t_{to}) \texttt{ :- } & \texttt{CurrView}_i(e_{from},t_{from},l,e_{to},t_{to}),\\
& \texttt{Group}(e_{from},t_{from},n,g),\\
& \texttt{Group}(e_{to},t_{to},\perp,\perp).\\
\texttt{CurrView}_{i+1}(e_{from},t_{from},l,n,g) \texttt{ :- } & \texttt{CurrView}_i(e_{from},t_{from},l,e_{to},t_{to}),\\
& \texttt{Group}(t_{to},e_{to},n,g),\\
& \texttt{Group}(e_{from},t_{from},\perp,\perp).\\
\texttt{CurrView}_{i+1}(n_1,g_1,l,n_2,g_2) \texttt{ :- } & \texttt{CurrView}_i(e_{from},t_{from},l,e_{to},t_{to}),\\
& \texttt{Group}(e_{from},t_{from},n_1,g_1),\\
& \texttt{Group}(t_{to},e_{to},n_2,g_2).
\end{aligned}$$

Grouping has implications on how inputs, outputs, and data dependencies across grouped entities are displayed. Inputs of grouped entities are computed by performing the union of all the inputs of those entities that are a source node in the graph with respect to entities to be grouped. Similarly, the outputs of grouped entities are computed by performing the union of all the outputs of those entities that are sink nodes with respect to the entities to be grouped. Also, as discussed in the previous section, data dependency views over grouped entities require computing the transitive dependency closure of inputs and outputs of grouped entities. We denote the source entities of a group as $\texttt{Group}_{src}$ and the sink entities of a group as $\texttt{Group}_{sink}$. These relations are computed by first deriving the dependency relations between the entities of the group and then checking which one has no incoming edges (for $\texttt{Group}_{src}$), and similarly, which one has no outgoing edges (for $\texttt{Group}_{sink}$).

The ungroup operator for the current view $\texttt{CurrView}_i$ is performed as follows: (1) for all invocations of the group, we retrieve their invocation dependencies; (2) for all source invocations of the group (with respect to the invocation dependencies), we add an edge to $\texttt{CurrView}_{i+1}$ from the $e_{to}$ to the source invocation; and (3) for all sink invocations of the group, we add an edge to $\texttt{CurrView}_{i+1}$ from the sink invocation to $e_{from}$. Ungrouping of an actor is done in a similar way.

$$\begin{aligned}
\texttt{CurrView}_{i+1}(i,\texttt{invoc},l,i_{dep},\texttt{invoc}) \texttt{ :- } & \texttt{IDep}^{(D)}(\texttt{r},i,i_{dep}),\\
& \texttt{Group}(i,\texttt{invoc},n,\texttt{invoc_group}),\\
& \texttt{Group}(i_{dep},\texttt{invoc},n,\texttt{invoc_group}).\\
\texttt{CurrView}_{i+1}(e_{from},t_{from},l,e,t) \texttt{ :- } & \texttt{CurrView}_i(e_{from},t_{from},l,n,g),\\
& \texttt{Group}_{src}(n,g,e,t).\\
\texttt{CurrView}_{i+1}(e,t,l,e_{to},t_{to}) \texttt{ :- } & \texttt{CurrView}_i(n,g,l,e_{to},t_{to}),\texttt{Group}_{sink}(n,g,e,t).
\end{aligned}$$

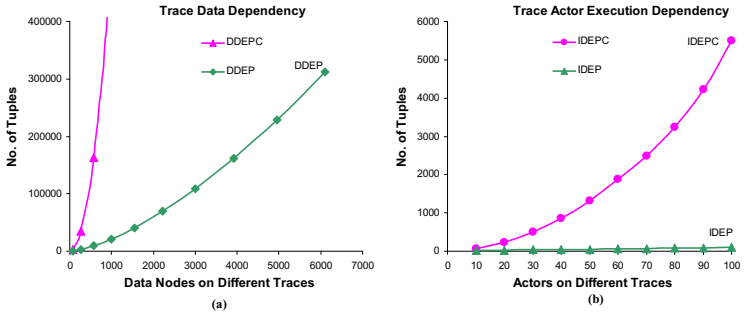



**Fig. 6.** (a) Data dependency and (b) actor invocation complexity of synthetic traces

**Expand, Collapse, and Aggregates.** We `expand` a given invocation `i` to show the dependency relationship between its inputs and outputs using the relation $\texttt{DepView}(n_{from}, i, n_{to})$ in $D$. We `collapse` an invocation $i$ by showing the actor corresponding to the invocation via the `Invoc` relation in $T$. We `expand` a given actor $a$ by obtaining the invocations of $a$ that are in $D$. The current view is constructed directly from these operations in a similar way as for grouping and ungrouping (i.e., by inserting the corresponding relations in the new current view $\texttt{CurrView}_{i+1}$). Finally, the aggregate operations are straightforward to compute using standard relational aggregation queries over the current view, dependency, and group relations.

## 5 Experimental Results

Here we evaluate the feasibility, scalability, and efficiency of executing the navigation operators over the approach in Section 4 on both real and synthetic traces. Real traces were generated from existing workflows implemented within the Kepler scientific workflow system. Our experiments were performed using a 2.8GHz Intel Core 2 duo PC with 4 GB RAM and 500 GB of disk space. Navigational operators were implemented as SQL queries (views over the schema), which were executed against a PostgreSQL database where all provenance traces were stored. The QLP parser was implemented in Java using JDBC to communicate with the provenance database.

We evaluated the feasibility of executing the navigation operator queries using the following real traces from scientific workflows implemented within Kepler: the *GBL* workflow [7] infers phylogenetic trees from protein and morphological sequence data; the *PC1* workflow was used in the first provenance challenge [19]; the *STAP* and *CYC* workflows are used in characterizing microbial communities by clustering and identifying DNA sequences of 16S ribosomal RNA; the *WAT* workflow characterizes microbial populations by producing phylogenetic trees from a list of sequence libraries; and the *PC3* workflow was used within the third provenance challenge[2]. These traces ranged from 100–10,000 immediate data dependencies and 200–20,000 transitive data dependencies. We also evaluated our approaches to determine the scalability of executing navigation operators queries using synthetic traces ranging from 500–100,000 immediate data dependencies, 1,000–$10^8$ transitive data dependencies, and data dependency

[2] see `http://twiki.ipaw.info/bin/view/Challenge/`

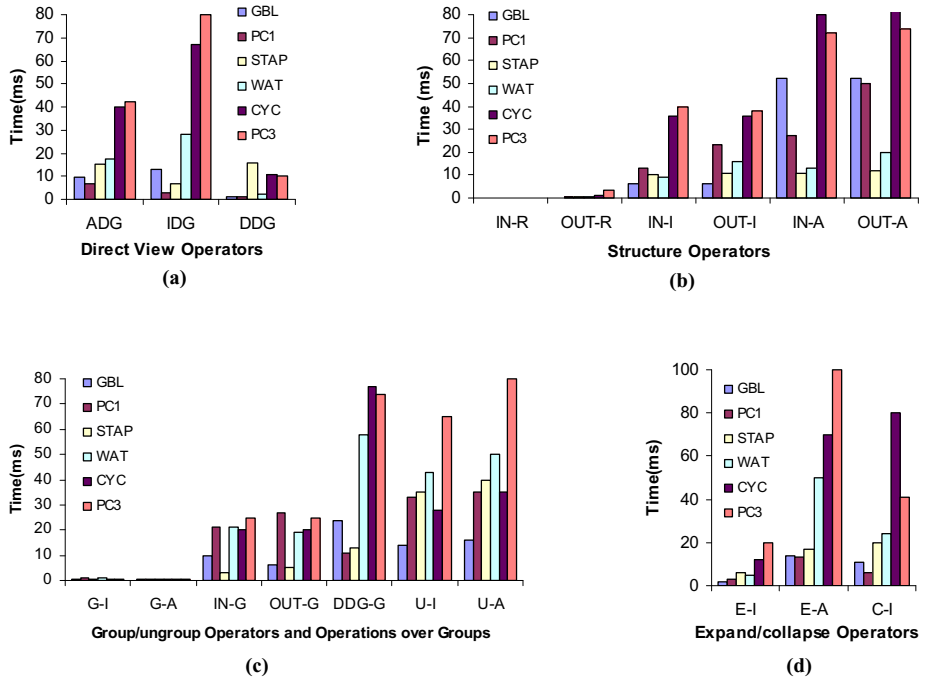



**Fig. 7.** Average query time for operators over real traces: (a) $Q_D$; (b) $Q_S$; (c) $Q_G$; and (d) $Q_E$

paths of length 10–100. These traces also contained 10–100 actors, 10–100 immediate invocation dependencies, and 10–6,000 transitive invocation dependencies. The synthetic traces were taken from [5], and represent typical dependency patterns of common scientific workflows [5,7,19]. Fig. 6a shows the complexity of data dependencies (immediate and transitive) as the number of nodes in the synthetic traces increase, and Fig. 6b shows the the complexity of invocation dependencies (immediate and transitive) as the number of actors in the synthetic traces increase.

We use four types of operators in our evaluation: ($Q_D$) queries to generate default ADG, IDG, and DDG views; ($Q_S$) queries to generate data structures (run, actor, and invocation inputs and outputs); ($Q_G$) queries to group and ungroup actors and invocations, and to retrieve their inputs, outputs, and fine-grained dependencies; and ($Q_E$) queries to expand and collapse actors and invocations. Section 4 details the underlying datalog queries for each operator type.

**Feasibility and Efficiency Results.** Fig. 7 shows timing results of the navigation operators over real traces. The time to execute DDG operations (Fig. 7a) is smaller since we store the result of previous QLP queries in the dependency instance $D$, which is used to generate the result of DDG calls, whereas ADG and IDG operations need additional queries over provenance schema tables. The time to retrieve the input and output structures of a run is less expensive than for an invocation (Fig. 7b). However, the time to retrieve input and output structures for an invocation is less expensive for an actor. This is because an actor can be invoked many times, which requires computing the union of structures for all such invocations. Grouping invocations and actors is also less

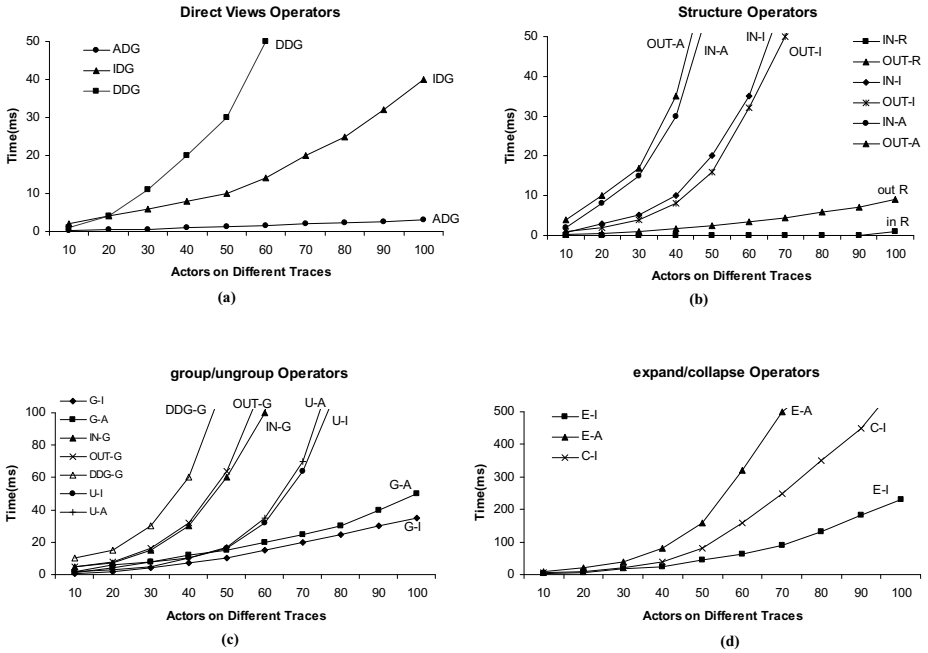



**Fig. 8.** Average query time for operators over synthetic traces: (a) $Q_D$; (b) $Q_S$; (c) $Q_G$; and (d) $Q_E$

expensive than their inverse ungrouping operations (see Fig. 7c), since ungrouping has to perform many conditional joins with the current view to reconstruct the ungrouped invocations, actors, and their relations to other items within the current view. Expanding an actor is more expensive than expanding an invocation (see Fig. 7d) since expanding an invocation involves dependencies that are already materialized as QLP query results, but expansion of an actor must execute queries against schema tables through additional conditional joins. Despite the complexity involved in executing such queries, our experimental results show that each type of operation takes *less then* 1 sec, demonstrating the feasibility and efficiency that can be obtained using a purely relational approach.

**Scalability Results.** Fig. 8 shows the results of executing navigation queries over the synthetic traces. As shown, most of the queries are still executed in less than 1 sec (100 ms) for larger trace sizes, suggesting that a purely relational approach can scale to larger trace sizes (compared with those obtained from the real traces used above). Note that queries for implementing the `expand` and `collapse` operators (i.e., of type $Q_E$) take more time than the other operator types, which is due to the number of conditional joins (as discussed earlier) that are required. Overall, however, the results for synthetic traces confirm those discussed in the case for real traces above.

## 6 Related Work

Current approaches for exploring workflow provenance are based on statically visualizing entire provenance graphs [16,14,13,7]. In these approaches, provenance graphs

are typically displayed at the lowest level of granularity (e.g., fine-grain dependencies). In the case of [14], query results are viewed independently of the rest of a provenance trace. Some systems divide provenance information into distinct layers, e.g., VisTrails divides provenance information into workflow evolution, workflow, and execution layers [8], and PASS divides provenance into data and process layers [21]. In all these approaches, however, these levels are largely either orthogonal or hierarchical, whereas the provenance views supported by our model (i) combine both hierarchical abstractions (i.e., ADGs, IDGs, and SFGs) with (ii) the ability to seamlessly navigate between these different levels of granularity, while (iii) allowing users to summarize, group, and filter portions of these views to create new views for further exploration of relevant provenance information. The Zoom*UserViews system [6] (extended in [2] to work with a fine-grained database provenance model for Pig Latin) provides a mechanism for defining composite actors to abstract away non-relevant provenance information. Composites are constructed over "relevant" actors to maintain certain dataflow connections, thereby generating a view over the composites that is similar to the original. However, unlike in our approach, users of the Zoom*UserViews system cannot explicitly define their own composites, and composition is defined only at the actor level (where each actor is assumed to have at most one invocation). Our approach also maintains grouping across views, maintains the original data dependencies for composites (unlike in the Zoom*UserViews approach, which switches to coarse-grain dependencies), and we support a more general provenance model that explicitly handles structured data.

Our navigation approach is inspired by and has similarities to those proposed previously for exploring object-oriented [9] and XML databases, where graphical environments allow users to "drill-down" from schema to instances and navigate relationships among data. For example, [9] provides an integrated browsing and querying environment that allows users to employ a "query-in-place" paradigm where navigation and query can be mixed. In contrast, provenance information is largely schema-free, i.e., the information contained within an ADG, IDG, and SFG is not constrained by an explicit schema, and queries in our model are posed directly against the items contained within these views (or generally, the fine-grain dependency graph).

## 7 Conclusion

We have presented a general model to help users explore provenance information through novel graph-based summarization, navigation, and query operators. The work presented here extends our prior work [7,3] by providing a significantly richer model for navigation (adding additional views, the `show` and `hide` constructs, and aggregation) as well as an implementation using standard relational database technology. We also provide experimental results demonstrating the feasibility, scalability, and efficiency of our approach. Because of the size and complexity of real-world scientific workflow provenance traces [10,4], providing users with high-level navigation operations and views for abstracting and summarizing provenance information can provide a powerful environment for scientists to explore and validate the results of scientific workflows.

**Acknowledgements.** Work supported through NSF grants IIS-1118088, DBI-0743429, DBI-0753144, DBI-0960535, and OCI-0722079.

## References


1. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. IJDL (1997)
2. Amsterdamer, Y., Davidson, S.B., Deutch, D., Milo, T., Stoyanovich, J., Tannen, V.: Putting lipstick on pig: Enabling database-style workflow provenance. PVLDB 5(4) (2011)
3. Anand, M.K., Bowers, S., Ludäscher, B.: A navigation model for exploring scientific workflow provenance graphs. In: Proc. of the Workshop on Workflows in Support of Large-Scale Science, WORKS (2009)
4. Anand, M.K., Bowers, S., Ludäscher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: EDBT, pp. 287–298 (2010)
5. Anand, M.K., Bowers, S., McPhillips, T.M., Ludäscher, B.: Efficient provenance storage over nested data collections. In: EDBT (2009)
6. Biton, O., Boulakia, S.C., Davidson, S.B., Hara, C.S.: Querying and managing provenance through user views in scientific workflows. In: ICDE (2008)
7. Bowers, S., McPhillips, T., Riddle, S., Anand, M.K., Ludäscher, B.: Kepler/pPOD: Scientific Workflow and Provenance Support for Assembling the Tree of Life. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 70–77. Springer, Heidelberg (2008)
8. Callahan, S., Freire, J., Santos, E., Scheidegger, C., Silva, C., Vo, H.: VisTrails: Visualization meets data management. In: SIGMOD (2006)
9. Carey, M.J., Haas, L.M., Maganty, V., Williams, J.H.: PESTO: An integrated query/browser for object databases. In: VLDB (1996)
10. Chapman, A., Jagadish, H.V., Ramanan, P.: Efficient provenance storage. In: SIGMOD (2008)
11. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: SIGMOD (2008)
12. He, H., Singh, A.K.: Graphs-at-a-time: Query language and access methods for graph databases. In: SIGMOD, pp. 405–418 (2008)
13. Hunter, J., Cheung, K.: Provenance explorer-a graphical interface for constructing scientific publication packages from provenance trails. Int. J. Digit. Libr. 7(1) (2007)
14. Lim, C., Lu, S., Chebotko, A., Fotouhi, F.: Opql: A first opm-level query language for scientific workflow provenance. In: IEEE SCC, pp. 136–143 (2011)
15. Ludäscher, B., et al.: Scientific workflow management and the Kepler system. Concurr. Comput.: Pract. Exper. 18(10) (2006)
16. Macko, P., Seltzer, M.: Provenance map orbiter: Interactive exploration of large provenance graphs. In: TAPP (2011)
17. Missier, P., Paton, N.W., Belhajjame, K.: Fine-grained and efficient lineage querying of collection-based workflow provenance. In: EDBT, pp. 299–310 (2010)
18. Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., Goble, C.: Taverna, Reloaded. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 471–481. Springer, Heidelberg (2010)
19. Moreau, L., et al.: The first provenance challenge. Concurr. Comput.: Pract. Exper. 20(5) (2008)
20. Moreau, L., et al.: The open provenance model core specification (v1.1). Future Generation Computer Systems 27(6), 743–756 (2011)
21. Muniswamy-Reddy, K.K., et al.: Layering in provenance systems. In: USENIX Annual Technical Conference (2009)