# Efficient Provenance Storage

Adriane P. Chapman
University of Michigan
Ann Arbor, MI 48109
apchapma@umich.edu

H.V. Jagadish
University of Michigan
Ann Arbor, MI 48109
jag@umich.edu

Prakash Ramanan
Wichita State University
Wichita, KS 67260
ramanan@cs.wichita.edu

## ABSTRACT

As the world is increasingly networked and digitized, the data we store has more and more frequently been chopped, baked, diced and stewed. In consequence, there is an increasing need to store and manage provenance for each data item stored in a database, describing exactly where it came from, and what manipulations have been applied to it. Storage of the complete provenance of each data item can become prohibitively expensive. In this paper, we identify important properties of provenance that can be used to considerably reduce the amount of storage required. We identify three different techniques: a family of factorization processes and two methods based on inheritance, to decrease the amount of storage required for provenance.

We have used the techniques described in this work to significantly reduce the provenance storage costs associated with constructing MiMI [22], a warehouse of data regarding protein interactions, as well as two provenance stores, Karma [31] and PReServ [20], produced through workflow execution. In these real provenance sets, we were able to reduce the size of the provenance by up to a factor of 20. Additionally, we show that this reduced store can be queried efficiently and further that incremental changes can be made inexpensively.

## Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Scientific databases; E.m [**Data**]: Miscellaneous

## General Terms

Algorithms, Documentation, Experimentation

## Keywords

Provenance storage, Provenance compression

## 1. INTRODUCTION

It is well recognized that *Provenance*, or the history of an item, is as important for scientific data as it is in the study of arts and antiquities. Derived from the Latin word *provenire* "to come forth", the provenance of an item describes where the item comes from [9], and why it is found in its current place [4, 12]. Increasingly, provenance capture, storage, querying and form have received much attention [4, 5, 7, 9, 11, 14, 15, 20, 25, 34].

Recently, several scientific endeavors have been coupled with provenance management studies. Chimera [14] has been used with physics and astronomy data; myGRID [19] with biological data; Collaboratory for Multi-Scale Chemical Science (CMCS) [27] with chemistry data; Earth System Science Workbench (ESSW) [16] with earth science data. These experiments can involve $\sim$10TB of actual base data [1]. Unfortunately, the provenance information can grow to be many times larger than the base data [1, 11, 19, 27]. This is particularly true if the provenance is fine-grained, particularly rich, or a large number of operations have been performed on each piece of data.

For instance, in a recent provenance use study [19], provenance was attached to an experiment to determine the structure of protein sequences using GRID technology [15]. Starting with sets of protein sequences, a workflow containing about 12 steps was run. The base data was about 100Kb; the provenance size was approximately 1MB, which is ten times the data size [19]. Other scientific experiments run in conjunction with provenance storage produce similar results. MiMI [22], an online protein interaction database is 270MB; its provenance store is 6GB. We also have anecdotal evidence of a real deployed scientific data system where provenance information was partially removed to reduce the storage overhead [30].

To gain an appreciation of where the enormous size of provenance comes from, consider the following small example:

EXAMPLE 1. *There are many large protein interaction datasets, including HPRD [28] and BIND [2]. Figures 1(a)–1(b) show a small extract from each. A biologist may wish to integrate information from these two sources. To do this, she must first create a unified schema and transform the individual datasets into it. Then, she merges the datasets such that overlapping entries from different sources are combined. Finally, she runs each protein through a name normalizing script.*

*Figure 1(c) depicts the workflow described above. Notice that a piece of data starts at the bottom of the workflow, and can follow any path through it depending on the data itself. Figure 1(d) depicts the resulting dataset, along with the provenance associated with each data item. Even using a small provenance record and minimal manipulations, the size of the provenance already outweighs the size of the dataset.*

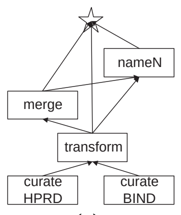
In this paper, we study how to reduce the space required to store provenance. Utilizing a generic provenance model, we describe two classes of space-saving algorithms. The first is a family of

```
<HPRD>
   <protein>
      <name>Wee1</name>
      <ref>P30291</ref>
      <descr>tyrosine kinase</descr>
      <PubMedID>15964826</PubMedID>
   </protein>
   <protein>
      <name>ABC1</name>
      <ref>O95477</ref>
      <descr>ATP binding cassette 1</descr>
      <PubMedID>16524875</PubMedID>
   </protein>
   <protein>
      <name>LXR</name>
      <ref>Q13133</ref>
      <descr>liver-X-receptor</descr>
      <PubMedID>16524875</PubMedID>
   </protein>
   <protein>
      <name>Chk1</name>
      <ref>AAC51736</ref>
      <descr>cell cycle checkpoint kinase</descr>
      <PubMedID>11251070</PubMedID>
   </protein>
</HPRD>
```

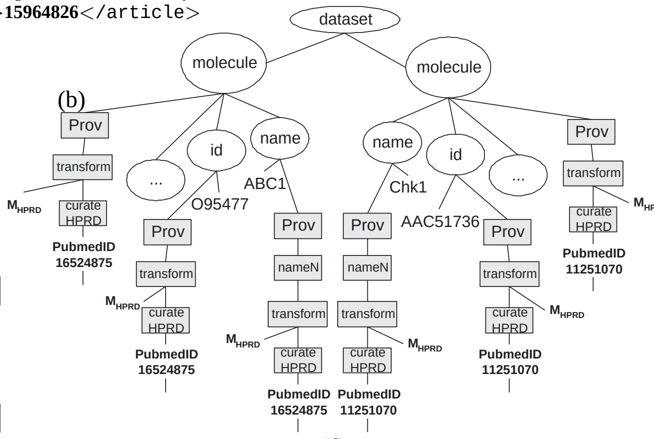
(a)

```
<BIND>
   <molecule>
      <name>WEE1</name>
      <extid>NP_003381</extid>
      <function>protein kinase activity</function>
      <article>15964826</article>
   </molecule>
</BIND>
```

(b)



(c)



(d)

**Figure 1: (a)-(b)A snapshot of proteins from HPRD and BIND. (c) The workflow used by the scientist. (d) Data and provenance for the ABC1 and Chk1 molecules, after transforming and merging HPRD and BIND.**

algorithms that reduce the size by removing duplicate provenance records and nodes. In any series of data manipulations, patterns can be found in the provenance data. A brief glance at Figure 1(d) can elucidate this even in our small example. We propose a series of *provenance factorization* techniques that find common subtrees and manipulate them to reduce the provenance size. Finding common subtrees is a known hard problem, studied in the context of eliminating common subexpressions [10, 13, 18]. We use this work to define a "Basic Factorization" algorithm. We then develop several enhancements crucial for good size reduction in our context. We additionally develop a second set of algorithms through *provenance inheritance*. There are two distinct algorithms in this set: one based on Structural Inheritance, and one based on Predicate Inheritance. Finally, we show how both types of Inheritance can be combined with Factorization to achieve maximum savings.

We require that provenance be queriable with the base data, so that queries such as "give me all molecules that came from HPRD" can be quickly answered. Strategies such as XMill [23] lead to a representation that is not queriable. Meanwhile, XML compressors such as XGRIND [32] that facilitate querying of compressed stores do not support a rich enough query language. In our context, it is imperative that users are able to specify relationships and joins between data and provenance information. Our methods meet this requirement, and can also be used in tandem with other XML compressors for maximum size reduction.

In Section 2, we lay the conceptual foundations needed to describe our methods. In Sections 3–4, we outline the Factorization and Inheritance algorithms used to reduce provenance size. In Section 5, we discuss how Factorization and Inheritance can be combined, the ability to query the reduced provenance stores, and effect of dataset incremental maintenance on the provenance stores. We test our reduction methods on real provenance stores, generated and stored via three distinct methods, and present results in Section 6. In Sections 7–8, we discuss related and future work as well as our conclusions.

## 2. THE PROVENANCE MODEL

There is currently no standard for representing provenance, although an initial attempt is the Open Provenance Model [26]. Several provenance capture systems exist [4, 7, 14, 19, 31], each with their own focus (actor vs data provenance), form (XML vs relational) and model. Also, several core systems have actively been used in the scientific process: Chimera [14], myGRID [19, 20], ESSW [16] and CMCS [27]. In addition, several workflow systems actively generate provenance [3, 17, 24, 29]. From a high-level perspective, all provenance systems have characteristics similar to the generic model we construct below.

Throughout this work, we call the basic logical data unit a *data item*. Data items may be tuples or attributes in a relational table, elements or attributes in XML, objects of arbitrary granularity in an OODB, etc. One data item may completely include, overlap with, or be totally disjoint from another data item. For example, in Figure 1(d), we show six data items: two molecule items, and their name and ID sub-items. A *dataset* is comprised of a set of data items. Datasets are often manipulated via workflows, whether explicit or implicit. A workflow is defined by an input description, output description and transformation rules. An explicit workflow is one generated by any number of workflow engines [3, 6, 24, 29]; an implicit workflow is executed by a user with a specific goal in mind, but without recording the executed processes. For example, MiMI [22] is created via an implicit workflow. A series of steps are executed, but they are neither executed within a formal workflow system, nor even fully documented. A workflow is modeled as a directed graph, where each node represents a manipulation (see Figure 1(c)).

DEFINITION 1. $Manipulation:$
*A $manipulation$ takes one or more datasets as input and produces a dataset as output.*

Thus, a manipulation is a discrete component of a workflow. An arc $(m_1, m_2)$ in a workflow graph indicates that the output of manipulation $m_1$ is fed as an input to manipulation $m_2$. We intentionally leave the granularity of a manipulation unspecified. Depending on the user's needs and the workflow system, this can be anything from a simple function to a whole program. A query can be a manipulation or a tree of manipulations within a workflow. The workflow in Figure 1(c) consists of five manipulations. A few common manipulations and examples follow:

MANIPULATION 1. $Selection$
*From an input dataset, selects a subset of data, based on some selection condition.*

EXAMPLE 2. *In the SDSS experiment [1], the first step is called fieldPrep. This manipulation extracts measurements of the galaxies of interest from the full dataset.*

MANIPULATION 2. $Translation$
*Transforms the input dataset $I$ based on a mapping $M$ and outputs dataset $I'$.*

EXAMPLE 3. *In Example 1, the input $I$ to the translation manipulation is the HPRD dataset and $M_{HPRD}$, a mapping from HPRD's schema to the researcher's own. The output $I'$ is the transformed HPRD dataset.*

The route a data item takes through a workflow can be represented by a tree. If a manipulation's output is an input to two different manipulations, this route can be represented by a tree with repeated nodes, similar to query evaluation plans. When an output data item $d$ results from an aggregation of two different input data items, its provenance record is a tree whose root element describes the aggregation step, and the two subtrees are the provenance structures associated with the two input data items. This tree is the provenance of the item $d$, and is shown in the "prov" subtrees in Figure 1(d). Note this is a tree-ified version of the provenance model described in the Open Provenance Model [26].

DEFINITION 2. $Provenance\ Record:$
*The record of input, and the manipulations applied to that input, to produce a new data item.*

DEFINITION 3. $Provenance\ Node$
*A single manipulation, its input and parameters, that comprise a part of the provenance record for a data item.*

DEFINITION 4. $Provenance\ Node\ Component$
*A single manipulation, input, or parameter that forms a part of a provenance node.*

A provenance record is a tree of *provenance nodes*. Each node in the tree corresponds to one manipulation, and has *components* that are inputs to the manipulation. For example, in Figure 1(d), the provenance record for the ABC1 molecule is a tree of two nodes. The transform node has the component $M_{HPRD}$. The curate node has the parameter PubMedID 16524875. Provenance contains a record of the manipulations used, and relates processes with input and output data. The provenance model we present is generic so that it can be applied to a variety of real-world provenance stores.

It would be incorrect to substitute the original workflow for information in the provenance store. This is because the provenance record for each data item is very specific, giving the exact path that data item took through the workflow: the original source data item it is based on, the exact parameters used in its manipulations, etc. For instance, did the transform manipulation for that data item use $M_{HPRD}$ or $M_{BIND}$? The workflow is much more general, applying to all data items.

As stated above, a data item may completely include another data item e.g. a tuple can contain an attribute. Each data item may have an associated a provenance record. In Figure 1(d), the ABC1 molecule data item has a provenance record, as does the O95477 ID data item contained within it.

DEFINITION 5. $Instance\text{-}level\ Provenance$
*The provenance record associated with a particular data item in the dataset.*

On the other hand, if a query was used to create the entire dataset, the query could be recorded as dataset-level provenance.

DEFINITION 6. $Dataset\text{-}level\ Provenance$
*The provenance record associated with an entire collection of data items.*

DEFINITION 7. $Provenance\ Store$
*The repository of all provenance records relating to a dataset and all data items in it.*

Throughout this paper, we let $D$ denote the original data store that contains $N$ data items (which may overlap), along with their provenance records (e.g. Figure 1(d)); let $size(D)$ denote the space used to store $D$. Each data item in $D$ has a provenance record associated with it; so the number of provenance records is also $N$. Each provenance record is a tree consisting of several provenance nodes. We let $n$ denote the total number of provenance nodes in $D$, where $n \geq N$.

## 3. PROVENANCE FACTORIZATION

Many items in a large data store may have similar or even identical provenance. If we could factor out common "sub-expressions" in the provenance of different items, these common portions could be stored just once for the whole data set rather than once for each item. We call this *provenance factorization*.

We consider Factorization at three different levels: factorization of identical provenance records, factorization of identical provenance nodes, and factorization of nodes that are identical except for their parameters.

After Factorization, the provenance records and nodes are stored in a provenance store that is separate from the data store. From each data item, there are one or more pointers to the provenance store, and in some cases, these pointers have some associated annotation.

### 3.1 Basic Factorization

Basic Factorization removes common provenance records; only one copy is stored. Each data item uses a provenance pointer to point to its provenance record. For example, in Figure 2(a), the ABC1 and LXR molecule data items are shown with their provenance. The Factorization algorithm discovers that the two provenance records are identical, and replaces each with a pointer to the record, now written separately in the provenance store, as shown in Figure 2(b).

The Basic Factorization Algorithm makes one pass over $D$, and separates the provenance records from the data items; its runtime is $O(size(D))$. When a provenance record is encountered, it is converted to a (possibly long) string; this string represents all the information in the record. The main data structure used is a hashtable on these strings; it is used to identify common provenance records. If the current provenance record $R$ is not found in the hashtable, a copy of it is stored in the new provenance store. In the data store,

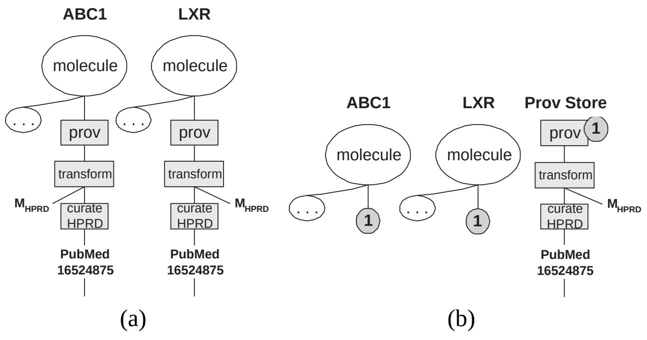


**Figure 2: Example of Basic Factorization. (a) ABC1 and LXR molecule data items. (b) Same data items after Basic Factorization.**

| Reduction Technique | Estimated Provenance Size |
|---|---|
| No Reduction | $N * S$ or $n * s$ |
| Basic Factorization | $N * p + N_1 * S$ |
| Node Factorization | $n * p + n_1 * s$ |
| Argument Factorization | $n * p + A * a + n_2 * s'$ |
| Structural Inheritance | $N_2 * S$ |
| Predicate Based Inheritance | $N * S/T$ |

Variables Used

| | |
|---|---|
| $N$ | total number of provenance records |
| $N_1$ | number of distinct provenance records; $N_1 \leq N$ |
| $N_2$ | number of data items whose provenance record is different from that of their parent data item; $N_2 \leq N$ |
| $n$ | number of provenance nodes; $n \geq N$ |
| $n_1$ | number of distinct provenance nodes; $n_1 \leq n$ |
| $n_2$ | number of distinct provenance nodes, after removing the arguments; $n_2 \leq n_1 \leq n$ |
| $S$ | average size of a provenance record |
| $s$ | average size of a provenance node |
| $s'$ | average size of a provenance node without arguments; $s' \leq s$ |
| $p$ | size of a pointer from the data store to the provenance store |
| $A$ | average size of an argument |
| $a$ | number of argument annotations |
| $T$ | number of data items that satisfy a predicate, and have common provenance records |

**Table 1: Estimated provenance size for each technique.**

the provenance record $R$ is replaced by a pointer to its copy in the provenance store.

Since there is one provenance record per data item, the number of (not necessarily distinct) provenance records is $N$. Let $N_1$ be the number of distinct provenance records. Let $S$ be the average size of a provenance record. The space used for storing provenance, before and after Basic Factorization, is shown in Table 1.

## 3.2 Node Factorization

Often, two data items will have distinct provenance records, but these provenance records will have many nodes in common. Node Factorization removes common provenance nodes. Only one copy of each node is stored in a separate provenance store. Provenance pointers are stored with data items to refer to these nodes.

Consider the workflow in Figure 1(c). Two distinct, but similar processes exist, curateHPRD and curateBIND. Consider two provenance records that contain different curation manipulations, but are otherwise identical. For instance, for provenance records $P_0 = A \rightarrow B \rightarrow C$, and $P_1 = A \rightarrow X \rightarrow C$, the provenance store after Basic Factorization will have one record for each of $P_0$ and $P_1$. Obviously, we can do better by factoring common nodes. This amounts to combining $P_0$ and $P_1$ as $A \rightarrow (B\ OR\ X) \rightarrow C$. The pointer from the data items to the provenance store is used to indicate which of $B$ or $X$ is present, i.e., which of $P_0$ or $P_1$ is the correct provenance record for that data item.

In order to accomplish this reduction, we must be able to determine that a) the $A$ nodes in $P_0$ and $P_1$ are equal, b) node $B$ in $P_0$ is similar to node $X$ in $P_1$, and c) the $C$ nodes in $P_0$ and $P_1$ are equal. Provenance Node Equality and Similarity are defined as follows.

DEFINITION 8. *Provenance Node Equality:*
*Two provenance nodes $a$ and $b$ are equal, denoted $a \overset{P}{=} b$, iff*
*i. they refer to the same manipulation,*
*ii. all parameters and input types to the manipulation are identical.*

DEFINITION 9. *Provenance Node Specific Similarity:*
*Two provenance nodes $a$ and $b$ are specifically similar, with respect to a similarity function $S_x$, if $S_x(a,b)$ = TRUE.*

Notice that similarity function values are dependent on the provenance nodes. For instance, we can define a similarity function $S_1(a, b) = \{a.name\ like\ \text{‘}curate\text{’}\ and\ b.name\ like\ \text{‘}curate\text{’}\}$. In this case $S_1(curateHPRD, curateBIND) = TRUE$, but $S_1(curateHPRD, transform) = FALSE$. We write $\mathcal{S}$ for the set of acceptable similarity functions, as defined by a provenance expert familiar with the provenance store in question.

DEFINITION 10. *Provenance Node Similarity:*
*Two provenance nodes $a$ and $b$ are similar, if they are specifically similar with respect to some similarity function $S_x() \in \mathcal{S}$.*

Provenance node similarity, as defined above, is a binary relation on the provenance nodes. We assume that the set $\mathcal{S}$ of similarity functions is such that this relation has the following properties.

- *Reflexive*: Each provenance node is similar to itself.
- *Symmetric*: If node $a$ is similar to node $b$, then $b$ is similar to $a$.
- *Transitive*: If $a$ is similar to $b$, and $b$ is similar to $c$, then $a$ is similar to $c$.

So, provenance node similarity is an equivalence relation. It divides the set of all provenance nodes in $D$ into equivalence classes, such that two nodes are similar iff they are in the same equivalence class. For example, consider the workflow shown in Figure 1c; there are five different kinds of manipulations. If we assume that all provenance nodes that pertain to each kind of manipulation are similar, then the similarity relation has five equivalence classes. If we further assume that all $curate_{HPRD}$ and $curate_{BIND}$ nodes are similar to each other, then the similarity relation has only four equivalence classes.

Using the above definitions, we can combine $P_0$ and $P_1$ as $A \rightarrow (B\ OR\ X) \rightarrow C$. But what happens if we change our provenance records slightly to: $P_3 = J \rightarrow K \rightarrow L \rightarrow M$ and $P_4 = J \rightarrow N \rightarrow O \rightarrow M$; we would like to combine them as $J \rightarrow (K\ OR\ N) \rightarrow (L\ OR\ O) \rightarrow M$. In other words, two provenance records could contain a long chain of similar provenance nodes. We can apply Node Factorization to such records using the following definitions.

DEFINITION 11. *Common Ancestor Node:*
*Two provenance nodes $a$ and $b$ have a common ancestor node if*
*i. $a.parent \overset{P}{=} b.parent$, or*
*ii. $a.parent$ and $b.parent$ are similar, and also have a Common Ancestor Node.*

DEFINITION 12. *Common Descendant Node:*
*Two provenance nodes $a$ and $b$ have a common descendant node if, for some children $c$ and $d$ of $a$ and $b$, respectively, we have*
*i. $c \overset{P}{=} d$, or*
*ii.$c$ and $d$ are similar, and also have a Common Descendant Node.*

DEFINITION 13. *Similar Chains:*
*Two equal length chains $C$ and $C'$ of provenance nodes are similar if*
*i. The topmost nodes in $C$ and $C'$ are equal,*
*ii. The bottommost nodes in $C$ and $C'$ are equal, and*
*iii. the $i^{th}$ node in $C$ and $C'$ are similar, $\forall i \neq top$ and $i \neq bottom$.*

Utilizing these definitions, our Node Factorization algorithm produces a smaller provenance store. When two nodes are determined to be Similar nodes in Similar Chains, they can be merged in the Provenance Store. The equivalence class that they belonged to and the Provenance Store now have one larger node. Moreover, because of the property of Similar Chains, the parents of these two merged nodes can also be merged and treated as one large node.

**Algorithm 1**: The Node Factorization with Similarity Algorithm.

```
    Input: Dataset $D$ with Provenance Records
    Input: Similarity Functions $\mathcal{S}$
    Output: Dataset with Provenance Store of Factorized Nodes
1   Hashtable H;
2   forall DataItems $d \in$ Dataset $D$ do
3       ProvenanceRecord r = d.provenance;
4       for ProvenanceNode n $\leftarrow$ r.nextNode() do
5           if ! H.contains( n ) then
6               H.put( n, pointer );
7           end
8           pointer = H.get( n );
9           writePointerInDataset( pointer );
10      end
11  end
12  forall ProvenanceNode n $\in$ Hashtable H do
13      $\mathcal{E}$= groupIntoEquivalenceClasses($\mathcal{S}$, H);
14  end
15  forall EquivalenceClasses $E \in \mathcal{E}$ do
16      forall ProvenanceNode $n, m \in E$ do
17          if n.parent isSimilarTo m.parent && n.child isSimilarTo m.child then
18              mergeInProvenanceStore();
19          else
20              writeInProvenanceStore();
21          end
22      end
23  end
```

Node Factorization makes one pass over $D$, and runs in time $O(size(D) + e^2h)$, where $e$ is the number of provenance nodes in an equivalence class and $h$ is the height of the provenance trees. In our experience, $size(D)$ greatly outweighs $e^2h$. Algorithm 1 contains the related pseudocode. The main data structure used is a hashtable on the provenance nodes. As each provenance node is encountered in the input data file, we search for it in the hashtable. When all provenance nodes have been seen, we find similar nodes in the provenance store. If a node $X$ is equal or similar to a node $B$ in the provenance store, and has a common ancestor and common a descendant with $B$, then $X$ and $B$ are unioned (i.e., OR-ed) in the provenance store; see the example of $(P_0, P_1)$ or $(P_3, P_4)$ given above. We further assume the similarity functions are coarse enough such that the following holds: the number of equivalence classes is some constant determined by $\mathcal{S}$, independent of $size(D)$.

We must expand the provenance pointer to include more information. The provenance pointer used in Basic Factorization is merely a pointer to the root of a particular tree in the reduced provenance store that corresponds to the provenance record of a data item. In our example, if only the base of the branch, $A$, were recorded for a data item's provenance, does the provenance contain $B$ or $X$? To remedy this, our provenance pointer must note which provenance nodes are being referenced.

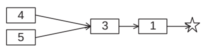
We have the following result about the content of the provenance store.

THEOREM 3.1. *Order Invariance:*
*Suppose that the set $\mathcal{S}$ of similarity functions is such that the provenance node similarity is an equivalence relation. Given a set of provenance records, the order in which they are merged into the provenance store by our Node Factorization algorithm does not affect the content of the provenance store.*
**Proof:** *Follows from the fact that the provenance nodes are divided into equivalence classes.*

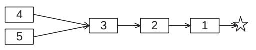
Recall from Section 2 that $n$ denotes the number of provenance nodes in $D$; let $s$ be their average size. Let $n_1$ be the number of distinct provenance nodes. The space used for provenance records, after Node Factorization, is shown in Table 1.

### 3.2.1 Factorization of Optional Nodes

Consider the two provenance records:

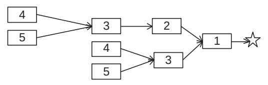
$P_0 =$ 

(1)

$P_1 =$ 

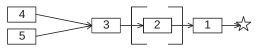
(2)

Node factorization will not combine them, because Manipulation_3 has different parents in $P_0$ and $P_1$. This will lead to the following provenance store:



(3)

This provenance store is larger than it could be. Instead, we would like a much smaller provenance store as:



(4)

where the square brackets indicate that [Manipulation_2] is optional.

We can achieve this result by using the provenance pointer to indicate whether the optional part applies in each instance. Once this machinery is in place, we can even merge two independent paths into one longer sequence. For example, $A \rightarrow B \rightarrow C \rightarrow D$ and $A \rightarrow E \rightarrow D$ can be merged as $A \rightarrow [(B \rightarrow C)\ OR\ E] \rightarrow D$, with the provenance pointer to indicate which of $(B \rightarrow C)$ and $E$ is present. Note that we no longer require similarity of merged nodes. In other words, $(B \rightarrow C)$ need not be similar to $E$.

Our algorithm for Node Factorization can be modified to also factor optional nodes; it will retain a single pass, $O(size(D))$ run time.

Unfortunately, we no longer have order invariance. Because the algorithm adds 'optional nodes' based on the parental ordering of the incoming provenance tree, and attaches them to the bottom of any other pre-existing optional nodes, the resulting provenance tree will be directly affected by the order in which we encounter the sequence of provenance nodes (e.g. $A \rightarrow [B \rightarrow C] \rightarrow [E] \rightarrow D$ is different from $A \rightarrow [E] \rightarrow [B \rightarrow C] \rightarrow D$).

## 3.3 Argument Factorization

We find that minor differences across provenance nodes can limit the utility of the Factorization algorithms discussed so far. For example, PubMedID, an input to the curateHPRD manipulation, can be different in otherwise identical provenance nodes. Because this one item is different, we no longer have a common provenance node to factor out. In Figure 3(a), the curateHPRD provenance nodes for the ABC1 and Chk1 molecules are identical except for the PubMedID, leading to no Basic or Node Factorization.

To permit maximum factorization of provenance under such circumstances, we consider provenance node components. We explicitly identify "arguments", and maintain them as part of the instance

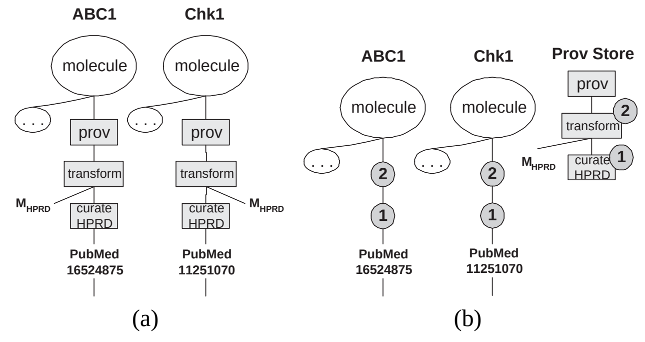


**Figure 3: Example of Argument Factorization. (a)** ABC1 **and** Chk1 molecule **data items. (b) Same data items with provenance pointers, after Argument Factorization.**

provenance pointer (from a data item to the provenance store) while factoring out the rest of the node. This begs the question, "What is an argument?" While the case is clear for the PubMedID in the example above, how about a parameter to a process that completely alters its execution? Rather than attempt to define the semantics of what is an argument, we say that a component is an argument if it exists in the provenance store less often than a user-specified threshold. The choice of this threshold is discussed in Section 6.7.

Argument Factorization involves two passes over $D$. The first pass uses a hashtable of provenance components; it is used to identify the arguments, by counting the number of times each component occurs. Using the provenance records in Figure 3(a) for example, we do a traversal of each provenance node component in each provenance record. The first component seen in this case would be PubMedID 16524875. It is placed in the hashtable. The next provenance component seen is the curateHPRD manipulation; it too is placed in the hashtable. This process continues until curateHPRD is seen again from the provenance record of Chk1. At this point, it is noted that curateHPRD, is already in the hashtable. As we continue through the rest of the the provenance nodes, we add new provenance components, and count those seen multiple times. Then, the components seen less often than the threshold (one in this example) are identified as arguments. The second pass is used to generate the new provenance store consisting of one copy of each distinct node sans its arguments; this process is similar to Node Factorization (Section 3.2). The result of these operations is shown in the provenance store of Figure 3(b).

Algorithm 2 contains the pseudocode for Argument Factorization. Argument Factorization makes two passes over $D$: one pass to place all the components into the hashtable (for determining the arguments), and one pass to factor the nodes sans their arguments. Each pass takes $O(size(D))$ time. Argument Factorization can use the same set of provenance pointers described in Section 3.2. The arguments are then attached to the provenance pointer. Additionally, we can make the following statements about Argument Factorization:

THEOREM 3.2. *Arg. Factorization Order Invariance: The order in which provenance records are added to the provenance store using Argument Factorization does not affect the final version of the provenance store.*
**Proof:***Proof is straightforward since factorization depends only on the count.*

**Algorithm 2**: The Argument Factorization Algorithm.

```
Input: Dataset $D$ with Provenance Records
Input: Arg_Threshold
Output: Dataset with Provenance Store of Argument Factorized Nodes
1  Hashtable H;
2  forall DataItems $d \in$ Dataset $D$ do
3      ProvenanceRecord r = d.provenance;
4      for ProvenanceNode n $\leftarrow$ r.nextNode() do
5          for ProvenanceComponent c $\leftarrow$ n.nextComponent() do
6              if H.contains( c ) then
7                  H.put( c, c.getCount++ );
8              else
9                  H.put( c, 1 );
10             end
11         end
12     end
13 end
14 forall DataItems $d \in$ Dataset $D$ do
15     ProvenanceRecord r = d.provenance;
16     for ProvenanceNode n $\leftarrow$ r.nextNode() do
17         for ProvenanceComponent c $\leftarrow$ n.nextComponent() do
18             int h = H.getCount( c );
19             if $h >$ Arg_Threshold then
20                 writePointerInDatasetToComponent;
21                 writeComponentInProvStore;
22             else
23                 writeArgumentInDataset;
24             end
25         end
26     end
27 end
```

Recall that $n$ is the number of original provenance nodes, and $n_1$ is the number of distinct provenance nodes; $s$ is their average size. Now, let $n_2$ be the number of distinct provenance nodes, after removing the arguments; so $n_2 \leq n_1 \leq n$. Let $s' \leq s$ be the average size of a node without arguments. Let $A$ be the average size of an argument, and let $a$ be the total number of argument annotations used on the pointers from the data store to the provenance store. The space used for provenance records, after Argument Factorization, is shown in Table 1.

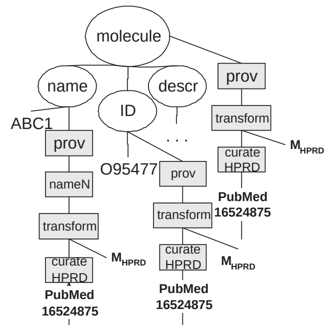
## 4. PROVENANCE INHERITANCE

Provenance Factorization, discussed above, finds similarities between the steps used to derive arbitrary data items. An orthogonal optimization finds similarities in a local portion of the data tree (Structural Inheritance) or between the provenance associated with data items of a particular type (Predicate Inheritance). When provenance is inherited by an item, there is no need to record any provenance with that item; the inheritance mechanism will correctly instantiate what is required.

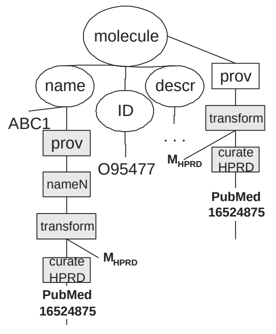
### 4.1 Structural Inheritance

There is often a repetition of provenance information at a fine-grained level because the same provenance is shared by data items that have a structural (parent-child or ancestor-descendant) relationship. Recall that data items can include other data items. For example, in Figure 4(a), the molecule data item contains the ID data item, which could in turn contain an idType data item. The provenance is the same for both the molecule and ID data items; however, both provenance records are recorded in a full provenance store. If, instead, we only record provenance for an item when it is different from that of its parent, we can reduce the space used. On the other hand, the name data item does not have the same provenance as the molecule data item, and so cannot inherit from its parent. Figure 4(b) depicts the provenance records using structural inheritance.

We use a single-pass, stack-based algorithm to determine ancestor-



(a)



(b)

**Figure 4: Example of Structural Inheritance. (a) The ABC1 molecule data item. (b) The same data item, after applying Structural Inheritance.**

descendant relationships and inheritance patterns; Algorithm 3 contains the pseudocode. Whenever we encounter a new data item, we compare its provenance with the provenance on top of the stack. If the two provenance records are not the same, write the provenance for the data item, otherwise write nothing. Push the provenance onto the stack. When we reach the end of a data item, pop a provenance from the stack. This one-pass algorithm takes $O(size(D))$ time.

Recall from Section 3.1 that the number of (not necessarily distinct) provenance records is $N$, and is the same as the number of data items; $S$ is the average size of a provenance record. Let $N_2$ be the number of data items whose provenance record is different from that of their parent data item. The space used for provenance records, with Structural Inheritance, is listed in Table 1.

**Algorithm 3**: The Structural Inheritance Algorithm.

```
   Input: Root DataItem, $d \in$ Dataset $D$
   Input: Stack S
   Output: Data Item with Structurally Inherited Provenance
   /*Note that this works through a dataset in tree form. If given a relational
     database, this method can still be used by mapping each data item to
     Database/table/tuple/row/ or Database/table/tuple/row/attribute etc. and
     building the tree in this manner.                                          */
1  ProvenanceRecord r = d.provenance;
2  ProvenanceRecord t = S.peek();
3  S.push( r );
4  if $r \neq t$ then
5      storeProvenanceWithDataItem;
6  end
7  for $d \leftarrow$ d.nextChild() do
8      structInherit( d, S );
9  end
10 S.pop();
```

## 4.2 Predicate Based Inheritance

Some provenance may apply to the dataset as a whole, or to items of a certain type within it. For instance, a query can be used to create an entire dataset; then, all data items in that set would have the same provenance. If every data item in a dataset contains the same provenance record, that record can be moved from the instance-level provenance to the dataset-level provenance. For instance, in Example 2, every data item was the result of the same selection process.

More frequently, it is the case that only some of the data in the dataset is created using a global operation. For instance, for each molecule, we may introduce a new attribute molecular weight computed based on its known sequence information. We would like to store the provenance once for all of the molecular weight items in the dataset, rather than storing it once for every data item. To accomplish this, we partition the data based on the satisfaction of a boolean predicate. An example of a valid predicate would an XPath expression such as $document(\text{“}dataset\text{”})//molecule$. If the associated provenance, or a subset of the provenance, is the same for all data items that satisfy some predicate, then the common provenance can be pulled out of each data instance. It can be stored at the dataset level, together with the boolean predicate that specifies the data items to which the provenance applies.

In general, there is a tradeoff between boolean predicate complexity and the efficiency of predicate-inherited provenance. It is possible to specify a boolean predicate that specifically targets just one data item within the dataset. In this case, it would be more efficient to merely store the provenance at the instance level. On the other hand, if the boolean predicate is not specific enough it will return too many data items and the likelihood of having a similar provenance among them is small. However, using some knowledge of the dataset, it is possible to find a set of boolean predicates that allow Predicate Based Inheritance on a large portion of the dataset. In our experiments, we use element type as the predicate. Thus, if all elements of the same name in our dataset contain nearly the same provenance, then the provenance, or subset of provenance components, can be stored at the dataset level, as shown in Figure 5. Note that we are agnostic about the actual schema used to represent the data set.

The Predicate Based Inheritance algorithm makes two passes over $D$; pseudocode can be found in Algorithm 4. In the first pass, we identify those provenance components that are common to all data items which satisfy a predicate; this is done for each predicate in a set of user-defined boolean predicates. If a data item $d$ satisfies the predicate $P$, and no provenance information yet exists for $P$ in the dataset-level provenance store, we create a new entry for $P$: It contains all the provenance components for $d$. If there already exists a predicate-provenance pair for $P$, we remove from it those components that are not in the provenance record for $d$. Once this first pass is completed, the provenance store will have a set of predicate-provenance pairs. A pair is present only if every data item that satisfies the predicate contains the same nonempty subset of provenance node components. A second pass over the entire dataset is then needed to write the remaining provenance that is not predicate-inherited.

Consider the runtime of our Predicate Based Inheritance algorithm. Let $Pred$ be a set of user-defined predicates that are $disjoint$ in the sense that no element can satisfy more than one predicate. Suppose that, for each element, it takes $O(t)$ time to determine which (if any) predicate in $Pred$ that element satisfies. Then the first pass takes time $O(Nt + size(D))$. The $O(size(D))$ part comes from the following: For each element $d \in D$ that satisfies a predicate $P \in Pred$, we either create a new predicate-provenance pair for $P$ (if $d$ is the first element seen that satisfies $P$), or modify the previously existing predicate-provenance pair for $P$. This takes time proportional to the size of the provenance record for $d$; over all $d \in D$, the total time is $O(size(D))$. The second pass involves, for each $d \in D$ satisfying predicate $P$, leaving out those components in the provenance record of $d$ that are in the dataset level predicate-provenance pair for $P$. This too takes time $\sum_d O(|provrecord(d)|) = O(size(D))$.

Recall from Section 3.1 that $N$ is the number of provenance records, and $S$ is their average size. Let $T$ be the average number of provenance records that satisfy a predicate, and have the same

**Algorithm 4**: The Predicate Inheritance Algorithm.

```
Input: Dataset $D$ with Provenance Records
Input: Predicate List $Pred$
Output: Dataset with Predicate Inherited Provenance
Hashtable H;
forall DataItems $d \in$ Dataset $D$ do
    if $d$ satisfies $P \in Pred$ then
        ProvenanceRecord r = d.provenance;
        if H.get( $P$ ) = null then
            List $M$;
            for ProvenanceNode n ← r.nextNode() do
                for ProvenanceComponent c ← n.nextComponent() do
                    $M$.add( c );
                    H.put( $P$, $M$ );
                end
            end
        else
            List $M$ = H.get( $P$ );
            List $N$;
            for ProvenanceNode n ← r.nextNode() do
                for ProvenanceComponent c ← n.nextComponent() do
                    $N$.add( c );
                end
            end
            forall $m \in M \notin N$ do
                $M$.remove( $m$ );
            end
        end
    end
end
forall DataItems $d \in$ Dataset $D$ do
    ProvenanceRecord r = d.provenance;
    if $d$ satisfies $P \in Pred$ then
        List $M$ = H.get( $P$ );
        if $M$ = null then
            writeProvForDataItem( r );
        else
            for ProvenanceNode n ← r.nextNode() do
                for ProvenanceComponent c ← n.nextComponent() do
                    if $c \in$ List $M$ then
                        r.remove( c );
                    end
                end
                if !r.isEmpty() then
                    writeProvForDataItem( r );
                end
            end
        end
    else
        writeProvForDataItem( r );
    end
end
forall $M \in$ Hashtable H do
    writePredicateProv();
end
```

provenance record. The space used for provenance records, using Predicate Inheritance, is shown in Table 1.

## 5. DISCUSSION

### 5.1 Combining Reduction Techniques

Any member of the Factorization Family (Basic, Node, Optional and Argument) can be applied independently to any dataset. Any member of the Factorization Family can also be used with Inheritance. Structural and Predicate Inheritance can also be combined. To apply such combinations, certain properties must be taken into account.

Using either Inheritance with any Factorization is straightforward, with two caveats: order and arguments. First, Inheritance should be performed before Factorization, since there will be fewer records to factor. Although the same correct results will occur regardless of ordering, the algorithms will run faster with Inheritance performed before Factorization. Second, provenance is not

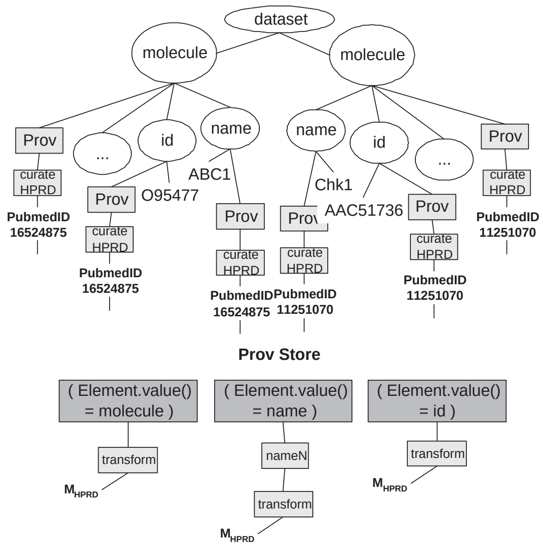


**Figure 5: The data and provenance after applying Predicate Inheritance to the data in Figure 1(d).**

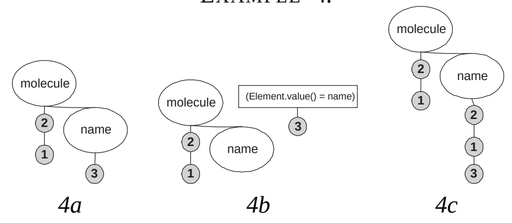
**Algorithm 5**: The Structural and Predicate Inheritance Algorithm.

```
Input: Root DataItem, $d \in$ Dataset $D$
Input: Predicate List $Pred$
Output: Dataset with Structural and Predicate Inherited Provenance
ProvenanceRecord r = d.provenance;
ProvenanceRecord t = S.peek();
S.push( r );
if $r \neq t$ then
    runPredicateInheritance( d, $Pred$ );
end
for d ← d.nextChild() do
    structAndPredInherit( d, S );
end
S.pop();
```

structurally inherited between data items that have the same set of manipulations but different arguments; only completely identical provenance records can be structurally inherited.

While both Structural and Predicate Inheritance can be applied individually to a dataset regardless of any Factorization usage, they can also be applied to a dataset jointly. Their conjunction is straightforward, with just a few details that should be noted. Structural Inheritance must be applied before Predicate Inheritance, as shown in Algorithm 5. Otherwise, reconstructing the provenance of a data item is potentially ambiguous. Consider the scenario:

EXAMPLE 4.



*Consider the* molecule *and* name *data items shown in 4a (grey circles are provenance nodes). If Predicate, then Structural Inheritance is applied to it, the reduced provenance will look like in 4b (assuming the provenance for the* name *data item gets moved to the dataset-level provenance store due to Predicate Inheritance).*

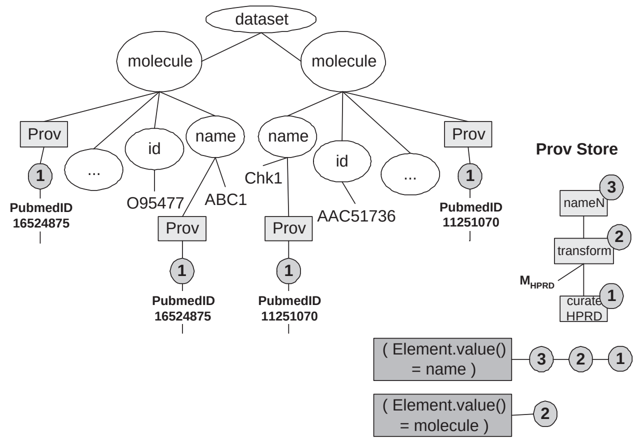


**Figure 6: The ABC1 and Chk1 records from Figure 1(d) after Structural and Predicate Inheritance and Argument Factorization.**

| Class | Description | Example |
|---|---|---|
| 1 | retrieve provenance for specific data | for $b in document("MiMI")/molecule where $b/name = "ABC1" return prov($b) |
| 2 | retrieve provenance of all items of type X | for $b in document("MiMI")/molecule return prov($b) |
| 3 | use provenance as a condition (low selectivity) | for $b in document("MiMI")/molecule where prov($b) = "HPRD" return $b |
| 4 | use provenance as a condition (high selectivity) | for $b in document("MiMI")/molecule where prov($b) = "PubMedID_15964826" return $b |
| 5 | join using provenance | for $b in document("MiMI")/molecule for $n in document("MiMI")/name where prov($b) = prov($n) return $b |

**Table 2: Sample provenance queries classed by complexity.**

*To re-instantiate the provenance, we would first look for Structural then Predicate Inheritance for the* name *data item and produce 4c; this is clearly incorrect. Because Structural Inheritance has the requirement that the entire provenance record is either inherited or not, this situation cannot occur if Structural Inheritance is performed before Predicate Inheritance.*

Figure 6 shows ABC1 and Chk1 with Structural then Predicate Inheritance applied to the entire dataset. The provenance for the ABC1 name data item is found at the dataset-level (predicate based) provenance, and in the reduced provenance pointer. The provenance store size estimation formulas in Table 1 can be modified to reflect combinations of techniques.

## 5.2 Querying Provenance

There are several classes of queries that utilize provenance. Table 2 describes some classes, and provides a sample query for each class from the MiMI query logs. Class 1 asks for the provenance of an individual data item. Class 2 seeks the provenance for all data items of a given type. In Classes 3–4, provenance is used as a selection condition for a data item, with low and high selectivity, respectively. Finally, Class 5 performs data item joins based on provenance information. These query classes were chosen from an analysis of MiMI's query logs, and represent a mixture of interest in the data item, based on its provenance, and the provenance itself.

## 5.3 Incremental Maintenance

We have described above how to reduce the cost of storing provenance, through Factorization and Inheritance, for a static data set with static provenance. We now consider what to do if changes are made to a data set and/or its associated provenance. How does the factorized and/or inherited provenance change? Can we manage these changes using incremental algorithms, without having to analyze the entire data set, and yet achieve the same small storage space as if the static algorithm had been run? Our answer is, for the most part, positive.

There are three different types of updates that we wish to consider. The first is deletion of data. This is simple – the only case needing any attention is a possible impact if the deleted item $d$ had children that structurally inherited provenance from it. In this case, we need to locally adjust the provenance for all children that inherited provenance from $d$.

The second type of update is insertion of data. For the entire family of Factorizations, the provenance associated with the new data is merged into the provenance store; only the new data and its provenance pointer(s) are written to the data store. If Structural Inheritance is used, the task is again simple – first consider the automatically inherited provenance at the newly inserted item $d$, and see if this is appropriate. If it is, we are done. If it is not, then we have to record the provenance with $d$. If $d$ has children, then the impact of the insertion on their structurally inherited provenance must also be considered. If this has changed, then the provenance recorded at these child items has to be modified accordingly. We can encounter a slightly more complicated problem when there is a data insertion while using Predicate Inheritance. Let the new data item $d$ satisfy a boolean predicate $P$ that has dataset-level provenance. If the dataset-level provenance for $P$ is a subset of $d$'s provenance, then this is easy: we store with $d$ only those provenance components that are not stored with $P$ at the dataset level. However, if the dataset-level provenance for $P$ is not a subset of $d$'s provenance, then we must do the following: Remove from the dataset-level provenance for $P$ those components that are not in $d$'s provenance, and re-insert those components as a provenance pointer at every data item (except $d$) that satisfies $P$.

The third case is where there is no change to the data, but we change the provenance associated with some data item (perhaps it had been recorded incorrectly). For this, the exact same steps occur as if the data item itself changed. Additionally, the provenance store can be added to, without making any changes to the instance-level provenance pointers.

## 6. EXPERIMENTAL EVALUATION

## 6.1 The Setup

Currently, few provenance stores exist along with datasets. Most are either destroyed after the dataset is created, never created. We were able to gain access to two very distinct styles of provenance stores. The first style is a complex workflow used to create a synthetic data set, involving 10 processes each consuming and producing 10 data items. Provenance storage for this workflow has been studied carefully, and in fact two different provenance storage structures have been used: Karma [31] and PReServ [20]. Even though both stores represent the same base provenance, the Karma provenance store is about 300MB while PReServ is about 500MB. The second style of provenance store is from an actual large data set, MiMI [22]. The implicit workflow to create each data item comprises only a few (2-4) steps, but with a very fine-grained approach. The base data in MiMI is 270MB, while the provenance store is 6GB.

| | Provenance Store |
|---|---|
| U | Unreduced Provenance Store |
| S | Structural Inheritance |
| P | Predicate Inheritance |
| SP | Structural & Predicate Inheritance |
| B | Basic Factorization |
| BS | Basic Factorization with Structural Inheritance |
| BP | Basic Factorization with Predicate Inheritance |
| BSP | Basic Factorization with Structural & Predicate Inheritance |
| N | Node Factorization |
| NS | Node Factorization with Structural Inheritance |
| NP | Node Factorization with Predicate Inheritance |
| NSP | Node Factorization with Structural & Predicate Inheritance |
| O | Optional Factorization |
| OS | Optional Factorization with Structural Inheritance |
| OP | Optional Factorization with Predicate Inheritance |
| OSP | Optional Factorization with Structural & Predicate Inheritance |
| A | Argument Factorization |
| AS | Argument Factorization with Structural Inheritance |
| AP | Argument Factorization with Predicate Inheritance |
| ASP | Argument Factorization with Structural & Predicate Inheritance |

**Table 3: Combinations of reduction techniques used in our experiments.**

We applied various combinations of our provenance reduction techniques, as shown in Table 3, to each provenance store. All experiments were run on a Dell Windows XP workstation with Celeron(R) CPU at 3.06GHz with 1.96GB RAM and 122GB disk space. The algorithms were implemented in Java, as a utility for reducing provenance storage after creation.

## 6.2 Storage Space

Figure 7(a) shows the space needed to store the provenance, according to each method; most techniques significantly reduce the size. As expected, Argument Factorization (A) does the same or better than Node (N) and Optional (O) for all the datasets. Whether Structural or Predicate Inheritance is better depends on the makeup of the dataset. MiMI has a very nested structure in which Structural Inheritance does very well. On the other hand, Karma and PReServ have flatter data unsuitable for Structural Inheritance, but use complex workflows that work well with Predicate Inheritance.

Inheritance combined with Factorization results in greater reduction for all data sets. Regardless of the Inheritance used, Argument Factorization is the clear winner. Using Argument Factorization with Structural Inheritance (AS), we produce a MiMI provenance store that is 5% the original size. Meanwhile, using Argument Factorization with Predicate Inheritance (AP) we can reduce the PReServ and Karma provenance stores to about 15% and 12%, respectively.

Because our reduction techniques are highly dependent on the data store and provenance store characteristics, we also created several artificial datasets to demonstrate each reduction technique's efficacy, based on the data and provenance characteristics; the results are shown in Figure 8. In Figure 8(a), the provenance store contained different amounts of provenance records, nodes and arguments, while the dataset and provenance store allowed contained different Structural and Predicate Inhertiance characteristics. It is clear that the Factorization techniques are highly dependent on the provenance store's distribution while the Inheritance techniques vary based on the dataset and provenance store.

Using a representative sample of the more interesting techniques, as the size of the provenance store grows, all our reduction algorithms remain $O(N)$, as shown in Figure 9.

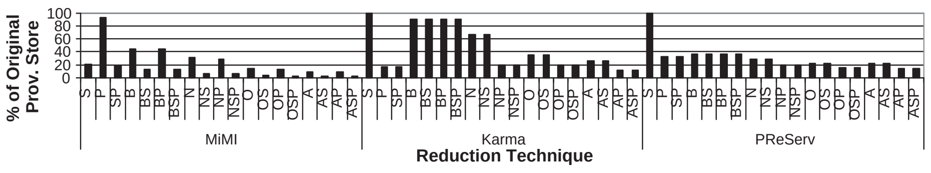
## 6.3 Reduction Time

Figure 7(b) shows the reduction time for each technique. As can be seen in Figure 7(b), the techniques perform differently on each provenance store. Reduction time is the worst for Node Factorization; Argument Factorization and Basic Factorization are not so bad. The reason for this is that Node Factorization maintains parental information, and will repeat the same node if it occurs in different places in the workflow, making the underlying data structures large and unwieldy. Argument Factorization has a large in memory structure to keep track of the arguments. However, because these arguments are not written, there are fewer round trips to the provenance store, thus keeping the time cost down. Karma and PReServ reduction is fast through all Factorization techniques. At first glance, it could be expected that the time to run Structural Inheritance should be less than the time to run both Structural Inheritance and Basic Factorization. However, we do not perform global Structural Inheritance then global Factorization which would make S <BS. Instead, for each data item, we test for Structural Inheritance, then immediately, reduce it via Factorization. The overall data structures are therefore smaller for BS than S, and this is reflected in the time. The reduction times presented were generated using an unoptimized implementation. Instead of reading provenance for a local tree, applying the reduction and writing it out, once the provenance structure is read in, it does not get written until the final provenance store build. In other words, as implemented, we have a large memory overhead which can be reduced by a more storage-intensive implementation. In this work, we are more concerned with the relative times between techniques.

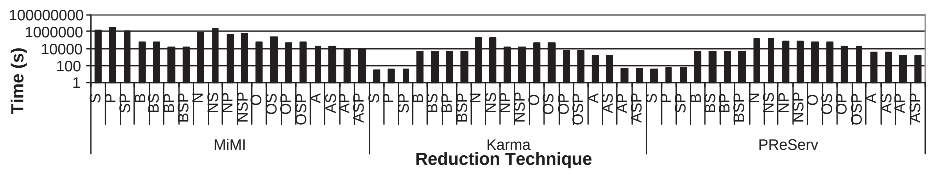
## 6.4 Query Time

The time it takes to reduce the provenance store, and the space used to store it, are only part of the overall needs of a functioning provenance system. It is imperative that the provenance remain queriable with the data itself. Because MiMI is queriable online, we were able to obtain the query logs, and use real queries generated by biologists. In Table 2, we describe five classes of queries from these real queries. Each query was run five times on a cold cache and the average of the three median times is reported. The only indexes built were element tag indexes. In order to accommodate Structural Inheritance, a new iterator was created. We obtained and modified Timber [21] such that it will find the provenance of a node even if it inherits from an ancestor. If the provenance is not found at a given node, the iterator returns the provenance of the parent node. Thus, this new iterator is at worst $O(h)$ time, where $h$ is the height of the data tree. Figure 10 shows the query execution time for queries in different classes.

Although our reduction techniques may make the provenance representation less straightforward, they not only save space, they can also reduce query time. A look at Figure 10 shows some interesting trends. For Classes 1, 3 and 4, in which queries have selectivity, queries on reduced stores perform on par, or better than the original store. In particular, Classes 3 and 4, using provenance as a condition in a low and high selectivity query respectively, show how the reduced stores can out-perform the original, based on size differences. Unfortunately, Class 2 queries perform worse on the reduced store. This is because every such query requires at least one join in the reduced stores. Finally, Class 5 query times on reduced stores are mixed compared to the original store. These queries require multiple joins, and it is impossible to push provenance instantiation higher in the query plan. This leads to poor performance in some cases, although Predicate Inheritance (P) and Argument Factorization with Structural and Predicate Inheritance (ASP) both do better than unreduced.

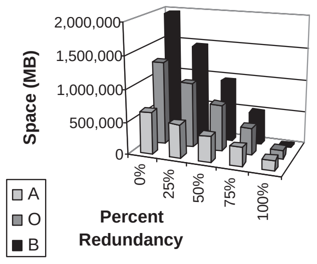


(a)

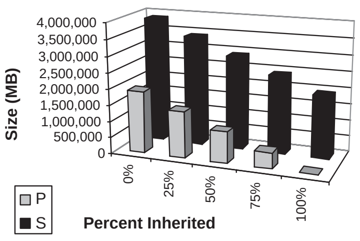


(b)

**Figure 7:** (a) Provenance storage space and (b) reduction time, for each method. See Table 3 for the key to letter codes.

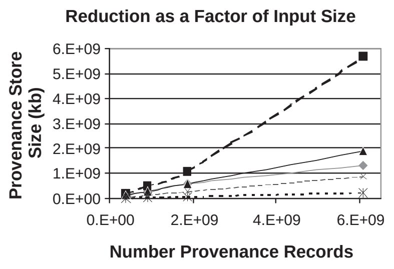


(a)

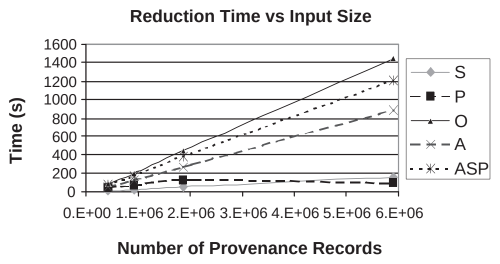


(b)

**Figure 8:** Provenance store size based on reduction technique, data and provenance characteristics. (a) Basic, Node and Argument Factorization. (b) Structural and Predicate Inheritance.



(a)



(b)

**Figure 9:** How the reduction algorithms scale based on input size in (a) space and (b) time.

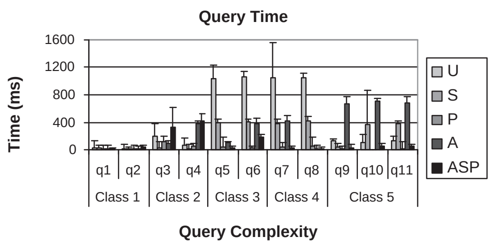


**Figure 10:** Query time for each query class on MiMI, for different reduction techniques.

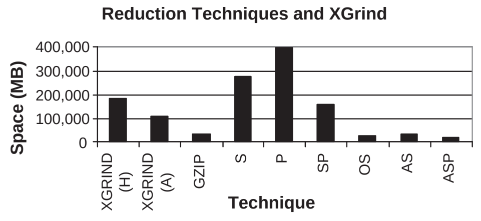


**Figure 11:** XGRIND, GZIP and a sample of Reduction Techniques applied to the MiMI Provenance Store.

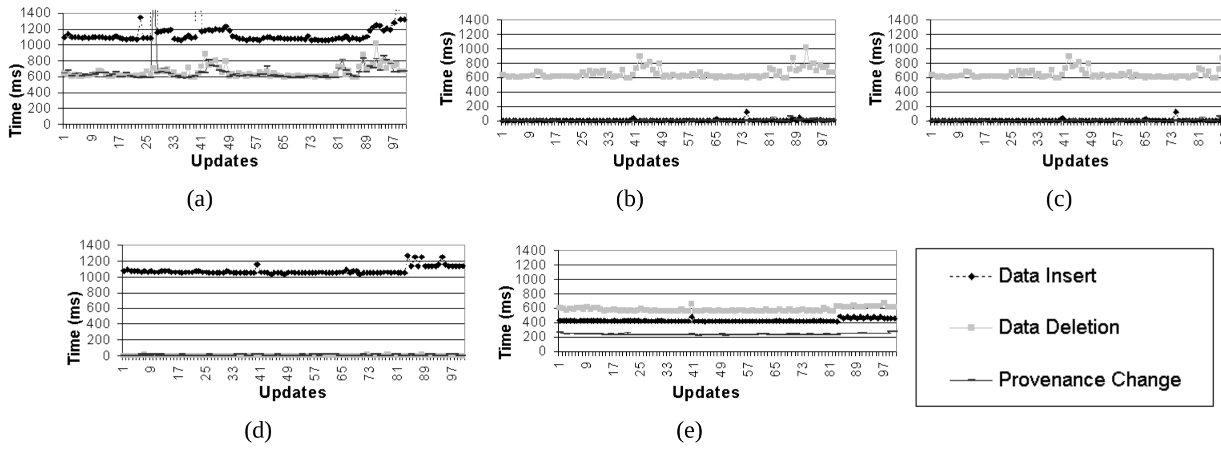


**Figure 12: Incremental Maintenance on provenance stores with (a) Structural Inheritance, (b) Predicate Inheritance, (c) Optional Factorization, (d) Argument Factorization, and (e) Argument Factorization with Structural and Predicate Inheritance.**

Structural Inheritance performs well across the board. This is due to a combination of reduced space and an $O(h)$ iterator. First, the provenance store is so reduced that the entire database is distinctly smaller. Second, no join needs to be performed, and the ancestor-lookup iterator is relatively fast. Predicate Inheritance appears all over the map in these queries. In some cases it does well, while in others it is almost the worst. Even within the same query class, it has wildly varying performance. A closer inspection of the provenance store itself contains the answer. In the case where there is a predicate-inherited item (e.g. type='name') in the provenance store, the method does very well. However, if no predicate inheritance exists for a certain element type, then the query performs poorly.

## 6.5 Incremental Maintenance

As discussed in Section 5.3, the reduction technique used can affect the complexity of incremental maintenance. Figure 12 shows how each store performs, for data insertion, data deletion and provenance changes. A random sequence of data inserts, deletions and provenance changes were performed, in equal measure, regardless of the reduction technique. For a provenance store with Structural Inheritance, Figures 12(a) and 12(e), the following inserts, deletes and provenance changes were performed: 1. insert a data item that Structurally Inherits provenance (from its parent); 2. insert a data item that does not Structurally Inherit provenance; 3. delete a data item with children that Structurally Inherit provenance from it; 4. delete a data item with no such children; 5. change provenance for a data item; with children that Structurally Inherit provenance from it; 6. change provenance for a data item with no such children. For a provenance store with Predicate Inheritance, Figures 12(b) and 12(e), the following inserts, deletes and provenance changes were performed: 1. insert a data item that Predicate Inherits provenance; 2. insert a data item that does not Predicate Inherit; 3. insert a data item that Predicate Inherits provenance, but breaks the inheritance pattern for all elements of that type; 4. delete a data item; 5. change provenance for a data item that Predicate Inherits provenance; 6. change provenance for a data item that does not Predicate Inherit; 7. change provenance for a data item that Predicate Inherits, but breaks the inheritance pattern for all elements of that type. For a provenance store with just Factorization, Figures 12(c) and 12(d), the following inserts, deletes and provenance changes were performed: 1. insert a data item; 2. delete a data item; 3. change provenance for a data item.

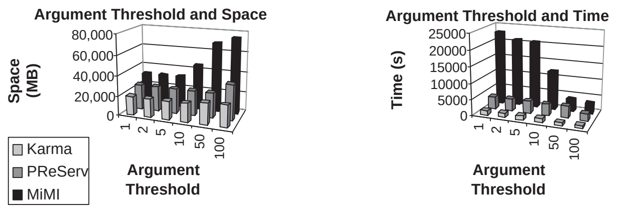
As shown in Figure 12, no matter what provenance reduction technique is used, updates are easy to perform. We would like to note that using Predicate Inheritance lowers the average time for a data insert. If the data item and provenance satisfies a predicate, then there is no need to manipulate the provenance store, thus saving time. Additionally, in provenance stores using straight Factorization, deletes and provenance changes are relatively cheap since there is no need to check inheritance dependencies. The take away point here is that incremental maintenance on a reduced provenance store is cheap.

## 6.6 Interaction with Other Compressors

As previously noted, traditional XML compression techniques are not suitable for our purposes because they do not result in a provenance store that is queriable along with base data. Even techniques such as XGRIND, which support keyword and path queries [32], do not have the full associative power needed to support joins between provenance and data. However, we have applied XGRIND using Huffman (H) and Arithmetic (A) encoding to the original provenance store, and compared the compressed size with our reduced stores. Additionally, although a gzipped document is not queriable, we included the gzipped provenance store as a well known comparison point. As shown in Figure 11, XGRIND on the unreduced provenance store creates a reduced store smaller than any of the Inheritance methods on their own. However, using combinations of reduction techniques it is possible to compress the provenance store smaller than XGRIND and still maintain the ability to query data and provenance together. Additionally, while we do not show the numbers here, it is possible to combine XGRIND and our reduction techniques to get an extremely small store.

## 6.7 Other Parameters

Figure 13 shows the relationship between Argument Threshold, the time to produce the reduced provenance store, and the size of the reduced store. In the case of MiMI, there is a drastic drop in the runtime between an Argument Threshold of 5 and 50. This drop is explained by the makeup of the provenance store. With a threshold



(a) (b)

**Figure 13: Argument Factorization efficiency depends on Argument Threshold.**

of 5, there are very few arguments; everything gets moved from the base store to the provenance store, with associated pointers needed. When the threshold moves to 50, however, a substantial part of provenance records get treated as arguments, and are left untouched in the provenance store. Unfortunately, as the threshold gets larger, there is again a disadvantage since fewer items qualify to be moved from the general store to the provenance store. Depending on the dataset, the Argument Threshold affects runtime and reducibility differently. Reduction of the Karma and PReServ stores performs best in time and space with an Argument threshold of 10. This is a reflection of how often common sources or manipulations are used in each. MiMI utilizes the same sources and manipulations over and over, while the processes used to generate Karma and PReServ do not.

Between 5-50 seems to be a robust range of values for the Argument Threshold, likely not to be too far from optimum. This is a good range in which to set the default value for this parameter, as a rule of thumb, although as shown with MiMI, some twiddling of this knob may be required.

## 6.8 Practitioner's Guide

If only one type of reduction were to be used, we would recommend Argument Factorization. We have already seen that it results in better reduction than the other Factorization techniques. Argument Factorization is the hands down winner for the following reasons:

- It has smaller reduction times due to a reduced number of writes.
- It is order invariant and does not depend upon whether user functions are reflexive, symmetric and transitive.

If further reduction is desired, we suggest the following setups based on data and usage criteria:

**For Best Storage Reductions**

| **Data Characteristics** | **Recommended Tech.** |
|---|---|
| All | Structural Inheritance |
| Most data types have specific process e.g. every name element gets normalized | Predicate Inheritance |

**For Best Query Times**

| **Query Characteristics** | **Recommended Tech.** |
|---|---|
| All | Structural Inheritance [1] |
| Uses provenance as a condition (high or low selectivity) | Argument Factorization |
| Uses provenance as a condition and data has type-specific processes | Predicate Inheritance |

Note that if the data or query contains several characteristics listed above, our techniques can be combined. The combination is synergistic, and will do more together than either alone.

## 7. RELATED WORK

Several real-world applications have generated and used provenance information [1, 3, 4, 14, 16, 19, 25, 27]. In previous studies [7, 20, 31], the major focus has been on creating a provenance record quickly enough to not substantially slow down the experimental application; the resulting provenance size was of less importance. Only Chimera [14] proposes a method for scaling to ever larger provenance records, and relies heavily on distributed systems and virtual provenance. Workflows systems [17, 24] can generate large amounts of provenance. Some workflow systems [3, 29] are also trying to reduce provenance size. These systems effectively normalize provenance data to minimize repetitions of manipulation information across provenance runs. In such cases, our Factorization Algorithms would not provide much benefit. However, reduction is possible using the Inheritance Algorithms. Additionally, if data is versioned, as in [8], our provenance store reductions can still be applied; distinct versions of a data item will point to different records in the provenance store.

The Factorization Algorithms are similar to work in workflow specification from process logs [11, 33], which attempts to create an accurate workflow, with an eye to processes, but our work attempts to understand and reduce the size of arguments found in provenance files. Compiler optimization [10] has also similarities to the provenance reduction studied here.

XML compression [23] creates a smaller store than the reduction provided in this work. However, the XML compression systems do not result in a store that can be queried with an uncompressed dataset via a standard query language. While XGRIND [32] does support exact and substring querying of the compressed store, it does not support joins and thus cannot build relationships among data and provenance elements; specifically, there is a lack of support for value or structural joins between provenance pointers and the provenance store. Luckily, these compression techniques can be further applied to the reduced provenance store we create.

## 8. CONCLUSION

Provenance storage is becoming essential for scientific research, but the size of provenance can overwhelm the size of data, in most cases. In this paper we presented a strategy to reduce provenance storage size. Specifically, we developed a family of Factorization algorithms, as well as algorithms that exploit Predicate and Structural Inheritance. We described how to apply all three techniques in tandem to the same data set.

Our experimental assessment showed that our strategy can reduce the size of provenance by up to a factor of 20. The reduction algorithm scales linearly with provenance store size. Provenance remains queriable, even after reduction using our strategy. In fact, some classes of queries run faster on the reduced store. Also, our reduction strategy is orthogonal to traditional text or XML compression: both can be applied in tandem to get additional reduction, if queriability is not a requirement.

Our work has assumed a generic enough provenance model that many existing systems could easily be mapped to. We are currently in conversations with owners of large scientific data sets to have them adopt our provenance reduction techniques on their production data.

[1]Requires availability of an iterator to trace ancestors.

## 9. ACKNOWLEDGMENTS

Our many thanks to Yogesh Simmhan for his generous gift of the PReServ and Karma provenance sets generated in [31]. This work was supported in part by NSF grant number IIS 0741620 and by NIH grant number U54 DA021519.

## 10. REFERENCES

[1] James Annis, Yong Zhao, Jens-S. Vöckler, Michael Wilde, Steve Kent, and Ian T. Foster. Applying chimera virtual data concepts to cluster finding in the sloan sky survey. In *SC*, pages 1–14, 2002.

[2] Gary Bader, D Betel, and Christopher W.V. Hogue. BIND: the biomolecule interaction network database. *Nucleic Acids Research*, 31(1):248–250, 2003.

[3] Roger S. Barga and Luciano A. Digiampietri. Automatic capture and efficient storage of escience experiment provenance. In *Concurrency and Computation: Practice and Experience*, 2007.

[4] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB Seoul, Korea*, pages 953–964, 2006.

[5] Deepavali Bhagwat et al. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[6] Shawn Bowers, Timothy McPhillips, Martin Wu, and Bertram Ludäscher. Project histories: Managing data provenance across collection-oriented scientific workflow runs. In *DILS*, pages 27–29, 2007.

[7] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *ACM SIGMOD*, pages 539–550, June 2006.

[8] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. In *ACM SIGMOD*, pages 1–12, June 2002.

[9] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and Where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[10] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[11] Shirley Cohen, Sarah Cohen Boulakia, and Susan Davidson. Towards a model of scientific workflows and user views. In *DILS*, pages 264–279, 2006.

[12] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. In *Proceedings of the 27th VLDB Conference, Roma, Italy*, pages 41–58, 2001.

[13] Jens Ernst, William Evans, Christopher Fraser, Steven Lucco, and Todd Proebsting. Code compression. In *ACM SIGPLAN*, pages 358–365, 1997.

[14] Ian Foster, Jens Vockler, Michael Eilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *International Conference on Scientific and Statistical Database Management*, pages 37–46, July 2002.

[15] Ian Foster, Jens Vockler, M Wilde, and Yong Zhao. The virtual data grid: a new model and architecture for data-intensive collaboration. In *CIDR*, 2003.

[16] James Frew and R Bose. Earth system science workbench: A data management infrastructure for earth science products. In *SSDBM*, pages 180–189, 2001.

[17] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):26–34, 2007.

[18] Roberto Grossi. On finding common subtrees. *Theor. Comput. Sci.*, 108(2):345–356, 1993.

[19] Paul Groth et al. Recording and using provenance in a protein compressibility experiment. In *HPDC*, 2005.

[20] Paul Groth, Simon Miles, and Luc Moreau. PReServ: Provenance recording for services. In *Proceedings of the UK OST e-Science second All Hands Meeting 2005 (AHM'05)*, 2005.

[21] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, et al. Timber: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[22] Magesh Jayapandian, Adriane Chapman, V.Glenn Tarcea, Cong Yu, Aaron Elkiss, Angela Ianni, Bin Liu, Arnab Nandi, Carlos Santos, Philip Andrews, Brian Athey, David States, and H.V. Jagadish. Michigan Molecular Interactions (MiMI): Putting the jigsaw puzzle together. *Nucleic Acids Research*, pages D566–D571, Jan 2007.

[23] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In *ACM Sigmod International Conference on Management of Data*, 2000.

[24] Luc Moreau, Bertram Ludäscher, et al. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, 2007. http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge .

[25] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, pages 43–56, 2006.

[26] Open provenance model. http://twiki.ipaw.info/bin/view/Challenge/OPM, 2008.

[27] Carmen Pancerella, John Hewson, Wendy Koegler, et al. Metadata in the collaboratory for multi-scale chemical science. In *Dublin Core Conference*, 2003.

[28] S Peri et al. Development of human protein reference database as an initial platform for approaching systems biology in humans. *Genome Research*, 13:2363–2371, 2003.

[29] Carlos Eduardo Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Cláudio T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.

[30] Amit Sheth. Metadata storage in the active semantic electronic medical record system (ASEMR) deployed at the athens heart center. personal communication, Oct 2006.

[31] Yogesh Simmhan, Beth Plale, and Dennis Gannon. A framework for collecting provenance in data-centric scientific workflows. In *ICWS*, 2006.

[32] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, pages 225–234, 2002.

[33] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[34] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *International Conference of Data Engineering*, pages 97–102, 1997.