Reconciling Provenance Policy Conflicts by Inventing Anonymous Nodes

Saumen Dey[1], Daniel Zinn[2], and Bertram Ludäscher[1,2]

[1] Dept. of Computer Science, University of California, Davis
[2] Genome Center, University of California, Davis

Abstract. In scientific collaborations, provenance is increasingly used to understand, debug, and explain the processing history of data, and to determine the validity and quality of data products. While provenance is easily recorded by scientific workflow systems, it can be infeasible or undesirable to publish provenance details for all data products of a workflow run. We have developed PROPUB, a system that allows users to publish a *customized* version of their data provenance, based on a set of publication and customization requests, while observing certain provenance publication policies, expressed as logic integrity constraints. When user requests conflict with provenance policies, repair actions become necessary. In prior work, we removed additional parts of the provenance graph (i.e., not directly requested by the user) to repair constraint violations. In this paper, we present an alternative approach, which ensures that all relevant nodes are retained in the provenance graph. The key idea is to introduce new anonymous nodes to represent lineage dependencies, without revealing information that the user wants to protect. With this new approach, a user may now explore different provenance publication strategies, and choose the most appropriate one before publishing sensitive provenance data.

1 Introduction

In the emerging paradigm of collaborative, data-intensive science, sharing data products even prior to publication may be desirable [1,2]. Yet, without a proper scientific publication associated with shared data, its validity and accuracy is difficult to assess. This is problematic in collaborative environments, where data shared by one scientist is used by another scientist as input for further studies. In such settings, *data provenance* (the lineage and processing history of data) can help to ensure data quality [3,4,5,6,7]. It is thus desirable to publish data products together with their provenance. In many cases, however, provenance data can be sensitive and may contain private information or intellectual property that should not be revealed [7,8,5]. Consequently, one has to balance between (i) the desire to publish provenance data so that collaborators can understand and rely on the shared data products, and (ii) the need to protect sensitive information, e.g., due to privacy concerns or intellectual property issues (Figure 1).

We view provenance as a bipartite, directed, acyclic graph, capturing which *data nodes* were consumed and produced, respectively, by *invocation nodes* (computations). Our model thus corresponds to the Open Provenance Model (OPM) which captures the dependencies between data artifacts and process invocations [9].

R. García-Castro et al. (Eds.): ESWC 2011 Workshops, LNCS 7117, pp. 172–185, 2012.

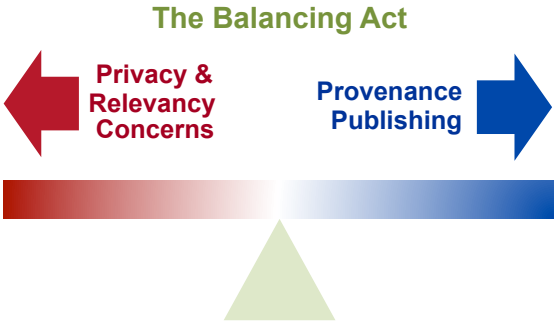


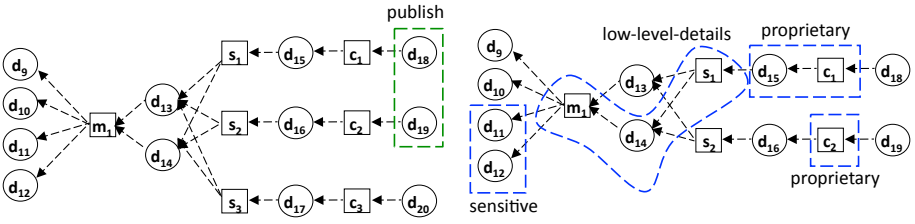
Fig. 1. In collaborative settings, scientists publish provenance for an improved understanding of the result data. With increasing privacy concerns, collaborators have to choose the right balance between providing sufficient provenance data and protecting sensitive information.

To sanitize provenance graphs, a scientist can remove sensitive data nodes or invocations nodes from the provenance graph. Alternatively, she can *abstract* a set of sensitive nodes by grouping them into a single, abstract node. These updates may violate some of the integrity constraints of the provenance graph [10]. For example, grouping multiple nodes into one abstraction node may introduce new dependencies, which were absent in the initial provenance graph. Hiding nodes may also make some nodes in the final graph appear independent of each other even though they are dependent in the original graph. Thus, one can no longer trust that the published provenance data is "correct" (e.g., there are no false dependencies) or "complete" (e.g., there are no false independencies). Therefore, we propose a system that allows a publisher to provide a high-level specification what parts of the provenance graph are to be published and what parts are to be sanitized, *while guaranteeing* that at the same time certain provenance publication constraints are observed.

2 Motivating Example

Figure 2(a) shows a simplified version of the provenance graph (PG) from the First Provenance Challenge [11]. Scientific workflow systems often automatically record such provenance [12,13], and the provenance graphs may resemble the workflow graph, i.e., the former can be seen as instances of the latter [2]. At the workflow specification level, *actors* are used to represent the computational steps, implemented by software components, while at the provenance (or instance) level, we have *invocations* of those actors. We depict *data nodes* as circles and *invocation nodes* as boxes. Dependencies among them are shown as directed edges. These edges capture the lineage of data nodes and thus are typically drawn from right (newer nodes) to left (older nodes), i.e., in the opposite direction of the dataflow edges in a workflow specification. For example, $\mathtt{d}_{16}$ *was generated by* an invocation $\mathtt{s}_2$, and was in turn *used* by invocation $\mathtt{c}_2$, denoted by, respectively $\mathtt{s}_2 \xleftarrow{gen\_by} \mathtt{d}_{16}$ and $\mathtt{d}_{16} \xleftarrow{used} \mathtt{c}_2$.

Assume the user wants to publish data products $\mathtt{d}_{18}$ and $\mathtt{d}_{19}$ along with their provenance information, i.e., the data lineage of these nodes. This publication request is



(a) Provenance graph (PG) and publication user requests

(b) User-selected $PG' \subseteq PG$ with various privacy issues

Fig. 2. (a) Publication requests to publish the lineage of $\{d_{18}, d_{19}\}$; and (b) privacy issues: (i) data nodes $\{d_{11}, d_{12}\}$ are sensitive, (ii) nodes $\{m_1, d_{14}, s_1\}$ are low level details (i.e. not very useful) for the intended user, and (iii) nodes $\{c_1, d_{15}, c_2\}$ are proprietary

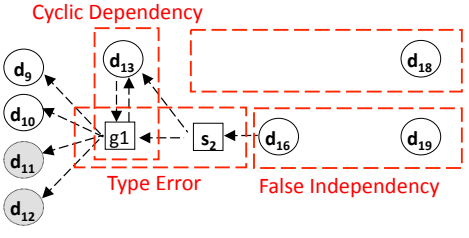


Fig. 3. Provenance graph after resolving all the privacy issues. The modified provenance graph introduces a cyclic dependency, a type error (the graph is non-bipartite), and a false independence.

shown in Figure 2(a). A recursive query is used to retrieve all the data and invocation nodes upstream from d_{18} and d_{19}, i.e., the nodes on which the latter depend. The resulting subgraph (PG') is shown in Figure 2(b). Note that the lineage of d_{20} up to s_3 is not in the lineage of d_{18} and d_{19} and hence not included in PG'. Further assume that before publishing PG', the user also wants to sanitize the provenance data as it may have various privacy issues as shown in Figure 2(b).

Figure 3 shows the provenance graph we get after sanitation by (i) removing the value refenences from data nodes d_{11} and d_{12}, (ii) abstracting nodes m_1, d_{14}, and s_1 into a group node g_1, and (iii) removing nodes c_1, d_{18}, and c_2. In this modified provenance graph, we see that there is a cycle between the data node d_{13} and invocation node g_1; a type error for the edge between invocation nodes s_2 and g_1 (the graph should be bipartite); and there are no dependencies from data nodes d_{18} and d_{19} to the rest of the graph. Thus, we need a systematic way to customize provenance while respecting general properties of the graph (e.g., acyclicity, bipartiteness) and while preserving correctness and completeness of the remaining provenance information.

In this work, we develop a strategy to (i) hide sensitive information as specified by the user, (ii) maintain all relevant nodes, which do not have low level details or proprietary information, in the customize provenance graph, (iii) maintain the original direct and transitive dependencies among the relevant nodes, and (iv) produce only

graphs that comply with the structural properties of provenance graphs (acyclicity and bipartiteness).

Outline. In Section 3, we describe the provenance model, user requests and provenance policies, and the logical architecture of the framework. Section 4 presents our key ideas, and techniques to solve individual policy violations by introducing new, anonymous nodes. We discuss related work and conclude in Section 5.

3 Provenance Publisher

In our recent work, we developed the system PROPUB [10], which uses a declarative approach to publish customized policy-aware provenance. PROPUB accepts a provenance graph and three inputs: (1) *user requests* to publish and customize provenance, (2) *provenance policies*, modeled as integrity constraints aiming to ensure the validity of the customized provenance graph, and (3) a (total) *preference order* among provenance policies. PROPUB checks whether all user requests and provenance policies can be satisfied together. If not, the approach selects a subset of requests and policies according to the user-specified ranking. The outputs of PROPUB are the customized provenance graph, as well as a list of satisfied and ignored user requests and policies.

In this work, we present an extension to PROPUB that invents new, anonymous nodes that are inserted in the customized graph. We show that with this technique, it is possible to always satisfy all user requests and policies simultaneously, without the need of a user-specified preference order. For example, by subsequently applying the user requests in a specific way, none of the provenance policies as described in Table 3 will be violated.

Provenance Model. Our provenance model is based on the Open Provenance Model OPM [14] and our earlier work [15]: A *provenance* (or *lineage*) *graph* is an acyclic graph $PG = (V, E)$, where the nodes $V = \mathrm{D} \cup \mathrm{I}$ represent either *data* items D or actor *invocations* I. The graph G is bipartite, i.e., the edges $E = E_{\texttt{use}} \cup E_{\texttt{gby}}$ are either *used* edges $E_{\texttt{use}} \subseteq \mathrm{I} \times \mathrm{D}$ or *generated-by* edges $E_{\texttt{gby}} \subseteq \mathrm{D} \times \mathrm{I}$. Here, a *used* edge $(\texttt{i}, \texttt{d}) \in E$ means that invocation `i` has read `d` as part of its *input*, while a *generated-by* edge $(\texttt{d}, \texttt{i}) \in E$ means that `d` was *output* data, written by invocation `i`. Data and invocation nodes have opaque identifiers. We use the relations `data` and `actor` to map each data and invocation node to a URL where the data value can be retrieved or the implenting actor identified. The PROPUB Datalog implementation uses the schema shown in Table 1.

User Requests. Table 2 summarizes the user requests supported by our system. User requests are asserted as relational facts, which together with PROPUB rules can be used by a Datalog rule engine to infer additional facts or to check integrity constraints. A user request can be a publication request or a customization request.

Figure 4 shows examples of publication and customization requests. The relation `lineage` defines the user's initial publication requests. The relations `abstract`, `hide`, and `anonymize` are used to abstract the nodes with low level details, to remove proprietary nodes, and to remove the value references from the sensitive nodes, respectively.

Table 1. PROPUB Provenance Model

Relation	Description
used(I, D)	An edge specifying that the invocation I used the data artifact D.
gen_by(D, I)	An edge to indicate that the data artifact D was generated by invocation I.
actor(I, A)	An invocation node I, which was executed by actor A.
data(D, R)	A data artifact node D, whose value can be retrieved using the reference R.
dep(X, Y)	Combined dependency relation dep = used ∪ gen_by. Specifies that node X depends on node Y, irrespective of their types.

Table 2. User requests for lineage publication and customization

User Request	Description
ur:lineage(D)	Selects the complete lineage for the data artifact D
ur:anonymize(N)	Erases the actor/process identity or the data reference from the node N
ur:hide(N)	Removes the invocation or data node N
ur:abstract(N, G)	Collapses all nodes N to the abstract group G

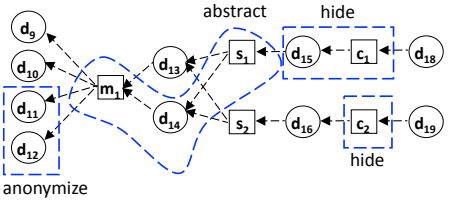
Provenance Policies. As mentioned above, a provenance graph is a bipartite DAG in our model. Moreover, an invocation can read (i.e., use) many data artifacts, but a data artifact is generated by exactly one invocation. We use three provenance policies, represented as logical integrity constraints, to verify if these structural properties are satisfied in the customized provenance graph CG that results from applying all customization requests to PG'. The framework supports two more provenance policies to ensure the correctness and completeness of information, see Table 3.

We use a set of integrity constraints (ICs) to check whether the provenance policies defined in Table 3 are satisfied. Table 4 lists the *witness relations* that are used to detect particular IC violations and report the "culprits". For example, we can detect a write conflict, where a data node D is created by different invocations X and Y, with the Datalog rule: `ic:wc(X,Y) :- gen_by(D,X), gen_by(D,Y), X ≠ Y.`

3.1 Logical Architecture

The logical architecture of the framework is shown in Figure 5. The user submits a set of publication and customization requests U_0. The module Direct-Conflict-Detection detects direct conflicts among the given user-requests. For example, a ur:hide and a ur:lineage request on the same node are directly in conflict. The user then needs to update her original requests until all direct conflicts are resolved, resulting in a consistent, conflict-free user request U. The Lineage-Selection module computes the subgraph PG', containing all to-be-published data items together with their data lineage.

The User-Request-Application module *applies* all the ur:hide, ur:abstract, and ur:anonymize requests in U on PG'. It deletes from PG' all data and invocation nodes selected by the ur:hide and ur:abstract requests, together with their associated gen_by and used edges. This module then applies the ur:anonymize requests to remove value references (in case of a data node) or references to the source code (in case of an invocation node). As this module deletes nodes and incident edges, two relevant nodes may now appear independent, even though they were dependent in PG'.



(a) Provenance subgraph PG' with outlined customization requests.

Publication User Requests
lineage(d_{18}). lineage(d_{19}).

Customization User Requests
abstract(d_{14}, g1). abstract(s_1, g1). abstract(m_1, g1).
hide(d_{15}). hide(c_1) . hide(c_2).
anonymize(d_{11}). anonymize(d_{12}).

(b) Publication and customization requests as Datalog facts.

Fig. 4. (a) User requests to abstract, anonymize, and hide parts of PG'; e.g., proprietary nodes as in Fig. 2(b), can be hidden using the `ur:hide` user request. (b) User requests represented as Datalog facts: e.g., to abstract nodes $\{m_1, d_{14}, s_1\}$ into an abstract node g_1, we use $\texttt{abstract}(m_1, g_1)$, $\texttt{abstract}(d_{14}, g_1)$, and $\texttt{abstract}(s_1, g_1)$. The module prefix "`ur:`" is optional here.

Table 3. Provenance Policies

Provenance Policy	Description
No-Write Conflict (NWC)	A data artifact can be written by only one invocation.
No-Cyclic Dependency (NCD)	There is no cycle in the provenance graph.
No-Type Error (NTE)	Bipartite graph: edges only between data and invocations.
No-False Dependence (NFD)	Two nodes are dependent in CG only if they are dependent in PG.
No-False Independence (NFI)	Two nodes are independent in CG only if they are independent in PG.

The Dependency-Injection module connects all relevant nodes in the customized provenance graph CG by reproducing the same dependencies as found originally in PG'. While connecting nodes, this module introduces anonymous nodes to avoid cycle-dependency, type-error, write-conflict, and false-dependency constraint violations.

The final output, the customized provenance graph CG, satisfies all the provenance policies mentioned in Table 3, honors all the conflict-free user requests, and maintains all relevant nodes.

4 Approach

The basic idea of our approach is to first remove data or invocation nodes based on the user's `hide` and `abstract` requests, and then to connect the remaining nodes using three key ideas: (i) maintain all relevant nodes, (ii) maintain their dependencies, and (iii) invent new, anonymous nodes to avoid policy violations.

Maintain Relevant Nodes. In case nodes have sensitive or proprietary information, or simply too much, low level details (cf. [16]), the user can request those nodes to be removed, abstracted, or anonymized using the requests described above. All the data and invocation nodes, which are not selected using `ur:abstract` or `ur:hide` are considered *relevant* nodes to the user. Our approach does not remove any of these nodes from PG' and in turn maintains them in CG.

Table 4. Integrity constraint relations used to detect policy violations

Constraint	Description
ic:wc(X, Y)	Write conflict: two invocations X and Y are generating the same data node.
ic:cd(X)	Cyclic dependency through node X.
ic:te(X, Y)	Type error: nodes X and Y are connected via *used* or *gen_by* edges, but don't have the corresponding node types.
ic:fd(X, Y)	False dependency: node Y depends on X in CG, but not in PG.
ic:fi(X, Y)	False independence: node Y depends on X in PG, but not in CG.

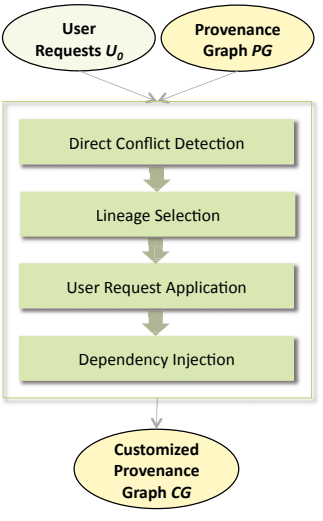


Fig. 5. Logical Architecture: The framework accepts a set of user requests and the provenance graph and runs through a series of four modules to produce the customized provenance graph.

Maintain Dependencies. While removing nodes from PG′, as a consequence of the user's customization requests, we also remove the associated gen_by and used edges. This may make CG incomplete (i.e., dependencies are omitted) as shown in Fig. 3. Our framework avoids these provenance policy violations by maintaining the dependencies among the relevant nodes as described in Section 4.2.

Inventing New Nodes. While connecting the remaining data and invocation nodes, the framework may invent new nodes to avoid policy violations (e.g., an invocation node is invented to connect two data nodes to avoid NTE violations, i.e., type errors). New nodes can be data or invocation nodes and have no relation to any of the nodes being replaced. In particular, new nodes are anonymous, i.e., do not have references associated, so that no sensitive information is revealed, as requested by the user.

4.1 Dealing with Structural Constraint Violations

Before describing our approach in detail (see Section 4.2), we first provide an overview of the possible remedies that we can use to deal with certain constraint violations.

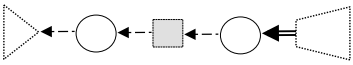


Fig. 6. An invocation node is invented to connect two data nodes, avoiding a NTE violation

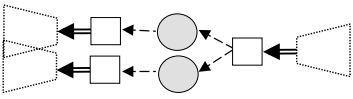


Fig. 7. Two data nodes are invented between invocation nodes, avoiding a NWC violation

No-Type Error. This policy is violated in case there is a direct dependency between two nodes of the same type (i.e., a direct dependency between data nodes or invocation nodes). While connecting two data nodes, our framework invents an invocation node to avoid this policy violation as shown in Fig. 6. Similarly, the framework invents a data node to connect two invocation nodes.

No-Write Conflict. If a data node depends on two different invocation nodes (i.e., the data node is generated by two different invocations), this policy is violated. This may occur when inventing a data node and connecting it with two or more `gen_by` edges to maintain dependencies. While adding edges to an invented data node or connecting a disconnected relevant data node, our framework ensures that only one `gen_by` edge is added. Thus, our framewrok avoids this policy violation as shown in Fig. 7.

No-False Independence. This policy is violated if two nodes are dependent in PG', but appear independent in CG. This may occur as a result of user requests, as shown in Fig. 8. Our framework connects the corresponding nodes in CG, to preserve the dependence present in PG'.

No-False Dependence. This policy is violated if two nodes are independent in PG', but appear dependent in CG (Fig. 9). Our framework avoids this conflict by connecting the relevant nodes using a number of anonymous nodes to preserve the dependencies in CG as they were in PG' (see Section 4.2 for details).

No-Cyclic Dependency. This policy is violated, if there is a cyclic dependency in the provenance graph, i.e., there are nodes in the graph that depend on themselves (either directly or indirectly via other nodes).[1] If the original PG' was acyclic, then the resulting graph CG will also be acylic, as we do not introduce cycles between nodes from PG', nor do we introduce cycles involving newly inserted nodes.

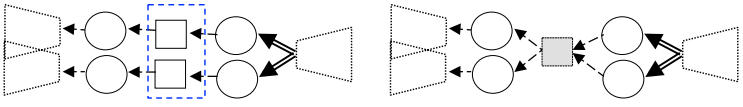
4.2 Module Implementation

We now provide more details about each of the modules mentioned in Section 3.1. In our framework, the provenance graph PG and user requests U_0 are given as logic facts

[1] Recall that provenance (lineage) graphs are inherently acyclic, since they behave like causality graphs, where an effect (the data output of a computation) cannot precede its cause (the inputs to that computation).

(a) User request to `hide` nodes (b) Policy Violation

Fig. 8. Once the requests to hide nodes are executed (a), a false independence arises (b)



(a) User request to `abstract` nodes (b) Policy Violation

Fig. 9. Once the invocation nodes are abstracted (a), false data dependencies appear (b)

(EDB or base relations in Datalog parlance). All four modules in our framework are specified declaratively, as a set of Datalog rules that are evaluated over PG and U_0, to derive the customized provenance graph CG. The modules Direct-Conflict-Detection and Lineage-Selection are implemented using Datalog rules as shown in our earlier work [10]. The lineage subgraph PG′, expressed as the dependency relation `dep'`, is calculated (based on the `ur:lineage` publication user requests) as follows:

```
dep'(X,Y) :- ur:lineage(X), dep(X,Y).
dep'(X,Y) :- dep'(_,X), dep(X,Y).
```

Note that `dep'` is not the transitive closure of `dep` but rather the subgraph of edges in `dep` that is reachable from the nodes in `lineage` that the user requested to be published.

User-Request-Application. This module accepts the provenance graph PG′ and the conflict-free user requests as inputs. It removes from PG′ the nodes selected by the `ur:abstract` and `ur:hide` requests and their incident edges. For example, the following rules are used to apply the `ur : hide` user requests:[2]

```
del_node(N) :- ur:hide(N), node'(N).
del_dep(X,Y) :- ur:hide(X), dep'(X,Y).
del_dep(X,Y) :- ur:hide(Y), dep'(X,Y).
```

In a similay way, `ur:abstract` user requests are applied. This module then applies all the `ur:anonymize` user requests by removing the references to the value for the selected data nodes and removing the references to the source code for the selected actor nodes. At the end of this module, we get a graph with only relevant nodes, in which some of them are anonymized. However, many of the dependencies among relevant nodes in PG′ may now be missing in this graph.

Dependency-Injection. The objectives of this module are to (i) connect all the relevant nodes (bringing back lost dependencies) using a minimum number of invented nodes, and (ii) maintain the original direct and transitive dependencies among the remaining

[2] Relations whose name starts with "`del_`" denote auxiliary relations that mark items to be deleted, here, e.g., nodes and edges.

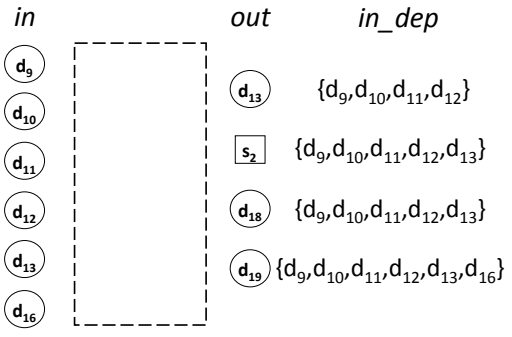


Fig. 10. Stage I: Nodes of the in and out sets and in_dep dependencies from out to in nodes

nodes. This module performs a 3-stage process to achieve these goals. We describe the stages below:

In the first stage, the framework develops in and out sets as shown in Fig. 10. Here in is a set of relevant nodes (those in $\texttt{dep}'$), on which one or more nodes from the `del_node` relation are dependent, while out is a set of relevant nodes, which are dependent on one or more nodes in the `del_node` set. In our running example from Fig. 4(a), we have $in = \{\texttt{d}_9, \texttt{d}_{10}, \texttt{d}_{11}, \texttt{d}_{12}, \texttt{d}_{13}, \texttt{d}_{16}\}$ and $out = \{\texttt{d}_{13}, \texttt{s}_2, \texttt{d}_{18}, \texttt{d}_{19}\}$.

The framework also calculates an in_dep relation for each node in out: $\texttt{in\_dep}'(o,i)$ holds for a node $o \in out$, if o depends (directly or transitively) on $i \in in$.

Given $o \in out$, the in_dep set (for o) is the set of inputs i on which o depends, i.e., $in\_dep_o = \{i \in in \mid \texttt{in\_dep}(o, i) \text{ holds}\}$. For example, data node $\texttt{d}_{19}$ is dependent on all the members of the in set (see Fig. 10).

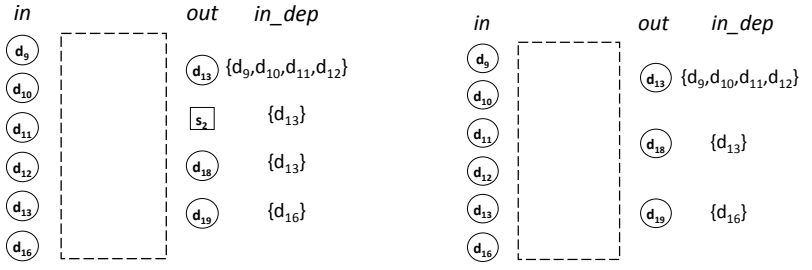
We use the following Datalog rules[3] to calculate in, out, and in_dep:

```
in(Y) :- del_node(X), dep'(X,Y), ¬del_node(Y).
out(X) :- del_node(Y), dep'(X,Y), ¬del_node(X).
in_dep(X,Y) :- dep'*(X,Y), in(Y), out(X).
```

In the second stage, the framework analyzes the dependencies among the nodes of a specific in_dep_o set of a node o. In case there is a node $i \in in\_dep_o$ which depends on another node $i' \in in\_dep_o$, the framework removes i' from in_dep_o. All these i' nodes in in_dep_o are called *redundant* for the node o. The reason is that when o depends on i, then o will also transitively depend on i' (in a sense, i' is "covered" by i, since the lineage of i includes the lineage of i'). For example, node $\texttt{d}_{18}$ from the out set has an in_dep set with nodes $\texttt{d}_9$, $\texttt{d}_{10}$, $\texttt{d}_{11}$, $\texttt{d}_{12}$ and $\texttt{d}_{13}$. Now, since $\texttt{d}_{13}$ is dependent on all of $\texttt{d}_9$, $\texttt{d}_{10}$, $\texttt{d}_{11}$, and $\texttt{d}_{12}$ as shown in Fig. 4(a), the framework optimizes this in_dep set by removing all nodes except the node $\texttt{d}_{13}$. This process is performed for all elements from the out set. The result is shown in Fig. 11(a).

Next, we check if there is any (transitive) dependency that uses only non-deleted nodes and edges between a node from the out set and the nodes of its in_dep set. In case there is, the respective node from the in_dep set is removed, since the required dependency is already present in the graph.

[3] Here $\texttt{dep}'^*$ denotes the transitive closure of $\texttt{dep}'$ and is defined as usual in Datalog.



(a) in, out, and reduced in_dep sets (b) reduced out set

Fig. 11. Stage II: (a) in_dep sets are "optimized" (reduced) by removing redundant nodes. (b) out is further reduced by removing nodes which are directly dependent on all in_dep nodes.

For example, there is a direct dependency from $\mathtt{s}_2$ to $\mathtt{d}_{13}$. Thus $\mathtt{d}_{13}$ is removed from the in_dep set of the invocation node $\mathtt{s}_2$. Finally, if an in_dep set for an out node becomes empty, the node is removed from the out set. This is the case for the invocation node $\mathtt{s}_2$ as shown in Fig. 11(b). Since there is already an edge between $\mathtt{s}_2$ and $\mathtt{d}_{13}$, the framework does not add an additional edge.

```
in_dep1(X,Y2):- in_dep(X,Y1), in_dep(X,Y2), dep'*(Y1,Y2).
in_dep2(X,Y) :- in_dep(X,Y), ¬in_dep1(X,Y).
dep''(X,Y) :- dep'(X,Y), ¬del_node(X), ¬del_node(Y).
dep''*(X,Y) :- dep''(X,Y). % transitive dependencies via remaining nodes
dep''*(X,Z) :- dep''(X,Y), dep''*(Y,Z).
in_dep3(X,Y):- in_dep2(X,Y), dep''*(X,Y).
in_dep4(X,Y) :- in_dep2(X,Y), ¬in_dep3(X,Y).
```

In the third and final stage, we invent one data node for each invocation node `A` in the in set and updates all the in_dep sets by replacing `X` with the respective newly invented data node $\mathtt{D} = \mathtt{f}(\mathtt{A})$ while keeping the dependencies by connecting `D` with `A`. This is done to avoid type-errors. The Datalog rules are as follows:

```
in_actor(A) :- in(A), actor(A,_).
ins_data(f(A)) :- in_actor(A).
ins_dep(f(A),A) :- in_actor(A).
in_dep5(X,Y) :- in_dep4(X,Y), ¬in_actor(Y).
in_dep5(X,f(Y)) :- in_dep4(X,Y), in_actor(Y).
```

We can now calculate distinct in_dep sets using the following Datalog rules:

```
diff_in_dep(X1,X2):- in_dep5(X1,Y),in_dep5(X2,_),¬in_dep5(X2,Y).
diff_in_dep(X1,X2):- diff_in_dep(X2,X1).
same_in_dep(X1,X2):- in_dep5(X1,_), in_dep5(X2,_),
                     ¬diff_in_dep(X1,X2).
not_smaller(X):- same_in_dep(X,Y), X > Y.
unique(X):- out(X), ¬not_smaller(X).
```

Here, the `unique` relation provides the list of out nodes with unique in_dep sets. The relation `same_in_dep` pairs out nodes having the same in_dep sets.

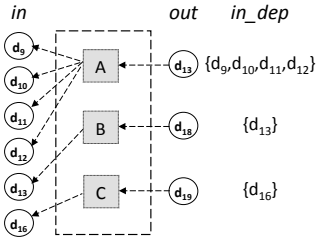


Fig. 12. Stage III: Dependencies are recreated among relevant nodes, based on the in_dep sets

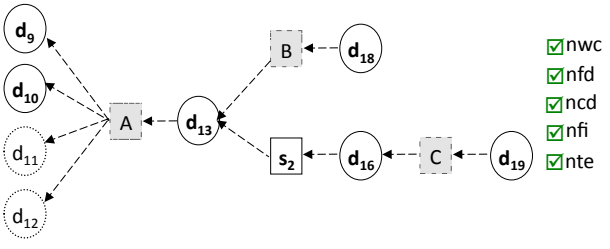


Fig. 13. Customized Provenance Graph CG after applying all user requests. CG satisfies all provenance policies and maintains all relevant PG nodes. `A`, `B`, and `C` are new anonymous nodes.

We then invents on data node `f(Y)` for each unique `same_in_dep` group `Y`, and inserts edges between `f(Y)` and all invocation nodes in this group:

```
ins_data(f(Y)) :- actor(X,_), same_in_dep(X,Y), unique(Y).
ins_dep(X,f(Y)) :- actor(X,_), same_in_dep(X,Y), unique(Y).
```

Finally, we invent one invocation node for each *same_in_dep* group, and insert dependency edges to connect the nodes in in and out based on the in_dep; see Fig. 12:

```
ins_actor(g(S)) :- unique(S).
ins_dep(D,g(S)) :- same_in_dep(D,S), unique(S), data(D,_).
ins_dep(f(S),g(S)) :- same_in_dep(X,S), unique(S), actor(X,_).
ins_dep(g(S),D) :- unique(S), in_dep5(S,D).
```

The result of this module is a graph with relevant and newly created nodes. For our example, it is shown in Fig. 13. ProPub has removed all the nodes selected using the `ur:abstract` and `ur:hide` user requests and invented three anonymous node `A`, `B`, and `C` to maintain the dependencies among the relevant nodes. The framework also anonymized the data nodes d_{11} and d_{12} selected using `ur:anonymize` user requests.

5 Summary and Conclusions

Data provenance can be used in many ways, e.g., to interpret results, diagnose errors, fix bugs, improve reproducibility, and generally to build trust on the final data products and the underlying processes [3,4,5,6,7]. In addition, provenance information can be

used to enhance exploratory processes [17,18,19], and techniques have been developed to efficiently store and query provenance from scientific workflow runs [20,21].

With the increasing use of provenance information, privacy issues become more important as well [7,8]. For example, provenance recorded by a scientific workflow system may carry sensitive information, such as data about human subjects in the case of biomedical studies, or proprietary information that a provenance provider might not want to reveal. By studying and analysing workflow provenance, one can, e.g., infer parts of the workflow specification or guess actor functionality from observing the relationships between inputs and outputs. The security view approach [5] limits the available provenance to a user by providing a partial view of the workflow through a role-based access control mechanism, and by defining a set of access permissions on actors, channels, and input/output ports as specified by the workflow owner at design time. The ZOOM*UserViews approach [16] allows to define a partial, zoomed-out view of a workflow, based on a user-defined distinction between relevant and irrelevant actors. Provenance information is restricted by the definition of that partial view of the workflow.

In our recent work [10], we developed ProPub, which uses a declarative approach to publish customized policy-aware provenance. Conflicts between user requests to hide or anonymize provenance information and provenance policies are resolved in [10] by removing additional nodes, beyond those requested by the user. In contrast, in this paper, we developed a new way to reconcile conflicts by inventing anonymous nodes that preserve the original lineage dependencies, without revealing information that the user wants to protect. Using this approach, we can now (i) honor all conflict-free user requests, (ii) comply with all provenance policies, (iii) maintain all relevant nodes in the final provenance graph, and (iv) maintain the original direct and transitive dependencies among the remaining nodes. Our current ProPub system is based on the open provenance model (OPM). We plan to extend our prototype to include provenance model extensions, e.g., to support structured data items, e.g., nested data collections [21].

References

1. Nature: 461, Special Issue on Data Sharing (September 2009)
2. Missier, P., Ludäscher, B., Bowers, S., Dey, S., Sarkar, A., Shrestha, B., Altintas, I., Anand, M., Goble, C.: Linking multiple workflow provenance traces for interoperable collaborative science. In: 2010 5th Workshop on Workflows in Support of Large-Scale Science (WORKS), pp. 1–8. IEEE (2010)
3. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: a survey. ACM Computing Surveys (CSUR) 37(1), 1–28 (2005)ss
4. Simmhan, Y., Plale, B., Gannon, D.: A survey of data provenance in e-science. ACM SIGMOD Record 34(3), 31–36 (2005)
5. Chebotko, A., Chang, S., Lu, S., Fotouhi, F., Yang, P.: Scientific workflow provenance querying with security views. In: The Ninth International Conference on Web-Age Information Management, WAIM 2008, pp. 349–356. IEEE (2008)
6. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for Computational Tasks: A Survey. Computing in Science and Engineering 10(3), 11–21 (2008)
7. Davidson, S., Khanna, S., Roy, S., Boulakia, S.: Privacy issues in scientific workflow provenance. In: Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science, pp. 1–6. ACM (2010)

8. Davidson, S.B., Khanna, S., Tannen, V., Roy, S., Chen, Y., Milo, T., Stoyanovich, J.: Enabling Privacy in Provenance-Aware Workflow Systems. In: CIDR, pp. 215–218 (2011)
9. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., et al.: The open provenance model core specification (v1. 1). Future Generation Computer Systems (2010)
10. Dey, S.C., Zinn, D., Ludäscher, B.: ProPub: Towards a Declarative Approach for Publishing Customized, Policy-Aware Provenance. In: Bayard Cushing, J., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 225–243. Springer, Heidelberg (2011)
11. Moreau, L., Ludäscher, B., Altintas, I., Barga, R., Bowers, S., Callahan, S., Chin, J., Clifford, B., Cohen, S., Cohen-Boulakia, S., et al.: Special issue: The first provenance challenge. Concurrency and Computation: Practice and Experience 20(5), 409–418 (2008)
12. Ludäscher, B., Bowers, S., McPhillips, T.M.: Scientific Workflows. In: Encyclopedia of Database Systems, pp. 2507–2511. Springer, Heidelberg (2009)
13. Davidson, S.B., Boulakia, S.C., Eyal, A., Ludäscher, B., McPhillips, T.M., Bowers, S., Anand, M.K., Freire, J.: Provenance in Scientific Workflow Systems. IEEE Data Engineering Bulletin 30(4), 44–50 (2007)
14. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., den Bussche, J.V.: The Open Provenance Model core specification (v1.1). Future Generation Computer Systems 27(6), 743–756 (2011)
15. Anand, M., Bowers, S., McPhillips, T., Ludäscher, B.: Exploring Scientific Workflow Provenance using Hybrid Queries over Nested data and Lineage Graphs. In: Winslett, M. (ed.) SSDBM 2009. LNCS, vol. 5566, pp. 237–254. Springer, Heidelberg (2009)
16. Biton, O., Cohen-Boulakia, S., Davidson, S.: Zoom* userviews: Querying relevant provenance in workflow systems. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment, pp. 1366–1369 (2007)
17. Davidson, S., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: SIGMOD Conference, pp. 1345–1350. Citeseer (2008)
18. Freire, J., Silva, C., Callahan, S., Santos, E., Scheidegger, C., Vo, H.: Managing rapidly-evolving scientific workflows. Provenance and Annotation of Data, 10–18 (2006)
19. Silva, C., Freire, J., Callahan, S.: Provenance for visualizations: Reproducibility and beyond. Computing in Science & Engineering, 82–89 (2007)
20. Heinis, T., Alonso, G.: Efficient Lineage Tracking For Scientific Workflows. In: SIGMOD, pp. 1007–1018 (2008)
21. Anand, M., Bowers, S., Ludäscher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: 13th Intl. Conf. on Extending Database Technology (EDBT), pp. 287–298 (2010)