Provenance Management in Curated Databases

Peter Buneman
University of Edinburgh
Edinburgh, UK
opb@inf.ed.ac.uk

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

James Cheney
University of Edinburgh
Edinburgh, UK
jcheney@inf.ed.ac.uk

ABSTRACT

Curated databases in bioinformatics and other disciplines are the result of a great deal of manual annotation, correction and transfer of data from other sources. Provenance information concerning the creation, attribution, or version history of such data is crucial for assessing its integrity and scientific value. General purpose database systems provide little support for tracking provenance, especially when data moves among databases. This paper investigates general-purpose techniques for recording provenance for data that is copied among databases. We describe an approach in which we track the user's actions while browsing source databases and copying data into a curated database, in order to record the user's actions in a convenient, queryable form. We present an implementation of this technique and use it to evaluate the feasibility of database support for provenance management. Our experiments show that although the overhead of a naïve approach is fairly high, it can be decreased to an acceptable level using simple optimizations.

1. INTRODUCTION

Modern science is becoming increasingly dependent on databases. This poses new challenges for database technology, many of them to do with scale and distributed processing [13]. However there are other issues concerned with the preservation of the "scientific record" – how and from where information was obtained. These issues are particularly important as database technology is employed not just to provide access to source data, but also to the derived knowledge of scientists who have interpreted the data. Many scientists believe that *provenance*, or metadata describing creation, recording, ownership, processing, or version history, is essential for assessing the value of such data. However, provenance management is not well understood; there are few guidelines concerning what information should be retained and how it should be managed. Current database technology provides little assistance for managing provenance.

In this paper we study the problem of tracking provenance of scientific data in *curated databases*, databases constructed by the "sweat of the brow" of scientists who manually assimilate information from several sources. First, it is important to understand the working practices and values of the scientists who maintain and use such databases.

1.1 Curated Databases

There are several hundred public-domain databases in the field of molecular biology [12]. Few contain raw experimental data; most represent an investment of a substantial amount of effort by individuals who have organized, interpreted or re-interpreted, and annotated data from other sources. The Uniprot [23] consortium lists upwards of seventy scientists, variously called curators or annotators, whose job it is to add to or correct the reference databases published by the consortium. At the other end of the scale there are relatively small databases managed by a single individual, such as the Nuclear Protein Database [9]. These databases are highly valued and have, in some cases, replaced paper publication as the medium of communication. Such databases are not confined to biology; they are also being developed in areas such as astronomy and geology. Reference manuals, dictionaries and gazetteers that have recently moved from paper publication to electronic dissemination are also examples of curated databases.

One of the characteristics of curated databases is that much of their content has been derived or copied from other sources, often other curated databases. Most curators believe that additional record keeping is needed to record where the data comes from – its provenance. There has been some examination [2, 8, 16, 22, 24] of provenance issues in data warehouses; that is, views of some underlying collection of data. But curated databases are not warehouses: they are manually constructed by highly skilled scientists; they are not computed automatically from existing data sets; they are not views.

1.1.1 Example

A molecular biologist is interested in how age and cholesterol efflux affect cholesterol levels and coronary artery disease. She keeps a simple database of proteins which may play a role in these systems; this database could be anything from a flat text or XML file to a full RDBMS. One day, while browsing abstracts of recent publications, she discovers some interesting proteins on SwissProt, and copies the records from a SwissProt web page into her database (Figure 1(a)). She then (Figure 1(b)) fixes the new entries so that the PTM (post-translational modification) found in SwissProt is not confused with PTMs in her database found from other sites. She also (Figure 1(c)) copies some publication details from Online Mendelian Inheritance in Man (OMIM) and some other related data from NCBI. Finally (Figure 1(d)), she notices a mistake in a PubMed publication number and corrects it. This manual curation process is repeated many times as the researcher conducts her investigation.

One year later, when reviewing her information, she finds a dis-

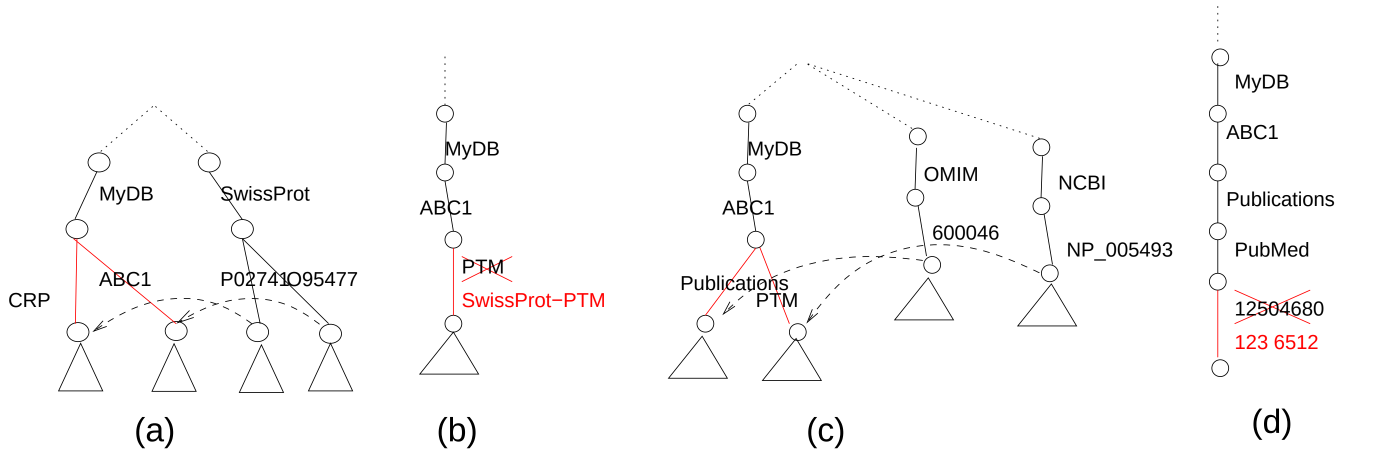

Figure 1: A biological database curation example. Dashed lines represent provenance links.

crepancy between two PTMs and the conditions under which they are found. Unfortunately, she cannot remember where the anomalous data came from, so cannot trace it to the source to resolve the conflict. Moreover, the databases from which the data was copied have changed; searching for the same data no longer gives the same results. The biologist may have no choice but to discard all of the anomalous data or spend a few hours tracking down the correct values. This would be especially embarrassing if the researcher had already published an article or version of her database based on the now-suspect data.

In some respects, the researcher was better off in the days of paper publication and record keeping, where there are well-defined standards for citation and some confidence that the cited data will not change. To recover these advantages for curated databases, it is necessary to retain provenance information describing the source and version history of the data. In Figure 1, this information is represented by the dashed lines connecting source data to copied data.

The current approach to managing provenance in curated databases is for the database designer to augment the schema with fields to contain provenance data [1, 18] and require curators to add and maintain the provenance information themselves. Such manual bookkeeping is time consuming, error-prone and often incomplete. It should not be necessary. We believe it is imperative to find ways of automating the process.

1.2 The problem

The term "provenance" has been used in a variety of senses in database and scientific computation research. One form of provenance is "workflow" or "coarse-grained" provenance: information describing how derived data has been calculated from raw observations [3, 10, 14, 21]. Workflow provenance is important in scientific computation, but is not a major concern in curated databases. Instead, we focus on "fine-grained" or "dataflow" provenance, which describes how data has moved through a network of databases.

Specifically, we consider the problem of tracking and managing provenance describing the user actions involved in constructing a curated database. This includes recording both local modifications to the database (inserting, deleting, and updating data) and global operations such as copying data from external sources. Because of the large number and variety of scientific databases, a realistic solution to this problem is subject to several constraints. The databases are all maintained independently, so it is (in the short term) unrealistic to expect all of them to adopt a standard for storing and exchanging provenance. A wide variety of data models is in use, and databases have widely varying practices for identifying or locating data. While the databases are not actively uncooperative, they may change silently and past versions may not be archived. Curators employ a wide variety of application programs, computing platforms, etc., including proprietary software that cannot be changed.

In light of these considerations, we believe it is reasonable to restrict attention to a subproblem that is simple enough that some progress can be made, yet which we believe provides benefits in common situations faced by database curators.

1.3 Our approach

In this paper, we propose and evaluate a practical approach to provenance tracking for data copied manually among databases. In our approach, we assume that all of the the user's actions in constructing the target database are captured as a sequence of insert, delete, copy, and paste actions by a provenance-aware application for browsing and editing databases. As the user copies, inserts, or deletes data in her local database T, provenance links are stored in an auxiliary provenance database P. These links relate data locations in T with locations in previous versions of T or in external source databases S. They can be used after the fact to review the process used to construct the data in T; in addition, if T is also being archived, the provenance links can provide further detail about how each version of T relates to the next. In order to ensure the consistency of the target database and its provenance record, it is essential that the target database and provenance record are writable only via high-level interfaces that track provenance.

The architecture is summarized in Figure 2. The new components are shaded, and the existing (and unchangeable) components are unshaded. The shaded triangles indicate *wrappers* mapping $S_1, \ldots, S_n, T$ to an XML view; the database P stores provenance information describing the updates performed by the editor. Alternatively, provenance information could be stored as annotations alongside data in T; however, this would require changing the structure of T. The only requirement we make is that there is a way to address each data element. We shall describe this in more detail shortly.

It is important to note that we propose using XML only as an

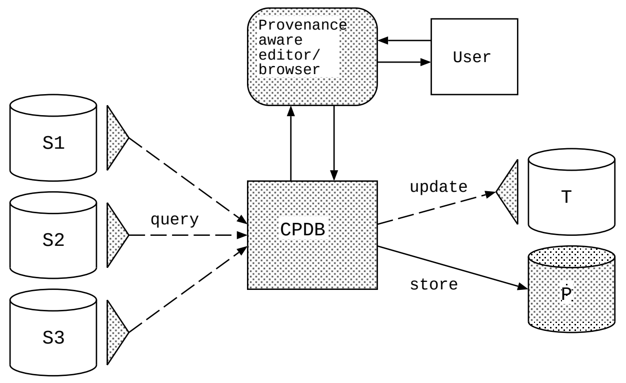

Figure 2: Provenance architecture

abstraction for exchanging and locating data in databases. Our approach does *not* require the underlying databases to store XML. Instead, source and target databases can be relational or XML DBMSs, or consist of files stored in filesystems or Web sites; all are common forms of scientific databases.

When provenance information is tracked manually or by a custom-built system, the user or designer typically decides what provenance information to record on a case-by-case basis. In contrast, our system records everything. The obvious concern is that the processing and storage costs for doing this could be unacceptably high. The main contribution of this paper is to show how such fine-grained provenance information can be tracked, stored, and queried efficiently.

We have implemented our approach and experimented with a number of ways of storing and querying provenance information, including a naïve approach and several more sophisticated techniques. Our results demonstrate that the processing overhead of the naïve approach is fairly high; it can increase the time to process each update by 28%, and the amount of provenance information stored is proportional to the size of the changed data. In addition, we also investigated the impact of two optimization techniques: *transactional* and *hierarchical* provenance management. Together, these optimizations typically reduce the added processing cost of provenance tracking to less than 5–10% per operation and reduce the storage cost by a factor of 5–7 relative to the naïve approach; moreover, the storage overhead is bounded by the lesser of the number of update operations and the amount of data touched. In addition, typical provenance queries can be executed more efficiently on such provenance records. We believe that these results make a compelling argument for the feasibility of our approach to provenance management.

The structure of the rest of this paper is as follows. Section 2 presents the conceptual foundation of our approach to provenance tracking. Section 3 presents the implementation of CPDB, an instance of our approach that uses MySQL and the Timber XML database [15]. In Section 4, we present and analyze the experimental results. We discuss other work on provenance and related areas such as logging, availability, schema evolution, and archiving in Section 5; Sections 6 and 7 discuss future work and conclude.

2. MANUAL UPDATES AND PROVENANCE

In order to discuss provenance we need to be able to describe *where a piece of data comes from*; that is, we need to have a means for describing the location of any data element. We make two assumptions about the data, which are already used in file synchronization [11] and database archiving [4] and appear to hold for a wide variety of scientific and other databases. The first is that the database can be viewed as a tree; the second is that the edges of that tree can be labeled in such a way that a given sequence of labels occurs on at most one path from the root and therefore identifies at most one data element. Traditional hierarchical file systems are a well-known example of this kind of structure. Relational databases also can be described hierarchically. For instance, the data values in a relational database can be addressed using four-level paths where $DB/R/tid/F$ addresses the field value F in the tuple with identifier or key tid in table R of database DB. Scientific databases already use paths such as $\mathsf{SwissProt/Release\{20\}/Q01780}$ to identify a specific entry, and this can be concatenated with a path such as $\mathsf{Citation\{3\}/Title}$ to identify a data element. XML data can be addressed by adding key information [4] or XPath/XPointer expressions.

Formally, we let Σ be a set of labels, and consider *paths* $p \in \Sigma^*$ as addresses of data in trees. The trees t we consider are unordered and store data values from some domain D only at the leaves. Such trees are written as $\{a_1 : v_1, \ldots, a_n : v_n\}$, where v_i is either a subtree or data value. We write $t.p$ for the subtree of t rooted at location p.

We model the the user's actions with a basic update language whose atomic update operations are of the form

$$u \quad ::= \quad \mathsf{ins}\ \{a : v\}\ \mathsf{into}\ p \mid \mathsf{del}\ a\ \mathsf{from}\ p \mid \mathsf{copy}\ q\ \mathsf{into}\ p$$

The insert operation inserts an edge labeled a with value v into the subtree at p; v can be either the empty tree or a data value. The delete operation deletes an edge and its subtree. The copy operation replaces the subtree at p with a copy of the subtree at location q. We write sequences U of atomic updates as $u_1; \ldots; u_n$. Using $[\![U]\!]$ for the function on trees induced by the update sequence U, we can give semantics to the operations is as follows.

$$\begin{array}{rcl} [\![\mathsf{ins}\ \{a : v\}\ \mathsf{into}\ p]\!](t) & = & t[p := (t.p \uplus \{a : v\})] \\ [\![\mathsf{del}\ a\ \mathsf{from}\ p]\!](t) & = & t[p := (t.p - a)] \\ [\![\mathsf{copy}\ q\ \mathsf{into}\ p]\!](t) & = & t[p := t.q] \\ [\![U; U']\!](t) & = & [\![U']\!]([\![U]\!](t)) \end{array}$$

Here, $t \uplus t'$ denotes the tree t with subtree t' added; this fails if there are any shared edge names at the top level of t and u; $t - a$ denotes the result of deleting the node labeled a, failing if no such node exists; and $t[p := t']$ denotes the result of replacing the subtree of t at p by u, failing if path p is not present in t. Insertions, copies, and deletes can only be performed in a subtree of the target database T.

As an example, consider the update operations in Figure 3. These operations copy some records from S_1 and S_2, then modify some of the field values. The result of executing this update operation on database T with source databases S_1, S_2 is shown in Figure 4. The initial version of the target database is labeled T, while the version after the transaction is labeled T'.

2.1 Provenance tracking

Figure 4 depicts *provenance links* (dashed lines) that connect copied data in the target database with source data. Of course, these links are not visible in the actual result of the update. In our approach, these links are stored "on the side" in an auxiliary table $\mathsf{Prov}(\underline{Tid}, Op, \underline{Loc}, Src)$, where Tid is a sequence number for the transaction that made the corresponding change; Op is one of I (insert), C (copy), or D (delete); Loc is the location of the affected data; and Src is the source location (for a copy). The Src field is ignored for inserts and deletes. Note that $\{Tid, Loc\}$ forms a key for Prov; that is, for each transaction, each location has either been inserted, deleted, or copied from somewhere in the input.

```
(1)  delete  c5         from T;
(2)  copy    S1/a1/y    into T/c1/y;
(3)  insert  {c2 : {}}  into T;
(4)  copy    S1/a2      into T/c2;
(5)  insert  {y : {}}   into T/c2;
(6)  copy    S2/b3/y    into T/c2/y;
(7)  copy    S1/a3      into T/c3;
(8)  insert  {c4 : {}}  into T;
(9)  copy    S2/b2      into T/c4;
(10) insert  {y : 12}   into T/c4;
```

Figure 3: An example copy-paste update operation.

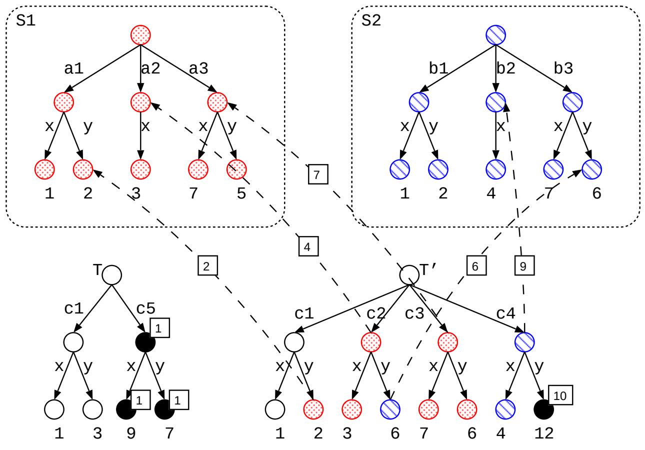

Figure 4: An example of executing the update in Figure 3. The upper two trees S_1, S_2 are XML views of source databases; the bottom trees T, T' are XML views of part of the target database at the beginning and end of the transaction. White nodes are unchanged; black nodes represent inserted or deleted nodes; other shadings indicate whether the node came from S_1 or S_2. Dashed lines indicate provenance links. Boxed numbers indicate the relevant copy-paste operation in Figure 3. Additional provenance links can be inferred from context.

Thus, Tid and Loc are natural candidates for indexing. Additional information about each transaction, such as commit time and user identity, can be stored in a separate table with key Tid.

We now examine several ways of storing provenance information.

2.1.1 Naïve provenance

The most straightforward method is to store one provenance record for each copied, inserted, or deleted node. In addition, each update operation is treated as a separate transaction. This technique may be wasteful in terms of space, because it introduces one provenance record for every node inserted, deleted, or copied throughout the update. However, it retains the maximum possible information about the user's actions. In fact, the exact update operation describing the user's sequence of actions can be recovered from the provenance table.

2.1.2 Transactional provenance

The second method is to assume the updated actions are grouped into transactions larger than a single operation, and to store only

(a) Prov

Tid	Op	Loc	Src
121	D	T/c_5	$\perp$
121	D	$T/c_5/x$	$\perp$
121	D	$T/c_5/y$	$\perp$
122	C	$T/c_1/y$	$S_1/a_1/y$
123	I	T/c_2	$\perp$
124	C	T/c_2	S_1/a_2
124	C	$T/c_2/x$	$S_1/a_2/x$
125	I	$T/c_2/y$	$\perp$
126	C	$T/c_2/y$	$S_2/b_3/y$
127	C	T/c_3	S_1/a_3
127	C	$T/c_3/x$	$S_1/a_3/x$
127	C	$T/c_3/y$	$S_1/a_3/y$
128	I	T/c_4	$\perp$
129	C	T/c_4	S_2/b_2
129	C	$T/c_4/x$	$S_2/b_2/x$
130	I	$T/c_4/y$	$\perp$

(b) Prov

Tid	Op	Loc	Src
121	D	T/c_5	$\perp$
121	D	$T/c_5/x$	$\perp$
121	D	$T/c_5/y$	$\perp$
121	C	$T/c_1/y$	$S_1/a_1/y$
121	C	T/c_2	S_1/a_2
121	C	$T/c_2/x$	$S_1/a_2/x$
121	C	$T/c_2/y$	$S_2/b_3/y$
121	C	T/c_3	S_1/a_3
121	C	$T/c_3/x$	$S_1/a_3/x$
121	C	$T/c_3/y$	$S_1/a_3/y$
121	C	T/c_4	S_2/b_2
121	C	$T/c_4/x$	$S_2/b_2/x$
121	I	$T/c_4/y$	$\perp$

(c) HProv

Tid	Op	Loc	Src
121	D	T/c_5	$\perp$
122	C	$T/c_1/y$	$S_1/a_1/y$
123	I	T/c_2	$\perp$
124	C	T/c_2	S_1/a_2
125	I	$T/c_2/y$	$\perp$
126	C	$T/c_2/y$	$S_2/b_3/y$
127	C	T/c_3	S_1/a_3
128	I	T/c_4	$\perp$
129	C	T/c_4	S_2/b_2
130	I	$T/c_4/y$	$\perp$

(d) HProv

Tid	Op	Loc	Src
121	D	T/c_5	$\perp$
121	C	$T/c_1/y$	$S_1/a_1/y$
121	C	T/c_2	S_1/a_2
121	C	$T/c_2/y$	$S_2/b_3/y$
121	C	T/c_3	S_1/a_3
121	C	T/c_4	S_2/b_2
121	I	$T/c_4/y$	$\perp$

Figure 5: The provenance tables for the update operation of Figure 3. (a) One transaction per line. (b) Entire update as one transaction. (c) Hierarchical version of (a). (d) Hierarchical version of (b).

provenance links describing the net changes resulting from a transaction. For example, if the user copies data from S_1, then on further reflection deletes it and uses data from S_2 instead, and finally commits, this has the same effect on provenance as if the user had only copied the data from S_2. Thus, details about intermediate states or temporary data storage in between "official" database versions are not retained. Transactional provenance may be less precise than the naïve approach, because information about intermediate states of the database is discarded. However, the decision when to commit is in the hands of the user; frequent commits can be used to record important intermediate states.

The storage cost for the provenance of a transaction is proportional to the number of nodes touched in the input and output of the transaction. That is, the number of transactional provenance records produced by an update transaction t is $i + d + c$, where i is the number of inserted nodes in the output, d is the number of nodes deleted from the input, and c is the number of copied nodes in the output.

2.1.3 Hierarchical provenance

Whether or not transactional provenance is used, much of the provenance information tends to be redundant (see Figure 5(a,b)), since in many cases the annotation of a child node can be inferred from its parent's annotation. Accordingly, we consider a second technique, called *hierarchical provenance*. The key observation is that we do not need to store all of the provenance links explicitly, because the provenance of a child of a copied node can often be inferred from its parent's provenance using a simple rule. Thus, in hierarchical provenance we store only the provenance links that cannot be so inferred. These non-inferable links correspond to the provenance links shown in Figure 4. A copy-paste operation copy p into q results in adding only a single record $\mathsf{HProv}(t, \mathsf{C}, q, p)$. Figure 5(c) shows the hierarchical provenance table HProv corresponding to the naïve version of Prov. In this case, the reduced table is about 25% smaller than Prov, but much larger savings are possible when entire records or subtrees are copied with little change.

Unlike transactional provenance, hierarchical provenance does not discard any information and does not require the user to group operations into transactions. We can define the full provenance table as a view of the hierarchical table as follows. If the provenance is specified in HProv, then it is just copied into Prov. Otherwise, the provenance of every target path p/a *not* mentioned in HProv is q/a, provided p was copied from q. If p was inserted, then we assume that p/a was also inserted; that is, children of inserted nodes are assumed to also have been inserted, unless there is a record in HProv indicating otherwise. Deletions are treated similarly. Formally, the full provenance table Prov can be defined in terms of HProv as the following recursive query:

$$\begin{array}{lcl} \mathsf{Infer}(t,p) & \leftarrow & \neg(\exists x, q.\mathsf{HProv}(t,x,p,q)) \\ \mathsf{Prov}(t,op,p,q) & \leftarrow & \mathsf{HProv}(t,op,p,q). \\ \mathsf{Prov}(t,\mathsf{C},p/a,q/a) & \leftarrow & \mathsf{Prov}(t,\mathsf{C},p,q), \mathsf{Infer}(t,p). \\ \mathsf{Prov}(t,\mathsf{I},p/a,\bot) & \leftarrow & \mathsf{Prov}(t,\mathsf{I},p,\bot), \mathsf{Infer}(t,p). \\ \mathsf{Prov}(t,\mathsf{D},p/a,\bot) & \leftarrow & \mathsf{Prov}(t,\mathsf{D},p,\bot), \mathsf{Infer}(t,p). \end{array}$$

We have to use an auxiliary table Infer to identify the nodes that have no explicit provenance in HProv, to ensure that only the provenance of the closest ancestor is used. In our implementation, Prov is calculated from HProv as necessary for paths in T, so this check is unnecessary. It is not difficult to show that an update sequence U can be described by a hierarchical provenance table with $|U|$ entries.

2.1.4 *Transactional-hierarchical provenance*

Finally, we considered the combination of transactional and hierarchical provenance techniques; it is not difficult to combine them. Figure 5(d) shows the transactional-hierarchical provenance of the transaction in Figure 3.

It is also easy to show that the storage of transactional-hierarchical provenance is $i + d + C$, where i and d are defined as in the discussion of transactional provenance and C is the number of *roots* of copied subtrees that appear in the output. This is bounded above by both $|U|$ and $i+d+c$, so transactional-hierarchical provenance may be more concise than either approach alone.

2.2 Provenance queries

How can we use the machinery developed in the previous section to answer some practical questions about data? Consider some simple questions:

Src What transaction first created the data at a location? This is particularly useful in the case of raw data; e.g., who entered your telephone number incorrectly?

Hist What is the sequence of all transactions that copied a node to its current position?

Mod What transactions were responsible for the creation or modification of the subtree under a node?

Hist and Mod provide very different information. A subtree may be copied many times without being modified.

We first define some convenient views of the raw Prov table (which, of course, may also be a view derived from HProv). We define the views $\mathsf{Unch}(t,p)$, $\mathsf{Ins}(t,p)$, $\mathsf{Del}(t,p)$, and $\mathsf{Copy}(t,p,q)$, which intuitively mean "p was unchanged, inserted, deleted, or copied from q during transaction t," respectively.

$$\begin{array}{lcl} \mathsf{Unch}(t,p) & \leftarrow & \neg(\exists x, q.\mathsf{Prov}(t,x,p,q)). \\ \mathsf{Ins}(t,p) & \leftarrow & \mathsf{Prov}(t,\mathsf{I},p,\bot) \\ \mathsf{Del}(t,p) & \leftarrow & \mathsf{Prov}(t,\mathsf{D},p,\bot) \\ \mathsf{Copy}(t,p,q) & \leftarrow & \mathsf{Prov}(t,\mathsf{C},p,q) \end{array}$$

We also consider a node p to "come from" q during transaction t (table $\mathsf{From}(t,p,q)$) provided it was either unchanged (and $p = q$) or p was copied from q.

$$\begin{array}{lcl} \mathsf{From}(t,p,q) & \leftarrow & \mathsf{Copy}(t,p,q) \\ \mathsf{From}(t,p,p) & \leftarrow & \mathsf{Unch}(t,p) \end{array}$$

Next, we define a $\mathsf{Trace}(p,t,q,u)$, which says that the data at location p at the end of transaction t "came from" the data at location q at the end of transaction u.

$$\begin{array}{lcl} \mathsf{Trace}(p,t,p,t). & & \\ \mathsf{Trace}(p,t,q,u) & \leftarrow & \mathsf{Trace}(p,t,r,s), \mathsf{Trace}(r,s,q,u). \\ \mathsf{Trace}(p,t,q,t-1) & \leftarrow & \mathsf{From}(t,p,q). \end{array}$$

Note that Trace is essentially the reflexive, transitive closure of From. Now to define the queries mentioned at the beginning of the section, suppose that t_{now} is the last transaction number in Prov, and define

$$\begin{array}{lcl} \mathsf{Src}(p) & = & \{u \mid \exists q.\mathsf{Trace}(p,t_{now},q,u), \mathsf{Ins}(u,q)\} \\ \mathsf{Hist}(p) & = & \{u \mid \exists q.\mathsf{Trace}(p,t_{now},q,u), \mathsf{Copy}(u,q)\} \\ \mathsf{Mod}(p) & = & \{u \mid \exists q.p \leq q, \mathsf{Trace}(q,t_{now},r,u), \neg\mathsf{Unch}(u,r)\} \end{array}$$

That is, $\mathsf{Src}(p)$ returns the number of the transaction that inserted the node now at p, while $\mathsf{Hist}(p)$ returns all transaction numbers that were involved in copying the data now at p. Finally, $\mathsf{Mod}(p)$ returns all transaction numbers that modified some data under p. This set could then be combined with additional information about transactions to identify all users that modified the subtree at p. Here, $p \leq q$ means p is a prefix of q. Despite the fact that there may be infinitely many paths q extending p, the answer $\mathsf{Mod}(p)$ is still finite, since there are only finitely many transaction identifiers in Prov. Moreover, Mod can be answered using only the data in Prov or HProv; it is not necessary to inspect the target database.

There are many interesting queries that mention both provenance and the raw data. Our system currently does not provide special support for such queries, but they can be written by explicitly constructing paths using string operations. For example, to project the A field out of relation $\mathsf{R}(\underline{Id}, A, B)$ along with its current provenance, we could use the query

$$\mathsf{Q}(x,p_x) \quad \leftarrow \quad \mathsf{R}(k,x,y), \mathsf{From}(t_{now}, \texttt{"R/"}+k+\texttt{"/A"}, p_x)$$

where k, x, p_x, and y are variables and $+$ denotes string concatenation. Such queries are tricky to write by hand, and we are interested in providing advanced support for provenance queries; however, this is future work.

The point of this discussion is to show that provenance mappings relating a sequence of versions of a database can be used to answer a wide variety of queries about the evolution of the data, even without cooperation from source databases. However, if only the target database tracks provenance, the information is necessarily partial. For example, the Src query above cannot tell us anything about data that was copied from elsewhere. Similarly, the Hist and Mod queries stop following the chain of provenance of a piece of data when it exits T. If some source databases do not track provenance and publish it in a consistent form, many queries only have incomplete answers.

Of course, if source databases also store provenance, we can provide more complete answers by combining the provenance information of all of the databases. In addition, there are queries which only make sense if several databases track provenance, such as:

Own What is the history of "ownership" of a piece of data? That is, what sequence of databases contained the previous copies of a node?

It would be extremely useful to be able to provide answers to such queries to scientists who wish to evaluate the quality of data found in scientific databases.

3. IMPLEMENTATION

We have implemented a "copy-paste database" CPDB that tracks the provenance of data copied from external sources to the target database. In order to demonstrate the flexibility of our approach, our system connects several different publicly downloadable databases. We have chosen to use MiMI [18], a biological database of curated datasets, as our target database (T in Figure 2). MiMI is a protein interaction database that runs on Timber [15], a native XML database. We used OrganelleDB [25], a database of protein localization information built on MySQL, as an example of a source database. Since the target database interacts with only one source database at a time, we only experimented with one source database. In addition, the provenance store was implemented as a MySQL table.

3.1 Overview

CPDB permits the user to connect to the external databases, copy source data into the target database, and modify the data to fit the target database's structure. The user's actions are intercepted and the resulting provenance information is recorded in a *provenance store*. Currently, CPDB provides a minimal Web interface for testing purposes, implemented using JavaScript and SOAP. The interface provides tree views of the databases which the user can use to select parts of the data along with buttons that allow the user to insert, delete, copy, and paste selected data. Providing a more user-friendly browsing/editing interface is important, but orthogonal to the data management issues that are our primary concern.

In order to allow the user to select pertinent information from the source and target databases, each database must be wrapped in a way that allows CPDB to extract the appropriate information. This wrapping is essentially the same as a "fully-keyed" XML view of the underlying data. In addition, the target database must also expose particular methods to allow for easy updating. Figure 6 describes the necessary functions that the source and target databases must implement. Essentially, the source and target databases must provide methods that map tree paths to the database's native data; in addition, the target database must be able to translate updates to the tree to updates to its internal data.

This approach does *not* require that any of the source or target databases represent data internally as XML. Any underlying data model for which path addresses make sense can be used. Also, the databases need not expose all of their data. Instead, it is up to the databases' administrators how much data to expose for copying or updating. In many cases, the data in scientific databases consists of a "catalog" relation that contains all the raw data, together with supporting cross-reference tables. Typically, it is only this catalog that would need to be made available by a source database. In such cases, it may be possible to write wrappers that are completely

SourceDB	
treeFromDB()	Returns a tree, with unique identifi ers, populated from the data. The SourceDB is responsible for determining how the data fi ts in the tree, e.g. mapping a relational database to tree format.
copyNode()	Returns a list of nodes that a user has copied. If the user copies a leaf node, the list is size 1. Otherwise, each node in the subtree of the selected node is contained in the list. Each node contains the identifying path and data value.
TargetDB	
addNode (String nodename)	Inserts a new, empty node with name=nodename in the target db according to the database's mapping from a tree to native format.
deleteNode()	Deletes the specifi ed node from the target database.
pasteNode(Node X)	Insert node X as a child of the specifi ed node according to the tree to database schema mapping.

Figure 6: Wrappers for Source and Target Databases

independent of (and do not require changing) the source database. However, this has to be done on a case by case basis.

3.2 Implementation of provenance tracking

Given wrapped source and target databases, CPDB maintains a provenance store that allows us to track any changes made to the target database incorporating data from the sources. To this end, during a copy-paste transaction, we write the data values to the target database, and write the provenance information to the provenance store. A user may specify any of the storage operations discussed in the previous section. In this section, we discuss how the implementations of provenance tracking and the Src, Hist, and Mod provenance queries differ from the idealized forms presented in Section 2.

3.2.1 Naïve provenance

The implementation of the naïve approach is a straightforward process of recording target and source information for every transaction that affects the target database. Whenever an insert, delete, or copy operation is performed, the corresponding tracking function $trackInsert$, $trackDelete$, $trackPaste$ is called with the transaction identifier and applicable source and target paths. These operations simply add the corresponding records to the provenance store. Note that for a paste operation, we add one record per node in the copied subtree.

3.2.2 Transactional provenance

In transactional provenance, the user decides how to segment the sequence of update operations into transactions. When the user decides to end a transaction and commit its changes, CPDB stores the provenance links connecting the current version with its predecessor, and the current version becomes the next reference copy of the database, to which future provenance links will refer. Only provenance links of data actually present in the output of a transaction are stored; no links corresponding to temporary data deleted or overwritten by the transaction are stored.

To support this behavior, the transactional provenance implementation maintains an active list, $provlist$, of provenance links that will be added to the provenance store when the user commits. When an atomic update is performed, the provenance store is unaffected, but any resulting provenance links are added to the list. Conversely, in the case of a copy or delete, any provenance links on the list corresponding to overwritten or deleted data are removed. At the time of the commit, the $commit()$ function is called, which writes the provenance of all items in the active list to the provenance store.

3.2.3 *Hierarchical Provenance*

In the hierarchical provenance storage method, we store at most one record per operation, and in particular, for a copy, we only store the record connecting the root of the copied tree to the root of the source.

3.2.4 *Hierarchical Transactional Provenance*

Combining the hierarchical and transactional provenance is straightforward; all we need to do is to maintain hierarchical provenance instead of naïve provenance records in $provlist$. One subtle issue is that when several operations are performed in one transaction, some of the resulting hierarchical provenance links could be redundant; for example, this happens if we copy S/a to T/a and then copy $S/a/b$ to $T/a/b$. It is possible to check for and remove such redundant links prior to committing $provlist$. However, such redundancy is unusual, so this extra processing appears not to be worthwhile in most cases.

3.3 Provenance Queries

We implemented the provenance queries Src, Mod, and Hist as programs that issue several basic queries, due to lack of support for the kind of recursion needed by the Trace query. For naïve and transactional provenance, we can directly query the provenance store. For hierarchical provenance, the provenance store corresponds to the HProv relation. Instead of building a view containing the full provenance relation, we query the provenance store directly and compute the appropriate provenance links on-the-fly. All versions of the queries are implemented as stored procedures written in Java running in MySQL.

4. EVALUATION

User requirements for provenance vary widely and are constantly evolving, so evaluating any approach to provenance is challenging. In addition, evaluating an approach to provenance tracking for manual curation is challenging because it is difficult to collect realistic manual update sequences. In order to evaluate the feasibility of our approach, we therefore chose to use random sequences of copy-paste operations to simulate worst-case behavior. We considered update sequences of 3500 and 14,000 steps; observations of real practices by database curators indicate that 14000 steps could account for roughly 6 months of changes produced by 4 curators.

We believe that the most important factors affecting the feasibility of provenance management are the costs of tracking and storing provenance, not querying provenance. Although the purpose of provenance tracking is to make it possible to answer provenance queries after the fact, such queries are rare compared to queries on the raw data. Thus, our experiments focused primarily on the storage and processing requirements of provenance tracking for the different approaches. We also performed experiments measuring the effect of the different approaches on provenance query processing; full exploration of provenance query optimization and database tuning is left for future work.

4.1 Experimental setup

The evaluation of CPDB was performed on a Dell workstation with Pentium 4 CPU at 2GHz with 640MB RAM and 74.4GB disk space running Windows XP. The target database was a 27.3MB copy of MiMI stored in Timber, and the source database was 6MB of data from OrganelleDB stored in MySQL. The provenance information was stored separately in MySQL. We used Timber version 1.1 and MySQL version 4.1.12a-nt via TCP/IP. CPDB was implemented as a Java application that communicates with MySQL via JDBC and Timber using SOAP.

We performed five sets of experiments to measure the relative performance of the naïve (N), transactional (T), hierarchical (H), and hierarchical-transactional (HT) provenance storage methods. Table 1 summarizes the experiments we report, including a description of the fixed and varying parameters, and listing the figures containing experimental results. We used six patterns of update operations, summarized in Table 2. The first five are random sequences of adds, deletes, and copies in various proportions. The copies were all of subtrees of size four (a parent with three children) from OrganelleDB to MiMI. The real update consisted of a regular pattern of copies, deletes, and inserts simulating the effect of a bulk update on MiMI that could be performed via a standard XQuery statement using XPath. It repeatedly copies a subtree into the target, then inserts three elements under the subtree root and deletes three existing subtree elements. We also used variations of the mix dataset that exhibited different deletion patterns, shown in Table 3.

In the first set of experiments we ran 3500-step updates on each of the first five update patterns using each storage method. For the transactional approaches, commits were performed after every five updates. In each case, we measured the amount of time needed for provenance manipulation, interaction with the target database, and interaction with the provenance database. We also measured the total size of the provenance store and target database (both in number of rows and in real storage terms) at the end of the transaction. Efficiency considerations precluded measuring the size of the provenance store or target database after each operation.

In the second experiment, we ran 14,000-step versions of the real and mix updates using all four provenance methods, with the same experimental methodology as for the 3500-step updates. These experiments were intended to determine how our techniques scale as larger numbers of realistic user actions are performed, so we did not run the less realistic add, delete, or copy update patterns of this length.

Figure 7 shows the total provenance storage in rows needed for each method and each run for the 3500-step updates. The real storage sizes in bytes display the same trends (each row requires between 100 and 200 bytes), so we omit this data. Figure 8 shows the total provenance storage in rows needed for each of the 14,000-step runs. Numbers at the top of each bar show the physical sizes of the tables. Figure 9 shows the average time for target database interaction, and average time per add, delete, copy, or commit operation for the 14,000-mix run. These results accurately reflect observed provenance processing times in all the other experiments, so we omit this data. In order to determine how expensive provenance tracking is per add, delete, or copy operation, we also calculated the average time for dataset manipulation by operation type; Figure 10 shows the overhead of provenance tracking for each operation as a percentage of base dataset manipulation time.

In the third experiment, we measured the effects of deletes on provenance storage. We performed five different versions of the 14,000-mix update with varying deletion patterns. These deletion patterns may not be representative of common user behavior, but demonstrate the storage performance of the various methods under different conditions. Figure 11 shows the results of this experiment. We plot two columns per provenance method, one (labeled “ac”)

Table 1: Summary of experiments

	Upd. Length	Trans. Length	Update Pattern	Prov. Method	Measured	Figures
1	3500	5	add, delete, copy, ac-mix, mix	N, H, T, HT	space	7
2	14000	5	mix, real	N, H, T, HT	space, time	8, 9, 10
3	14000	5	del-random, del-add, del-mix, del-copy, del-real	N, H, T, HT	space	11
4	3500	7, 100, 500, 1000	real	HT	time	12
5	14000	5	real	N, H, T, HT	query time	13

Table 2: Update patterns

add	All random adds
delete	All random deletes
copy	All random copies
ac-mix	Equal mix of random adds and copies
mix	Equal mix of random adds, deletes, copies
real	Copy one subtree, add 3 nodes, delete 3 nodes

Table 3: Deletion patterns

del-random	Paths deleted at random
del-add	All added paths deleted
del-copy	Only copies deleted
del-mix	50–50 mix of adds and copies deleted
del-real	3 nodes from copied subtree deleted

showing the provenance table size when only the adds and copies are performed, the other (labeled "acd") showing the size when the deletes are also performed.

The fourth experiment measured the effect of transaction length on provenance processing time. It consisted of running the 3500-real update for the hierarchical-transactional method with transaction lengths 7, 100, 500, and 1000. We measured the processing time required for each operation. Figure 12 summarizes the results of this experiment; it shows the average time needed for each add, delete, copy, and commit for each run. Also, the "amortized" data series shows the average time per operation with commit time amortized over all operations.

Finally, the fifth experiment measured the cost of answering some typical provenance queries. For each storage method, we measured the average query processing time for $getSrc$, $getMod$, $getHist$ queries of random locations run at the end of a 14,000-real run. Figure 13 shows the results. Error bars indicate the typical ranges of response times. No indexing was performed on the provenance relation, so these query times represent worst-case behavior.

4.2 Analysis

As can be seen in Figures 7 and 8, either a hierarchical or transactional strategy can provide substantial space savings. Figure 7 shows how the storage methods perform for different types of actions. Perhaps unsurprisingly, inserts and deletes are handled essentially the same by all methods. Only copy operations really stress the system. The naïve and transactional approaches store four provenance records per copy (recall that all copies are of subtrees of size four), whereas the hierarchical techniques store only one such record per copy. The hierarchical-transactional technique provides the most efficient storage overall. The results in Figure 8 confirm these trends for longer sequences of updates.

Figure 9 shows the time spent on storing provenance information for all the techniques. For comparison, the average dataset

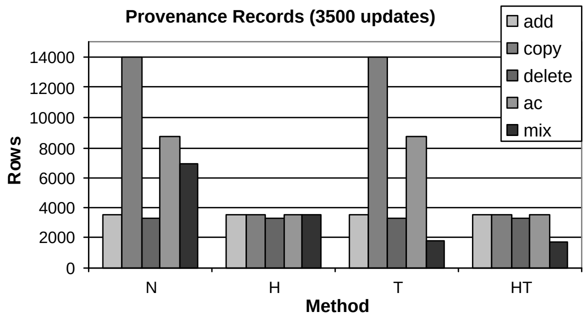

Figure 7: Number of entries in the provenance store after a variety of update patterns of length 3500.

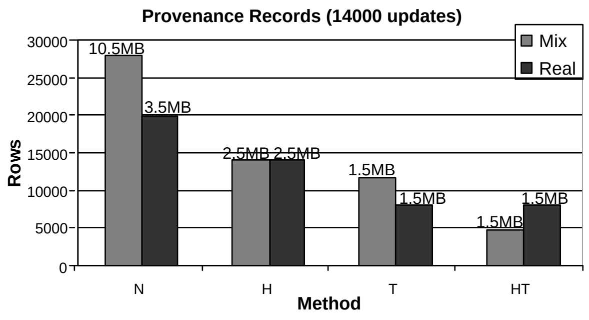

Figure 8: Number of entries in the provenance store after mix and real update patterns of length 14,000. The number at the top of each bar shows the physical size of the table.

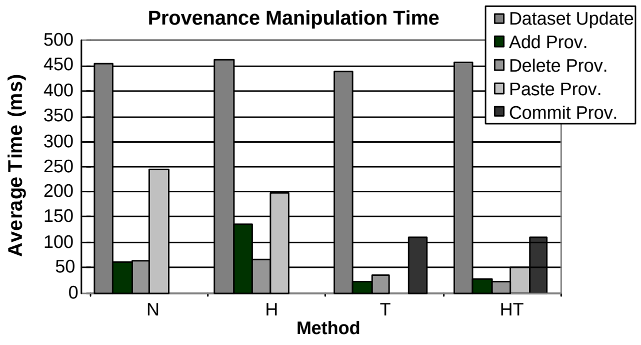

Figure 9: The average amount of time for target database processing and for add, delete, copy, and commit operations on the provenance store during a 14000-mix update.

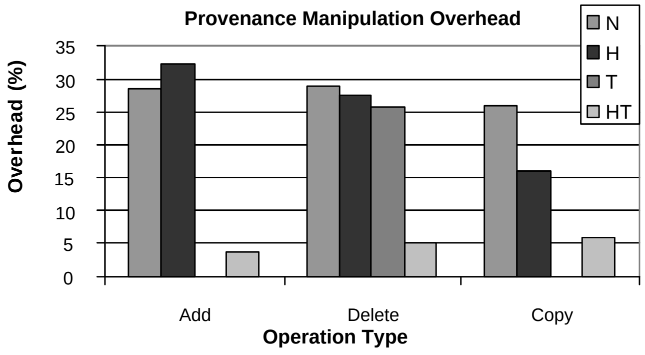

Figure 10: The overhead of provenance tracking per operation, as a percentage of the time to perform each basic operation.

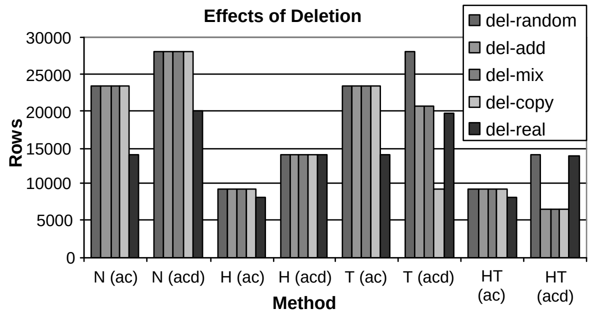

Figure 11: The effect of deletion on the provenance store. The notation (ac) indicates provenance table size when only add and copy operations are performed while (acd) includes deletes.

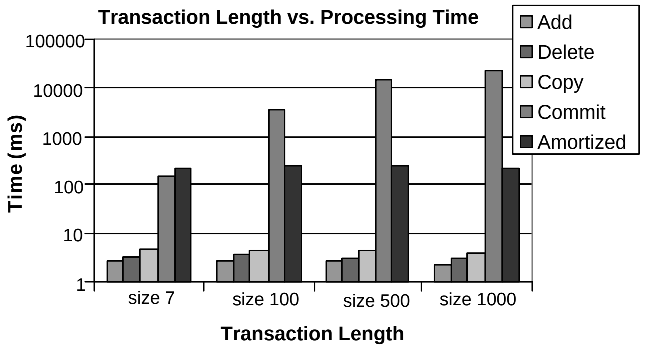

Figure 12: The effect of transaction size on provenance processing time.

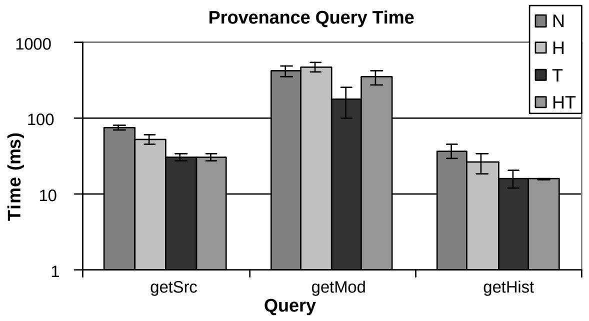

Figure 13: The time needed to perform basic provenance queries.

processing time and average commit times are shown as well. Note that the time for copying in transactional provenance is not zero; it is just close to zero because copies do not involve interaction with the provenance store in transactional provenance. Figure 10 depicts the average overhead of provenance processing per individual add, delete, or copy operation. For naïve storage, the add, delete and copy operations require less than 30% of the processing time needed for interaction with the target database. Although hierarchical provenance is much faster for copies, it requires more time to process inserts. (Deletes are unaffected because hierarchical provenance treats deletes exactly as naïve provenance does.) More time is needed because we must first query the provenance database to determine whether to add the provenance record. Transactional provenance, on the other hand, is much more responsive. Inserts and copies run essentially instantaneously, because no interaction with the target database or provenance store is needed. Moreover, commits require about 25% of the average time for database interaction, but only occur once every five steps. The savings seem to be due to the reduced number of round-trips to the provenance database. For hierarchical-transactional storage, more time is needed for copies and inserts, but all the basic operations take at most 6% of the total time. Commits take the same amount of time on average as for hierarchical provenance.

The effects of deletion are shown in Figure 11. For naïve and hierarchical provenance, deletion simply adds provenance records. For transactional provenance, some deletion patterns result in fewer overall records being stored, because some data is inserted and deleted in the same transaction. However, hierarchical-transactional provenance displays the most stable behavior, and stores the fewest records among the approaches for each update pattern.

The effect of transaction length on processing time is shown in Figure 12. Processing time per basic operation does not vary much with transaction size, while the amount of time needed to process a commit grows approximately linearly with transaction length. The average overall time per operation remains about the same. These results reflect the expected behavior, and illustrate that our approach works at interactive speeds (at most one or two seconds) for transactions of up to 100 operations. Committing the corresponding changes to the target database is likely to take as long or longer. More sophisticated techniques that minimize network round trips during commits could further reduce the overall processing time.

Finally, Figure 13 displays the time needed to query the various forms of provenance using the $getSrc$, $getMod$, and $getHist$ queries. In general, it is expected that $getHist$ will outperform $getSrc$, and both will do better than $getMod$ based on the provenance store access patterns and data manipulation inherent in each. The $getSrc$ and $getHist$ queries run slightly (15%) faster for hierarchical provenance, but interestingly, the $getMod$ query is about 20% slower; there is no benefit over the naïve version since each query must process all the descendants of a node, including ones not listed in the provenance store. The queries ran fastest for transactional provenance; for all three queries, we observed a speedup of roughly a factor of 2.5 relative to the naïve approach. This makes sense because transactional provenance stores only about 25–35% as many records as the naïve approach. Of course, this is because transactional provenance is less descriptive than the naïve approach; however, this seems like a reasonable tradeoff, especially since the user can decide which versions of the database to commit. Hierarchical-transactional provenance benefits from the reduced number of records inherent in the transactional method, so both $getSrc$ and $getHist$ perform as well as for the transactional approach, but $getMod$ runs only slightly faster than for the naïve approach.

5. RELATED WORK

Biological and other scientific databases have dealt with provenance tracking in *ad hoc* ways. The Saccaromyces Genome Database [7] uses triggers to store records of updates to the database, but this only provides the history of local changes. In most cases the provenance of copied data is recorded manually. For example, in the Nuclear Protein Database, links to the PubMed bibliographic database or to other protein databases such as Entrez or UniProt are entered manually by the curator alongside the relevant data.

As mentioned in the introduction, "workflow" or "coarse-grained" provenance has been studied extensively in the context of scientific computation [10, 14, 26]; Bose and Frew [3] and Simmhan et al. [21] survey most existing research on such systems. These approaches record the process used to derive processed data products from raw data. In scientific computation, especially Grid computing [27], provenance is an important issue for two reasons. First, it is important to record the conditions under which source data was recorded (that is, instrument settings and other conditions) in order to ensure repeatability. Second, a Grid computation (workflow) may run computations on several different machines on the Grid, each running different operating systems, different versions of analysis software, etc., all of which may affect the results in subtle ways. Provenance is important for tracking down the source of anomalies, for ensuring experiments are repeatable, and for assessing the quality of the results. Also, provenance can be used to identify and avoid costly duplicate recomputations.

We have intentionally focused on provenance tracking for step-by-step copy-paste updates because we feel that this is an important problem that has received relatively little attention. Of course, it is also important to be able to propagate provenance and other annotations through more traditional relational database queries (or extensions to complex objects or XML), and to support provenance tracking for bulk updates. A fair amount of work has already been done on provenance and annotation for relational queries (including [2, 6, 8, 16, 22, 24]); however, issues such as how to assign provenance information to the results of joins and unions which may "fuse" data from different sources remain ill-understood. One popular approach (used in [2, 22]) to dealing with joins and unions that appears to be consistent with our framework is to interpret data that has been "fused" as "coming from" both source locations.

To deal with grouping and aggregation operations such as SQL's `GROUP-BY` and `SUM`, `AVERAGE`, etc., one obvious approach is to say that the provenance of the result of an aggregation operation is $\perp$, since the result is not *copied* from the input, but instead *computed*. Another approach is to say that it "came from" all of the locations involved in computing it; this is what is done in work on "why-provenance" and lineage [6, 8, 22, 24]. A third, and we believe the most general, approach is to define the provenance of a computed value as an expression describing how the value was computed. For example, if R contains tuples t_1 and t_2, then the provenance of the result of `SELECT COUNT(*) FROM R` would be an expression such as $\mathtt{count}(R/t_1, R/t_2)$. This is similar to the approach taken in "workflow" provenance for scientific computation such as Chimera [10].

Our approach to provenance overlaps with several other well-studied areas, including transaction logging, data availability, schema evolution, archiving, file synchronization, and version control. In the rest of this section we discuss the relationship between our work and these areas.

Logging Many database and filesystems systems use *transaction logging* or *journaling* in order to provide crash recovery. Such logs store detailed information about update operations applied to the database. This information is necessary to undo the effects of any transactions that had not committed at the time of a crash. Since provenance tracking is similar in some respects to logging, one might argue that provenance tracking is redundant or unnecessary in a database system that already performs logging. However, logging serves a much different purpose, and transaction logs do not provide as much information as provenance; so, to achieve the same effect, it would be necessary to add extra instrumentation that stores additional information to the logging system. In our opinion, this would be a mistake: such application-level code and data has no place in a system-critical mechanism.

Data availability One natural question is whether it makes sense to retain provenance information if the original data source becomes unavailable. The answer is an emphatic *yes*: such provenance information is impossible to reproduce, so potentially priceless. Provenance information for "lost" data can even help us recover the lost data from copies. For example, suppose two databases T_1 and T_2 are constructed using data from S, that the construction process is recorded by provenance stores P_1, P_2, and that later S disappears. We can still be fairly certain about the contents of S, since we can use the provenance records of T_1 and T_2 to partially reconstruct S. Even if T_1 and T_2 disagree about the contents of S (which could easily happen due to changes to S or due to errors in T_1, T_2, P_1 or P_2), this information may be better than nothing.

Schema evolution Another natural question is how our approach relates to schema evolution. For example, suppose we construct T from S_1 and S_2, storing provenance in P, and later the schema of S_1 (or symmetrically, T) changes. Does this invalidate the provenance information in P, and if so, how should we bring it up to date? There are at least two sensible reactions. First, we could take the view that the provenance information should always reflect the "real" relationship between the *current versions* of the databases involved. This is the approach traditionally taken in data integration and schema evolution. Then changes to the schemas *do* make the provenance information out-of-date. However, this tends to be expensive to correct, and to require centralization or cooperation among databases to control and coordinate changes to schemas. Moreover, this imposes on each database curator the responsibility to monitor changes to all relevant source databases, and to update the provenance information whenever any of them changes.

Alternatively, we could take the view that the provenance information *records* what happened as it happened, so that later changes to schemas do not render the information meaningless; they only make it harder to *interpret* the provenance information with respect to the *current* version. Moreover, in this setting, the relationship between the old and new versions of S_1 can also be captured by a provenance mapping; thus, if S_1 is tracks and publishes its own provenance, this information can be used to relate the provenance stored by T to the current version of S_1.

Version control, archiving, and synchronization Version control [17], archiving [5], and file synchronization [11] are closely related to our approach to provenance, but they do not address the same problem. Such techniques aim to preserve or reconcile the states of the data as it evolves over time, but they tell us only how the versions differ, not how the changes were *actually* performed. Moreover, these systems typically do not track changes that span multiple systems. Conversely, provenance identifies the source of information in the current version, but gives us no guarantee that the cited information has been preserved. The information may be in a database that has been updated since the data was extracted, and if the database has not been archived, there will be no confirming evidence for the information that has been extracted. We believe that both provenance recording and archiving are necessary in order to preserve completely the "scientific record."

6. FUTURE WORK

There are several obvious extensions to this work. The first is to provide a user interface that is acceptable to the curator; that is, it should not be too different from what is currently being used. The current practice in many situations is to use web browsers to acquire and update information. Reimer and Douglas [20] have investigated usability and architectural considerations in the development of a prototype *Web notebook* called NetNotes. In NetNotes, users can copy data from Web sites and paste it into a personal workspace or notebook. When data is copied in this system, it is copied as HTML structure rather than text, with reference to the source page. O'Mullane et al. [19] and other researchers in Grid and scientific computing have investigated similar "personal workspace" approaches for supporting scientific data analysis needs. We believe that our approach can be combined with such systems to provide support for tracking the provenance of both copied and computed data with little change to current practice.

We are currently looking at extensions to the basic language of atomic updates to languages that allow "bulk" updates. For example, it is common in curated databases to copy citation data from standard sources, and it may be laborious to do this for thousands of citations, each of which may need to be restructured according to some standard recipe. The technical challenge here is to connect the semantics of a bulk update language based on copy-paste operations with that of standard languages such as the query and update languages of SQL. In this setting transactional provenance is most natural because of the inherent parallelism in conventional update and query languages. To use naïve provenance would negate almost any form of query optimization.

In the presence of bulk updates, the amount of provenance information could become overwhelming, since the number of provenance records corresponding to a query or bulk update may be proportional to the size of the database, not the size of the transformation. An alternative is to store *approximate* provenance records. For example, we have experimented with using XPath expressions to over-approximate the full set of provenance links generated by bulk updates. In this approach, a record such as

$$\mathsf{Prov}(t, \mathsf{C}, T/a/*/b, S/a/*/b)$$

indicates that transaction t may include some links from target paths matching $T/a/*/b$ to source paths matching $S/a/*/b$; this single link may abbreviate a large number of more detailed links containing explicit identifier information. The storage needed for approximate provenance remains proportional to the size of the query or update, so is negligible compared to the requirements for the full provenance table. However, provenance queries can no longer be answered with certainty. Instead, we can only say that some data *may* (or *cannot*) have come from a given source location. This may be an acceptable price to pay to be able to store simple provenance information much more efficiently for bulk updates.

7. CONCLUSIONS

Provenance information is essential for assessing the integrity and value of data, especially in scientific databases. Managing provenance metadata alongside ordinary data adds to the high cost of scientific databases that are "curated", or constructed by hand by expert users who either enter raw data or copy existing data from other sources. Therefore, automatic techniques for collecting and managing provenance in such situations would be very beneficial. However, this is a challenging problem because it requires tracking data as it is copied between databases or modified by curators.

In this paper, we have proposed a realistic approach to automatic provenance tracking in curated databases. We have implemented our approach and conducted an experimental evaluation of several methods of storing and managing provenance. The most naïve approach we investigated has relatively high storage cost (storage overhead is proportional to the amount of data touched by an update), moderate processing cost (overhead of up to 30% of update processing time), and even simple provenance queries are fairly expensive to answer. However, the hierarchical-transactional technique reduced the storage overhead in our experiments by around a factor of 5, while decreasing the processing overhead per update operation to at most 6% and providing improved performance on provenance queries.

These experimental results affirm that provenance can be tracked and managed efficiently using our approach. We believe that this is a promising first step towards providing powerful, general-purpose tools that will make life easier for scientific data curators and increase the reliability and transparency of the scientific record.

Acknowledgments

We wish to thank Wendy Bickmore, H. V. Jagadish, Val Tannen and Stijn Vansummeren for discussions on this work. This work was supported by NSF through a graduate fellowship, by NIH project 1-U54-DA021519-01A1, by EPSRC project R37476 and by funding from the Royal Society.

8. REFERENCES

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

[2] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 900–911. Morgan Kaufmann, 2004.

[3] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[4] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. *Computer Networks*, 39(5), August 2002.

[5] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.

[6] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[7] J. Cherry, C. Adler, C. Ball, S. Chervitz, S. Dwight, E. Hester, Y. Jia, G. Juvik, T. Roe, M. Schroeder, S. Weng, and D. Botstein. SGD: Saccharomyces genome database. *Nucleic Acids Res.*, 26(1):73–79, 1998.

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

[9] G. Dellaire, R. Farrall, and W. A. Bickmore. The nuclear protein database (NPD): sub-nuclear localisation and functional annotation of the nuclear proteome. *Nucleic Acids Research*, 31(1):328–330, 2003.

[10] I. Foster, J. Vockler, M. Eilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *International Conference on Scientific and Statistical Database Management*, pages 1–10, July 2002.

[11] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005.

[12] M. Y. Galperin. The molecular biology database collection: 2006 update. *Nucl. Acids Res.*, 34:D3–D5, Jan 2006. doi:10.1093/nar/gkj162.

[13] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, G. Heber, and D. DeWitt. Scientific data management in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Research, January 2005.

[14] P. Groth, S. Miles, W. Fang, S. C. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05)*, 2005.

[15] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[16] T. Lee, S. Bressan, and S. E. Madnick. Source attribution for querying against semi-structured documents. In *Workshop on Web Information and Data Management*, pages 33–39, 1998.

[17] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 581–590. Morgan Kaufmann, 2001.

[18] Mimi. http://mimi.ctaalliance.org.

[19] W. O'Mullane, J. Gray, N. Li, T. Budavari, M. A. Nieto-Santisteban, and A. Szalay. Batch query system with interactive local storage for SDSS and the VO. In F. Ochsenbein, M. Allen, and D. Egret, editors, *Astronomical Data Analysis Software and Systems XIII*, volume 314 of *ASP Conference Series*, 2004.

[20] Y. Reimer and S. A. Douglas. Implementation challenges associated with developing a web-based e-notebook. *Journal of Digital Information (JoDI)*, 4(3), 2003.

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

[22] W. Tan. Containment of relational queries with annotation propagation. In *Proceedings of the International Workshop on Database and Programming Languages (DBPL)*, 2003.

[23] UniProt. http://www.ebi.ac.uk/uniprot/.

[24] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[25] N. Wiwatwattana and A. Kumar. Organelle DB: a cross-species database of protein localization and function. *Nucleic Acids Research*, 33:D598–604, 2005.

[26] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *International Conference of Data Engineering*, 1997.

[27] J. Zhao, C. A. Goble, R. Stevens, and S. Bechhofer. Semantically linking and browsing provenance logs for e-science. In *ICSNW*, pages 158–176, 2004.