Lost Source Provenance

Jing Zhang[*]
University of Michigan
4957 CSE Building
2260 Hayward Street
Ann Arbor, MI 48109
jingzh@umich.edu

H.V. Jagadish
University of Michigan
4601 CSE Building
2260 Hayward Street
Ann Arbor, MI 48109
jag@umich.edu

ABSTRACT

As the use of derived information has grown in recent years, the importance of provenance has been recognized, and there has been a great deal of effort devoted to developing techniques to identify individual source tuples used in the derivation of any result tuple. Often, however, the source database may have been updated since the result was derived, and the source tuples of interest are not in the database any more. In such situations, the provenance management system has to reconstruct relevant historical fragments of the source database as they were at derivation time. In this paper, we develop techniques to address this problem. Our experimental assessment shows that these techniques do so efficiently, and with low storage overhead.

1. INTRODUCTION

Provenance of a derived data item in a database explains how this data item is derived from other (source) data items. In a database that allows overwriting operations, such as deletes or updates, the source data items might be modified or removed after the derivation was finished.

Consider a data repository that provides data to distributed online users. Suppose a user derived a data item D from a data item S in the repository by using a query Q, and stored D in his local database. After the user finished the derivation, S in the repository was updated to a new value. Later, the user wanted to retrieve the source data item that is used to derive D. Where to find the proper value of S that is used in the derivation, given the query Q and the data item D? If the value of S had been updated more than once, then which one is the proper one?

EXAMPLE 1.1. *Consider a simple source database comprising two tables[1] Book and Price as shown in Figure 1 and Figure 2 respectively. A query Q_{bb} is shown in Figure 4, which selects all the books whose prices are under or equal to 10 dollars. The query result of Q_{bb} is stored in a new table called BargainBook, as shown in Figure 3. After the query Q_{bb} is executed, the update U_p (Figure 4) on the table Price is executed, which increases the prices of books by Stephen Hawking.*

ISBN	Title	Author	since
0007208642	1940s Omnibus	A. Christie	0
0002310198	After the Funeral	A. Christie	0
0553380168	A Brief History of Time	S.W. Hawking	0
0742627098	Adventures of Gerard	A.C. Doyle	0

Figure 1: Book

ISBN	Price	since
0007208642	9	0
0002310198	12	0
0553380168	10	0
0742627098	25	0

Figure 2: Price

Title	Price	since
1940s Omnibus	9	1
A Brief History of Time	10	1

Figure 3: BargainBook

Q_{bb}	`SELECT b.Title, p.Price`
	`FROM PRICE p INNER JOIN BOOK b`
	`ON p.ISBN=b.ISBN`
	`WHERE p.Price<=10`

U_p	`UPDATE PRICE`
	`SET p.Price=p.Price*1.1`
	`FROM PRICE p INNER JOIN BOOK b`
	`ON p.ISBN=b.ISBN`
	`WHERE b.Author='Stephen Hawking'`

Figure 4: Two Database Operations

Taking the tuple ⟨"A Brief History of Time", 10⟩ as an example. Its source tuple in the table *Price* is ⟨0553380168, 10⟩, which is no longer in the table *Price* after the two operations in Figure 4 finished executing. In fact, this book is now priced 11 dollars, and would not be considered a bargain book at its current price. Explaining its inclusion in the

[*]This work was supported in part by NIH grant #U54DA021519.

[1]The attribute named *since* in the tables is reserved for our provenance approach. Its meaning will be explained in Section 2.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

BargainBook table requires an understanding of its historical price.

Existing techniques of provenance retrieval, such as tracing queries [7] or propagating the identifiers of the source rows as annotations [11], rely on an assumption that the source tuples are still in the database. Thus, they can be retrieved either by executing tracing queries or by referring to their identifiers. This assumption holds when the derived data is synchronized with the source data, e.g., the data in a materialized view is synchronized with the source data in the base tables. However, in many scenarios, derived data does not reflect the changes made to the source data, e.g., the derived data is stored locally while the source data is in some remote repository.

When the source data items are not current in the database, for the existing techniques to find them, a historical version of the database needs to be reconstructed such that this historical version is exactly what the database was when the derivation happened. Thus, the existing techniques can be applied to this historical version. However, to reconstruct a complete database to its state at some previous time is expensive.

Usually only a small portion of the database is used in the derivation, which means during the provenance retrieval, only this portion of the database at derivation time needs to be reconstructed. In particular, the tracing queries can be extended such that they search not only the current database but also the relevant historical data, which is just enough to reconstruct the involved portion of the database.

In this paper, we describe such an improved approach to provenance capture and retrieval, which is aware of historical data and can retrieve source data items when they are not current in the database. This paper is organized as follows: Section 2 is the background information about database provenance and historical data; Section 3 gives the overview of our approach and introduces the necessary data structures required by our approach; Section 4 describes the provenance reconstruction algorithm in detail and analyzes its time and space costs; Section 5 shows the experimental results of these costs in a realistic setting; finally, Section 6 and Section 7 are the related work and the conclusion respectively.

2. PRELIMINARIES

In a relational database, the database operations are usually intended to manipulate tuples. Therefore, we focus on the provenance of tuples.

We denote the database as D. The tables inside D are denoted using capital letters, e.g., T. We assume set semantics, therefore, every table is a set of tuples. Tuple variables are denoted with small letters, e.g., t or s.

For each table T, its attribute is denoted as capital letters, e.g., $T.A$, where A is an attribute. The schema of a table T is denoted as $T : \langle A_1, ..., A_m\rangle$, where $A_1, ..., A_m$ are attributes.

2.1 Query Language

A safe query on the database can be expressed in Tuple Relational Calculus (TRC) as $\{t|f(t)\}$, where t is the only free tuple variable in the formula $f(t)$. In this TRC query, the part before | is called *answer*. Sometimes, a TRC query also takes the form $\{t : \langle A_1, ..., A_n\rangle \mid f(t)\}$, where $\langle A_1, ..., A_n\rangle$ is a list of attributes in the answer tuple, also called *target list*.

In this paper, we only consider conjunctive queries with aggregations, i.e., ASPJ queries. An SPJ query (a.k.a. a conjunctive query) expressed in TRC is of the form $\{t|\exists s_1, ..., s_m\ S_1(s_1)\wedge ...\wedge S_m(s_m)\wedge f(s_1, ..., s_m, t)\}$. $S_i(s_i)$ $(i = 1, ..., m)$ is an atomic formula that evaluates to true if s_i is a tuple in the table S_i. $f(s_1, ..., s_m, t)$ is a conjunction of atomic formulas. The atomic formula is either a predicate, e.g., $S_1(s_1)$, or the comparison of a table's attribute to some value or some other attribute, e.g., $S_1.A = 2$. In particular, $f(t)$ in an SPJ query does not contain the universal quantifier $\forall$ or the negation $\neg$ or the disjunction $\vee$.

An SPJ query can be extended with aggregations by allowing aggregate attributes in the target list, e.g., $\{t : \langle A_1, ..., A_n, G\ AS\ agg(A_{n+1})\rangle \mid \exists s_1, ..., s_m\ S_1(s_1)\wedge ...\wedge S_m(s_m)\wedge f(s_1, ..., s_m, t)\}$. In this form, A_1 through A_n are grouping attributes, A_{n+1} is the attribute to which the aggregate function agg is applied, and G is a new attribute storing aggregate values. If in a query, aggregate attributes appear in the formula part (the part after |), the query can be decomposed into several queries by introducing new intermediate tables, and each of them either has aggregate attributes only in its target list or does not have aggregations. We will show how this is done in Appendix B. In fact, this decomposition corresponds to the decomposition of an aggregate query into canonical segments in [7]. Since an ASPJ query sometimes needs to be divided into several (A)SPJ queries, the provenance retrieval for this ASPJ query becomes a recursive process correspondingly.

From here on, we use $\{t : \langle A_1, ..., A_n, G\ AS\ agg(A_{n+1})\rangle \mid \exists s_1, ..., s_m\ S_1(s_1) \wedge ... \wedge S_m(s_m) \wedge f(s_1, ..., s_m, t)\}$ as the general form of queries under our consideration. Notice that S_i and S_j $(i \neq j)$ may refer to the same table.

2.2 Provenance of Tuples

The provenance of a given tuple can be defined in many different ways. In this paper, we adopt the definition from [7]. We restate the definition here.

DEFINITION 2.1. ***[Provenance]*** *Assume a database D have tables $T_1, ..., T_n$. Given a tuple t in the result set of a query Q executed on D, denoted as $t \in Q(T_1, ..., T_n)$, the provenance of t is a subset of D that has tables $T'_1, ..., T'_n$, where $T'_1, ..., T'_n$ are the* ***maximal*** *subsets of $T_1, ..., T_n$ such that:*

1. $\{t\} = Q(T'_1, ..., T'_n)$
2. $\forall T'_k : \forall t' \in T'_k : Q(T'_1, ..., T'_{k-1}, \{t'\}, T'_{k+1}, ..., T'_n) \neq \varnothing$

If a single table is referenced more than once in the query, e.g., in the case of self-joins, each instance is regarded as a separate table, and is renamed correspondingly such that each table name only appears once in the query. With this treatment, there always exists a set of tables, $T'_1, ..., T'_n$, that satisfies the two requirements listed in the above definition. Moreover, in this paper, we assume set semantics. Under set semantics, it has been proved in [7] that the provenance defined above for a given tuple is unique.

2.3 Historical Data

When a tuple in the database is deleted or updated, this tuple becomes a historical tuple. As illustrated in the example in Section 1, this historical tuple can be the source tuple of some derived tuple and may need to be retrieved. In such cases, the archiving of historical tuples is essential to the retrieval of provenance.

The implementation of storing historical data can be done in many different ways as explored in the temporal database literature. Although the intensive study on data models, indexes and query languages of historical data is essential to temporal databases, it is out of the scope of this paper. We adopt a simple implementation of historical data storage, since it does not impact our purpose to show the improvement of our provenance approach over the baseline approach.

In the next section, we are going to describe our way of storing historical data, together with other auxiliary data structures necessary for the retrieval of non-current provenance.

3. APPROACH OVERVIEW AND AUXILIARY DATA STRUCTURES

Our approach to provenance retrieval consists of two steps: constructing an extended tracing query and executing it. The extended tracing query, in order to retrieve provenance that is not current in a database, should be able to retrieve both from the historical data and from the current database.

The historical data is stored in several data structures. These extra data structures are populated every time a data operation happens and are queried by our extended tracing queries.

We define two additional table-like data structures: a *provenance log* and a set of *shadow tables*. Furthermore, we define an extra annotation attribute *since* in each regular relational table.

3.1 Provenance Log

Most, if not all, databases have logs. Usually, there is more than one log, each for a specific purpose. Therefore, each of these logs can be designed in a way such that it can best serve a specific purpose.

In our approach, we define a provenance-oriented log: *provenance log*.

If a specific DBMS is under consideration, its existing logs may already be sufficient for provenance purpose in the sense that they have all the information that the provenance log has. If so, the provenance log does not necessarily have to be an additional log, but instead it can be some view defined over the existing logs.

The provenance log, denoted as $Plog$, consists of a sequence of log entries. Each entry corresponds to an operation executed in the database system. Each entry has the structure $(ID, timestamp, user, sqlStatement)$. ID is an unique ID assigned to every entry in the log, and an operation that is committed later has a bigger ID for its corresponding log entry. That is to say, the ID indicates the order of the commitment of all the operations. $sqlStatement$ stores the SQL statement of the committed operation. $timestamp$ is the time when the operation is committed. $user$ specifies the user who commits the operation.

Recall Example 1.1. The provenance log after the execution of the two operations Q_{bb} and U_p is shown in Figure 5.

ID	$timestamp$	$user$	$sqlStatement$
1	2009-08-01 01:00:00	Alice	Q_{bb}
2	2009-08-02 11:00:00	Bob	U_p

Figure 5: Provenance Log Example

3.2 Shadow Table

Historical tuples are stored in shadow tables. For each regular table in the database, we define a corresponding shadow table.

For example, if a regular table is of schema $T : \langle a_1, a_2 \rangle$, then the shadow table of T is $T_{sh} : \langle a_1, a_2, begin, end \rangle$. The attributes *begin* and *end* are foreign keys referring to the attribute ID in the provenance log. The attribute *begin* stores an ID whose corresponding entry in the provenance log records the operation that generates this tuple. The attribute *end* stores an ID whose corresponding entry in the provenance log records the operation that removes this tuple.

The attributes *begin* and *end* are to specify the time period when the historical tuple was current. We choose to use the IDs of log entries instead of the actual times to avoid ambiguity: two committed operations can have the same time of commit but can not have the same log entry ID.

Recall Example 1.1. After the update U_p, the table $Price$ is like the one shown in Figure 7; and its shadow table $Price_{sh}$ is shown in Figure 6.

ISBN	Price	begin	end
0553380168	10	0	2

Figure 6: Shadow Table $Price_{sh}$ After U_p

ISBN	Price	since
0007208642	9	0
0002310198	12	0
0553380168	11	2
0742627098	25	0

Figure 7: Table $Price$ After U_p

3.3 Annotation Attribute

Current tuples are stored in regular tables. An extra annotation attribute called *since* is added to each regular table, which is a foreign key referring to the attribute ID in the provenance log. The attribute *since* stores an ID whose correspondent entry in the provenance log stores the operation that generates this tuple.

This extra annotation attribute is not visible to the users of the database, and thus it can not be manipulated by the users. The provenance capture and retrieval are the only procedures that can set its value or query it.

It is desirable that this annotation attribute is included in the schema during the database design and before the database population. If a database is already populated with a schema without this attribute, the alteration of adding this attribute to each table is not very welcome. Therefore, an alternative approach will be creating a separate table that links each tuple in the database, via some unique tuple ID, to this attribute. In this paper, we assume that this annotation attribute *since* is already included in each table in the database schema.

3.4 Populating Auxiliary Data Structures

All the auxiliary data structures are populated whenever a database operation takes place.

1. When a database operation is committed, a new entry is created in the provenance log and a unique ID is as-

signed to this new entry. When multiple operations are committed together as in a single transaction, each of them will have a log entry in the provenance log upon the time of commitment. The order of these entries is the execution order of the corresponding operations. Any operation that is not committed will not have an entry in the provenance log.

2. When a tuple is inserted into a table due to this database operation, the value of its *since* attribute is set with the ID of the newly created entry in the provenance log.

3. When a tuple is removed from a table due to this database operation, either by a delete or by an update, the removed tuple is inserted into the corresponding shadow table. As for this new tuple in the shadow table, the value of the *begin* attribute is set with the value of the *since* attribute in the removed tuple; the value of the *end* attribute is set with the value of the ID of the newly created entry in the provenance log.

This populating of auxiliary data structures is in fact our provenance capture procedure. As we will see in the next section, all the provenance information we need is recorded in these auxiliary structures.

4. PROVENANCE RETRIEVAL

In this section, we show how to build the extended tracing queries to retrieve provenance using both the current database and the historical data for a given derived tuple.

The tracing query introduced in [7] is able to retrieve the provenance defined in Definition 2.1, if the provenance is current in the database. Those tracing queries are constructed based on the original query Q and the derived tuple t, and they use only the current database. When the provenance is not current, they need to be extended to make use of historical data.

We divide our discussion of the extended tracing queries into three parts. First, we revisit the tracing query introduced in [7]. Then, we extend the tracing queries such that they can find the (current or historical) provenance by utilizing the current database, the provenance log and the shadow tables. Finally, we analyze the time and space costs of provenance capture and retrieval.

4.1 Tracing Query Revisited

In [7], the tracing queries and the original queries are expressed in a relational algebra extended with aggregations. In this paper, we use Tuple Relational Calculus (TRC) instead, which can be extended with aggregations as well [14]. Moreover, this extended TRC is equivalent to the extended relational algebra [14].

Recall the example query Q_{bb} shown in Figure 4. The SQL statement of Q_{bb} can be expressed in TRC as follows:

$$\left\{ \begin{array}{l} t : \langle Title, Price \rangle \ | \\ \exists s_1 : \langle ISBN, Title \rangle, s_2 : \langle ISBN, Price \rangle \\ \big(Book(s_1) \wedge Price(s_2) \\ \wedge s_1.ISBN = s_2.ISBN \wedge s_2.Price <= 10 \\ \wedge t.Title = s_1.Title \wedge t.Price = s_2.Price \big) \end{array} \right\}$$

In the above TRC query, t, s_1, s_2 are tuple variables. $t.Title$, $s_1.ISBN$, etc. are attribute qualified tuple variables. $Book(s_1)$ and $Price(s_2)$ are atomic formulas. $Book(s_1)$ ($Price(s_2)$) evaluates to true, if s_1 (s_2) is a tuple from the table *Book* (*Price*). $s_1.ISBN = s_2.ISBN$, $s_2.Price <= 10$, etc. are also atomic formulas. Judging from its form, the above query is a safe conjunctive query (SPJ query).

The answer to a TRC query is a set of tuples. Each of these tuples, when assigned to the tuple variable in the answer part of the query, can make the formula part evaluate to true. Since the answer to a TRC query is a set of tuples, it can been seen as a relation or a table.

When the original query Q_{bb} is expressed in TRC, its tracing query can be expressed in TRC as well. The tuple ⟨"A Brief History of Time", 10⟩ is an answer tuple to Q_{bb}. To retrieve its provenance in the table *Book*, we can use the following tracing query:

$$\left\{ \begin{array}{l} \underline{s_1 : \langle ISBN, Title \rangle} \ | \\ \exists s_2 : \langle ISBN, Price \rangle, \underline{t : \langle Title, Price \rangle} \\ \big(Book(s_1) \wedge Price(s_2) \\ \wedge s_1.ISBN = s_2.ISBN \wedge s_2.Price <= 10 \\ \wedge t.Title = s_1.Title \wedge t.Price = s_2.Price \\ \wedge \underline{t.Title = \text{"A Brief History of Time"}} \wedge \underline{t.Price = 10} \big) \end{array} \right\}$$

The tracing query is like a "re-organization" of the original query, as shown in the above example with underlines.

1. The tuple variables s_1 and t are switched such that the source tuple variable s_1 is now in the answer part and t is now in the formula part.

2. More conditions are added to the formula part, i.e., the value of each attribute of the derived tuple t is specified, e.g., the last line of the above tracing query.

In general, given a query Q as

$$\left\{ \begin{array}{l} t : \langle A_1, ..., A_n, G \ AS \ agg(A_{n+1}) \rangle \ | \\ \exists s_1, ..., s_m \\ \big(T_1(s_1) \wedge ... \wedge T_m(s_m) \wedge f(s_1, ..., s_m, t) \big) \end{array} \right\} \quad (1)$$

and given a tuple $t = \langle a_1, ..., a_n, g \rangle$ in the query result, the tracing query to find its provenance in the table T_k is, assuming T_k has a schema $B_1, ..., B_l$,

$$\left\{ \begin{array}{l} s_k : \langle B_1, ..., B_l \rangle \ | \\ \exists t, s_1, ..., s_{k-1}, s_{k+1}, ..., s_m \\ \big(T_1(s_1) \wedge ... \wedge T_m(s_m) \wedge f(s_1, ..., s_m, t) \\ \wedge t.A_1 = a_1 \wedge ... \wedge t.A_n = a_n \big) \end{array} \right\} \quad (2)$$

We show in Appendix A that this tracing query can retrieve the provenance defined in Definition 2.1.

4.2 Extended Tracing Query Aware of Historical Data

Given a derived tuple t, if its provenance is not current in the database, we can retrieve its provenance with our extended tracing queries. Compared to the classic tracing query as in Equation 2, the extended tracing query need an extra piece of information, i.e., the ID of the provenance log entry that records the original query.

Recall our discussion in Section 3.2. The IDs of provenance log entries can be used as timestamps to indicate time points or periods of time, e.g., storing these IDs in the attributes *begin*, *end* and *since*. We have also argued that these IDs are even better than real timestamps since they incur no ambiguity.

Similarly, the ID of the provenance log entry that records the original query represents the derivation time, i.e., the time when the original query was executed. Therefore, with this ID, our extended tracing query is able to decide which historical data is proper to retrieve provenance from, i.e., the data that was current in the database at the derivation time is proper.

Recall the book example, for the tuple ⟨"A Brief History of Time", 10⟩ generated by the query Q_{bb}, an extended tracing query that can retrieve the provenance of it in the table *Price* is

$$\Big\{ \; s_2 : \langle ISBN, Price\rangle \;\; | \\ \exists s_1 : \langle ISBN, Title\rangle, t : \langle Title, Price\rangle \\ \big(\underline{Book^H(s_1)} \wedge \underline{Price^H(s_2)} \\ \wedge s_1.ISBN = s_2.ISBN \wedge s_2.Price <= 10 \\ \wedge s_1.ISBN = t.ISBN \wedge s_2.Price = t.Price \\ \wedge t.Title = \text{“A Brief History of Time”} \wedge t.Price = 10 \;\; \Big\}$$

Compared to the classic tracing query, the difference is that the *Book* and *Price* predicates are changed into $Book^H$ and $Price^H$ respectively. $Book^H$ ($Price^H$) is the historical version of the table $Book(Price)$ when Q_{bb} took place.

In the book example, there are no updates on the table *Book*. Therefore, $Book^H$ is the same as *Book*. However, $Price^H$ and *Price* are different due to the update U_p. To construct $Price^H$, we should remove any tuple that enters the table *Price* after the execution of Q_{bb}; and add any tuple that leaves the table *Price* after the execution of Q_{bb}. Therefore, $Price^H$ can be constructed as a view using the following query:

$$\Big\{ \; s : \langle ISBN, Price\rangle \;\; | \\ \big(Price(s) \wedge \underline{s.since < 1}\big)\vee \\ \exists s_h\big(Price_{sh}(s_h) \wedge \underline{s_h.begin < 1} \wedge \underline{s_h.end >= 1} \\ \wedge s.ISBN = s_h.ISBN \wedge s.Price = s_h.Price\big) \;\; \Big\}$$

In this example of extended tracing query, the query formula consists of two formulas connected by a union. The first formula selects source tuples from the current table *Price*. In order for a current *Price* tuple to be possible provenance, it should have been current before Q_{bb} happened. In this example, Q_{bb} is logged in the provenance log with an entry ID 1. Therefore, the first formula has a condition $\underline{s.since < 1}$. This condition is to make sure the source tuple is already in the database when Q_{bb} took place.

The second formula selects source tuples not currently in the table *Price*. These historical tuples are stored in the shadow table of *Price*, i.e., $Price_{sh}$. Similar to the case of current tuples, in order for a historical tuple to be possible provenance, it should be current when Q_{bb} took place. This requirement is checked through the conditions $\underline{s_h.since < 1} \wedge$ $\underline{s_h.end >= 1}$.

From the above example, we can see that, in order to build an extended tracing query, we need the ID of the provenance log entry that records the original derivation query. Thus, the construction of an extended tracing query needs three pieces of information:

1. the derived tuple
2. the original query
3. the ID of the provenance log entry recording the original query

In general, if given a tuple t, whose derivation query Q is logged in a provenance log entry with ID being id, assuming Q is of the form as shown in Equation 1, then the extended tracing query to retrieve provenance in the table T_k is

$$\Big\{ \; s_k : \langle B_1, ..., B_l\rangle \;\; | \\ \exists t, s_1, ..., s_{k-1}, s_{k+1}, ..., s_m \\ \big(T_1^H(s_1) \wedge ... \wedge T_m^H(s_m) \wedge f(s_1, ..., s_m, t) \\ \wedge t.A_1 = a_1 \wedge ... \wedge t.A_n = a_n\big) \;\; \Big\} \tag{3}$$

where T_k^H, assuming the shadow table of T_k is T_{k_sh}, is

$$\Big\{ \; s_k : \langle B_1, ..., B_l\rangle \;\; | \\ \big(T_k(s_k) \wedge s_k.since < id\big)\vee \\ \exists s'_k\big(T_{k_sh}(s'_k) \wedge s'_k.begin < id \wedge s'_k.end >= id \\ \wedge s'_k.B_1 = s_k.B_1 \wedge ... \wedge s'_k.B_l = s_k.B_l\big) \;\; \Big\} \tag{4}$$

Notice that, although the original derivation query is a conjunctive query with possible aggregations, the extended tracing query is not a conjunctive query, because of the union connective used in Equation 4.

4.3 Analysis of Efficiency of Extended Tracing Queries

We now analyze the space and time efficiency of our approach.

4.3.1 *Space Cost*

The archiving of historical data can be done at different granularities. For example, if one attribute in one tuple in a table in a database is updated, to store the historical data, before the update, we can back up (i) the whole database, (ii) the updated table, (iii) the updated tuple, or (iv) just the updated attribute in the tuple.

The size of the storage of historical data obviously depends on the granularity used in archiving. In the above example, each way of archiving can enable the recovering of the database before update, however, the last one incurs the minimum amount of storage.

In our approach, we archive the historical data at the granularity level of tuples, i.e., we archive a tuple in a proper shadow table when one or multiple attributes in this tuple are updated. Assume the average size of a tuple is $size_t$, and the number of tuples affected by an operation is n. Thus, after this operation, the size of the shadow tables is increased by $(size_t + C) \times n$, where C is a constant being the size of the two attributes *begin* and *end*.

Notice that the decreasing in the space cost also means the increasing in the time cost of reconstructing previous versions using historical data. For example, if the whole database is archived, the reconstruction of any table in the database at a previous time involves no complex queries

but almost merely selecting. Comparatively, since we only archive the updated tuple when one or more attributes in it are changed, the reconstruction of the involved table needs to run a query as shown in Equation 4.

Besides the archive of historical data, i.e., the shadow tables, our approach also incurs extra space cost due to the provenance log and the annotation attribute *since*. The size of the provenance log is linear to the number of entries in it if we assume a fixed size of each entry, which is possible if a maximum length of SQL statements is assumed. The size of the attribute *since* is the same as that of *begin* or *end*, since they all refer to a provenance log entry ID. The total cost of this attribute across the entire database will be the number of tuples in the entire database times the size of this attribute.

4.3.2 Time Cost

There are two types of time cost: the time cost of provenance capture and the time cost of provenance retrieval. The time cost of provenance capture is relatively smaller and more straightforward than that of provenance retrieval.

Provenance capture for every database operation is a two-step procedure: computing one new provenance log entry and/or new shadow table tuples; and inserting them into the provenance log and/or shadow tables.

The computation time is negligible, since the computation of either the log entry or the shadow table tuples is fairly simple. The insertion time of the log entry is constant, since there is always one log entry with a fixed size. The insertion time of shadow table tuples depends on the number of shadow table tuples generated by this operation. Assume n tuples are updated during an operation, and $insert_time_t$ is the average time of inserting one shadow table tuple. Then the time of inserting into shadow tables for this operation will be $insert_time_t \times n$.

The time cost of our provenance retrieval primarily consists of constructing an extended tracing query and executing it. The construction of an extended tracing query takes roughly a constant amount of time. On the other hand, the time of executing it varies with the reconstructed historical versions.

The historical version of a table consists of tuples from current table and shadow tables. The executing time of the extended tracing query is affected by both the number of tuples in the historical version and the location of these tuples. The former is easier to understand, since retrieving from a table/view with more tuples takes more time than retrieving from a table/view with less tuples. However, the second relationship is not so obvious.

In fact, if the reconstructed historical version has n tuples, and m of them comes from the shadow table, then the executing time is roughly proportional to m/n. This is later shown by an experiment in Section 5. The cause of this may be the specific physical plan of the union operations used in reconstruction. In the physical plan, the union operation is implemented as (i) two index seeks on the two tables, (ii) a concatenation of the outputs of the index seeks, and finally (iii) a sort of the output of concatenation. If most of the tuples in the output of concatenation are from the same table, the sort may be faster than in the case where tuples come evenly from the two tables.

5. EVALUATION

In this section, we evaluate, through experiments, the sizes of the provenance log and shadow tables, and the time of provenance retrieval.

In each experiment, we start with a set of tables; execute a workload consisting of queries, inserts, updates and deletes; then retrieve provenance for selected tuples in the result sets of the executed queries. The workload is specially made up such that some of the derived tuples in the result sets do have provenance that is not current in the database.

Our experimental tables and workloads are based on the database and transactions specified in TPC-E benchmark [12] with some simplifications and modifications. The reason we use TPC-E benchmark is that it has quite a few transactions that contain updates and deletes.

5.1 Tables And Workloads

The TPC-E benchmark simulates the activity of a brokerage company. The brokerage company interacts with customers and the financial market. A customer can have more than one account with the broker company. The customer can place trade orders through any of her accounts. The brokerage company buys or sells securities on the market according to the customers' trade orders. In the TPC-E specification, there are 33 tables and 13 transactions. In our experiments, we use 4 transactions out of 13 and these four transactions use 9 tables out of 33.

The transactions we used are customer_position (c-p), trade_order (t-o), trade_result (t-r) and market_feed (m-f). The tables we used are CUSTOMER (C), CUSTOMER_ACCOUNT (CA), HOLDING_SUMMARY (HS), TRADE (T), LAST_TRADE (LT), TRADE_HISTORY (TH), STATUS_TYPE (ST), SETTLEMENT (S), CASH_TRANSACTION (CT). Each of those 9 tables is read and/or written by some of these four transactions.

5.1.1 Table Generation

We generate the tables specified in TPC-E through EGen package. The sizes of the tables can be scaled through several parameters: the number of customers (NoC), the scaling factor (SF), the initial trade days (ITD). For example, the size of the table TRADE is ((ITD * 8 * 3600)/SF) * NoC. We set the number of customers to be 5000, the scaling factor to be 4500 and the initial trade days to be 30. Notice that the scaling factor is the number of customer rows per single Transaction-Per-Second-E(tpsE), and the scaling factor for nominal throughput is 500 [12]. Therefore, the scaling factor we choose is too big for a nominal output. Since we do not intend to report the database performance under TPC-E but to make use of the table settings and workloads, we use this big scaling factor in order to keep the database small. Given the parameters we choose, we have 33 tables of a total size being 3.4G bytes.

5.1.2 Transaction Generation

In the four transactions we use in our experiments, customer_position is a read-only transaction, and the other three transactions are read-write transactions. Each transaction can have more than one read and/or write. All the reads are queries. The writes can be inserts or updates or deletes. Some of the writes modify the tables that are read by customer_position. All these reads and writes are called database operations.

Although we execute every database operation (read or

write) in these four transaction, we do not capture every database operation in the provenance log. When a read only reads tables that are not used by any write, we do not capture provenance for it. This is because the result tuples of this type of read always have provenance in the current database, thus we are not interested in experimenting with them. When a write only writes tables that are not used by any read, we do not capture the provenance for it. This is because writes of this type do not have affect on the provenance of result tuples of reads, thus, we are not interested in these writes. All the other reads and writes are captured in the provenance log when they take place.

In Figure 8, we label the database operations, i.e., reads or writes, in the four transactions that the provenance capture is aware of. We also show the tables used in each of them. The database operation denoted as R is a read and the one denoted as W is a write.

Each of the 5 tables, shown in Figure 8, has a corresponding shadow table.

		CA	HS	T	LT	TH
c-p	$R^{\text{c-p}}_{\text{CA,HS,LT}}$	r	r		r	
	$R^{\text{c-p}}_{\text{ST,T,TH}}$			r		r
t-o	$W^{\text{t-o}}_{\text{T}}$			w		
	$W^{\text{t-o}}_{\text{TH}}$					w
t-r	$W^{\text{t-r}}_{\text{HS}}$		w			
	$W^{\text{t-r},1}_{\text{T}}$			w		
	$W^{\text{t-r},2}_{\text{T}}$			w		
	$W^{\text{t-r}}_{\text{TH}}$					w
	$W^{\text{t-r}}_{\text{CA}}$	w				
m-k	$W^{\text{m-f}}_{\text{LT}}$				w	
	$W^{\text{m-f}}_{\text{T}}$	w				
	$W^{\text{m-f}}_{\text{TH}}$					w

Figure 8: Reads and Writes of Tables in Transactions

5.1.3 Workload Generation

The central task we wish to evaluate is that of retrieving provenance by reconstructing the source tuples that are not in the current database. Therefore, in the workloads we use for the experiments, we want some updates that come after some queries and change (some of) the source tuples used by those queries. We generate two types of workload, both fulfilling this specific purpose.

The first type of workloads has a workload pattern that is the recurring sequence of the 4 transactions in the order of customer_position, trade_order, trade_result and market_feed. The order of trade_result and market_feed may be switched depending on if the order is a market order or a limit order. In particular, in such a workload, the tuples used in customer_position as source tuples are later modified by the following trade_order, trade_result and market_feed. The workloads of this type all have roughly the same ratio of read to write.

The second type of workloads always has a single customer_position at the beginning of the workload, and then has many recurring sequences of 3 transactions, i.e., trade_order, trade_result and market_feed. Unlike the workloads of the first type, the workloads of this type can have different ratios of read to write. With this type of workloads, we can manage to achieve different percentages of historical tuples in the reconstructed historical view, as will be explained in the discussion of the second experiment on time cost (Figure 13).

These two types of workload are both obtained by modifying the workloads generated by the CEE class in EGen package, since the original workloads generated by CEE contain transactions other than the four we use in this experimental evaluation.

1. we take a workload generated by the CEE class in EGen package;
2. keep only the transactions of trade_order;
3. for each trade_order, we generate a trade_result and a market_feed after it, and
   (a) for the first type of workload, a customer_position is generated before it
   (b) for the second type of workload, a customer_position is generated before it only when this trade_order is the first transaction in the workload.

5.1.4 Metrics

We have two metrics of primary interest: space and time. In space, we are primarily concerned with the space requirements of the auxiliary data structures that we require, to retain sufficient historical provenance information. We would like to measure this overhead. In time, we are concerned with the time required to reconstruct at least enough history to complete provenance explanation for a data item derived from updated sources. Obviously, we would like to minimize this time. We report measurements for both metrics in turn below.

5.1.5 Baseline for Comparison

The most important question to address is whether the space and time costs are acceptable, in an absolute sense. Is the overhead affordable to obtain the benefits of historical provenance? Of course, this question is addressed in the experimental results reported below. But there is also an additional question of interest: how much did our cleverness buy us? How do the techniques we developed in this paper compare against the state of the art before our work. How much better are we than a baseline? Of course, this begs the question of defining a suitable baseline. Since most provenance techniques are not capable of handling updatable sources, we really cannot use them directly for effective comparison. In fact, we already know, even without performing any experiments, that our techniques reduce to the method of tracing queries if there happen to be no updates performed to the source data.

Since the primary barrier to the use of classic provenance techniques in our problem scenario is the need for historical source data, a baseline approach will be leveraging the transaction-time temporal databases to query some previous state of the database and apply the existing provenance techniques to that previous state. However, it has two disadvantages. First, temporal databases are either queried via an extended language to SQL, e.g., introducing a new keyword "AS OF" [15]; or queried via XQuery, e.g., [17]. The former needs an extended query language standard, while the latter enforces the retrieval of provenance by several queries.

Second, it is a challenging problem in temporal databases to acquire transaction timestamps that are consistent with the transaction serialization order [13]. However, as long as the provenance retrieval is concerned, the sole usage of the serialization order is sufficient to reconstruct necessary historical data, as demonstrated by the usage of log entry IDs in our approach. Thus, there is no need of introducing transaction timestamping to cause unnecessary complexity. As a comparison, our approach exclusively uses SQL, and only depends on the commit order of transactions to achieve the reconstruction of previous states.

5.2 Experiments

We implement all our experiments in Java. We run the code using the JRE 6 update 14 from Sun, installed on a machine of 3.06GHz Celeron CPU with 1.96GB RAM memory running Microsoft Windows XP Professional 2002 SP 3.

5.2.1 Space Cost

The size of provenance log grows with the number of committed database operations. The size of a shadow table grows with the number of tuples updated or removed from the corresponding regular table.

We have 5 workloads of the first type with increasing amounts of transactions shown in Figure 9. In these 5 workloads, the ratios of read to write are roughly the same. The space cost of each of these 5 workloads is shown in Figure 10. The size of provenance log is close to linear with the number of committed operations, which can be queries, inserts, updates or deletes. In this particular experiment, the size of a single log entry, i.e., the provenance log cost per committed operation, is around 150 bytes. The size of shadow tables for each workload, shown in Figure 10, is the sum of 5 shadow tables, i.e., the shadow tables for TRADE, TRADE_HISTORY, CUSTOMER_ACCOUNT, HOLDING_SUMMARY and LAST_TRADE respectively. This total size of shadow tables is roughly linear with the number of updates in the workload. In this particular experiment, the space cost of shadow tables is around one third of the cost of the provenance log. In this experiment, the number of writes is a little more than double the number of reads. If the writes are less frequent, the space cost of shadow tables can be further reduced.

	Workload				
	1	2	3	4	5
$R^{c\text{-}p}_{CA,HS,LT}$	58	563	1123	1678	2248
$R^{c\text{-}p}_{ST,T,TH}$					
$W^{t\text{-}o}_{T}$	58	563	1123	1678	2248
$W^{t\text{-}o}_{TH}$					
$W^{t\text{-}r}_{HS}$	58	563	1123	1678	2248
$W^{t\text{-}r}_{TH}$					
$W^{t\text{-}r}_{CA}$	57	519	1031	1529	2070
$W^{t\text{-}r,1}_{T}$	13	100	190	316	423
$W^{t\text{-}r,2}_{T}$	58	563	1123	1678	2248
$W^{m\text{-}f}_{LT}$	58	563	1123	1678	2248
$W^{m\text{-}f}_{T}$	14	214	447	626	875
$W^{m\text{-}f}_{TH}$					
Total	562	5551	11099	16521	22227

Figure 9: Workloads

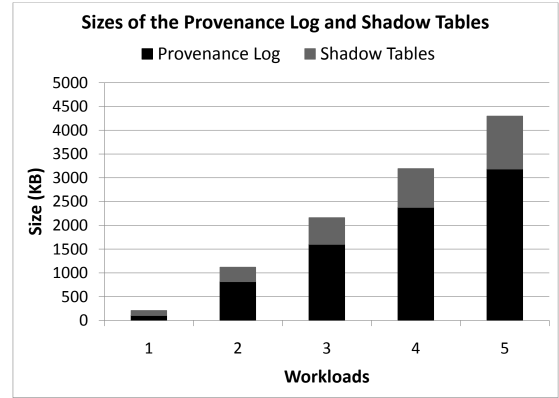

Figure 10: Sizes of the Provenance Log and Shadow Tables

5.2.2 Time Cost

We have two experiments showing respectively the absolute time cost of provenance retrieval and the relationship between the time cost and the ratio of historical tuples in the reconstructed historical views.

To show the absolute time cost of retrieving provenance using extended tracing queries, we experiment with the first workload as shown in Figure 9. In this workload, the query $R^{c\text{-}p}_{CA,HS,LT}$ is executed 58 times, and generates 322 tuples in total. Among these 58 executions, a later execution uses a different version of the database than an earlier execution, since we have padded updates between any two consecutive executions of the query. Therefore, the reconstructed historical versions of these 322 tuples are different.

The retrieval times of these tuples are shown in Figure 11. The vertical axis is the time of retrieval. The horizontal axis is the total size of all the reconstructed views in one tracing query. The size is measured with the number of tuples. Every point in the plot is the retrieval of provenance for one derived tuple.

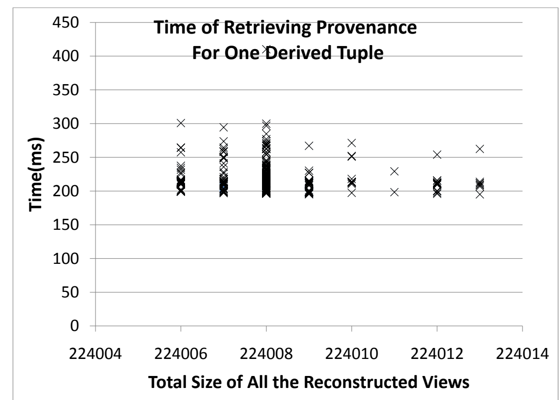

Figure 11: Time of Provenance Retrieval For a Derived Tuple

We can see from Figure 11 that the absolute time of retrieval falls in a range from 200ms to 300ms when the total number of tuples in all the reconstructed views is around 224000. Also, we notice in Figure 11 that for a fixed size of reconstructed views, the retrieval time varies. That is because of the different ratios of historical tuples in the reconstructed views.

To show the relationship between the retrieval time and the ratio of historical tuples in the reconstructed views, we execute the four workloads of the second type shown in Figure 12.

	Workload			
	6	7	8	9
$R^{c\text{-}p}_{CA,HS,LT}$	1	1	1	1
$R^{c\text{-}p}_{ST,T,TH}$				
$W^{t\text{-}o}_{T}$	58	563	1817	2840
$W^{t\text{-}o}_{TH}$				
$W^{t\text{-}r}_{HS}$	58	563	1799	2816
$W^{t\text{-}r}_{TH}$				
$W^{t\text{-}r}_{CA}$	57	519	1648	2577
$W^{t\text{-}r,1}_{T}$	13	100	309	499
$W^{t\text{-}r,2}_{T}$	58	563	1799	2816
$W^{m\text{-}f}_{LT}$	58	563	1799	2816
$W^{m\text{-}f}_{T}$	14	214	698	1197
$W^{m\text{-}f}_{TH}$				
Total	448	4427	14185	22416

Figure 12: Workloads (Continued)

In each of these four workloads, the query $R^{c\text{-}p}_{CA,HS,LT}$ is executed only once and is executed at the beginning of the workload. Since it is executed at the beginning and the initial databases for these four workloads are the same, the reconstructed view of each involved table is the same across these four workloads. In particular, the total number of tuples in the reconstructed views of all the involved tables is 224007.

When retrieving the provenance for a derived tuple by $R^{c\text{-}p}_{CA,HS,LT}$, the more writes following this query, the more historical tuples in the reconstructed views. Therefore, the ninth workload has the highest ratio of historical tuples to the size of the reconstructed views.

The provenance retrieval time for a tuple derived by the query $R^{c\text{-}p}_{CA,HS,LT}$ in each of these four workloads is shown in Figure 13. As can be seen from this figure, the executing time of the extended tracing query grows when the ratio of historical tuples increases.

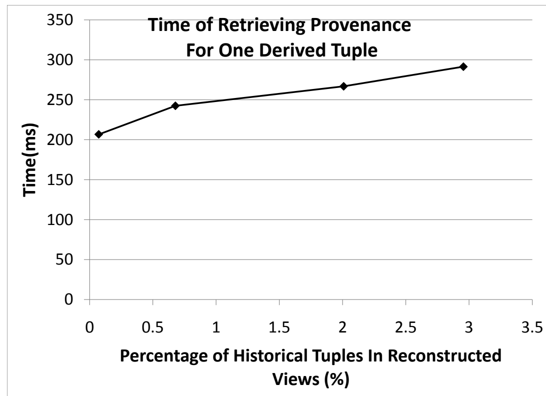

Figure 13: Time Cost of Examining a Provenance Log Entry With Fixed Reconstructed Views

6. RELATED WORK

Provenance provides information about the origin of data. This information can be used in many different ways. For example, in a scientific computing workflow, the origin of data can help to find the cause of errors in the data. Also, the origin of each form on a secure web page can help prevent leaking private information to malicious hackers.

Due to the usefulness of provenance, it has attracted more and more attention. There are quite a few studies on provenance and in particular in database applications [1–6, 8–11].

In general, the provenance of a data item in a database includes the source data items used to derive it and the derivation process. The approaches to the retrieval of these source items can be classified into three categories: inversion based, annotation based, and log based. In an inversion based approach [8], the source data items are located by executing tracing queries, which are constructed based on the original queries and the derived data item. In an annotation based approach [2, 5, 9–11], the source data items are located by referring the annotation of the derived data items, which contains either the identifiers of the source data items or the source data items. In a log based approach [1], every database operation is logged and the source data items or the referring to them are directly recorded in the log.

The inversion based approach is the computation-on-request type, i.e., the provenance is computed only when the user asks for it. It particularly addresses the derivation that is done through SQL queries. On the other hand, it does not provide provenance for a derived data item that is generated through copy-paste operations. The annotation based approach can apply whether the derivation is through SQL type operations or copy-paste type operations. It requires that the annotations be propagated during the execution of derivation operations. This annotation propagation mechanism is not yet a widely supported feature of commercial systems. The log based approach suits the curated databases best. The provenance stored is like a log of operations, and the storage cost can be achieved by removing entries that could be inferred from the other entries as illustrated in [1].

Although much work has been done on database provenance, the effect of in-place update on provenance has not been paid attention until recently [2, 3, 6].

[3] proposed an update language that implicitly propagates color annotations of the objects where the color annotations are a type of provenance representation. These provenance-aware updates propagate the color annotation in a "kind-preserving" way, which means that if a value appears in the output with a given color, then the corresponding value in the input must have the same type (atom, record, or set); furthermore, if the value is an atom, then the corresponding value in the input must be the same atom [2]. For example, if a cell of a tuple is updated by a provenance-aware update operation, the color annotation of the tuple and other cells stay the same while the color annotation of this cell changes. However, kind-preserving is a very weak condition [2]. For example, if the value of every cell in a tuple is replaced by some random value, the tuple still keeps its color. It is not clear that this is a desirable property. Moreover, [3] does not provide for the retrieval of previous values.

Another area of related work to this paper is the work on temporal databases. In a temporal database, the tuple getting replaced is not gone but still stored somewhere, and thus

becomes a historical tuple. Each historical tuple has transaction times associated with it indicating the time period during which the tuple was present in the database. Other ways of archiving historical tuples have been explored too. For example, recent works [16, 17] proposed efficient ways of storing all the previous values and/or previous schemas by archiving the information of previous versions into XML files. In contrast to the focus of provenance approach, temporal databases do not address the derivation relationship that may exist between these historical data items and other data items that are derived from these historical values.

7. CONCLUSION

In this paper, we introduced an approach to the retrieval of provenance (source data items used in a derivation) that is not current in the database. In order to retrieve it, the provenance capture and provenance retrieval process should be aware of the database operations that overwrite existing values in the database. We developed techniques that would maintain the least amount of historical information necessary to reconstruct provenance accurately. We demonstrated experimentally that our techniques result in reasonable costs both in terms of storage overhead and in terms of provenance reconstruction time.

8. REFERENCES

[1] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550, New York, NY, USA, 2006. ACM.

[2] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2008. ACM.

[3] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *In ICDT 2007, number 4353 in Lecture Notes in Computer Science*, pages 209–223. Springer, 2007.

[4] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 523–534, New York, NY, USA, 2009. ACM.

[5] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006, New York, NY, USA, 2008. ACM.

[6] J. Cheney, U. A. Acar, and A. Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

[7] Y. Cui. *Lineage Tracing In Data Warehouses.* PhD thesis, Stanford University, Dec. 2001.

[8] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *In ICDE*, pages 367–378, 1999.

[9] J. N. Foster, T. J. Green, and V. Tannen. Annotated xml: queries and provenance. In M. Lenzerini and D. Lembo, editors, *PODS*, pages 271–280. ACM, 2008.

[10] B. Glavic and G. Alonso. Provenance for nested subqueries. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 982–993, New York, NY, USA, 2009. ACM.

[11] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, New York, NY, USA, 2007. ACM.

[12] http://www.tpc.org/tpce/default.asp. TPC Benchmark^{TM} E (TPC-E).

[13] C. S. Jensen and D. B. Lomet. Transaction timestamping in (temporal) databases. In *In Proceedings of the 27th VLDB Conference*, pages 441–450, 2001.

[14] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

[15] D. Lomet, R. Barga, and R. Wang. Transaction time support inside a database engine. In *In Proceedings of the 22nd ICDE Conference*, 2006.

[16] H. J. Moon, C. A. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, 2008.

[17] F. Wang, C. Zaniolo, and X. Zhou. Archis: an xml-based approach to transaction-time temporal database systems. *The VLDB Journal*, 17(6):1445–1463, 2008.

APPENDIX

A. TRACING QUERIES AND PROVENANCE

In this appendix, we are going to show that the provenance retrieved by the tracing query as shown in Equation 2 is actually the provenance defined in Definition 2.1. If this is shown to be true, then the extended tracing query as shown in Equation 3 can retrieve the provenance defined in Definition 2.1 as well, since the the extended tracing query is essentially a tracing query with the source tables used in the query being some reconstructed historical versions of the source tables.

We first show that the argument is true for the case where there is no aggregation in the original derivation query, and then show the argument is also true where there are aggregations in the target list of the original query.

For clarity, we state our argument using a specific case where there are exactly two source tables. Other cases are similar.

Given the original derivation query being Q as $\{t : \langle A_1, ..., A_m\rangle \mid \exists s_1, s_2\ T_1(s_1) \wedge T_2(s_2) \wedge f(s_1, s_2, t)\}$.

Then the tracing query TQ_1 to find provenance in table T_1 for a given tuple $\bar{t}$ is $\{s_1 : \langle B_1, ..., B_n\rangle \mid \exists s_2, t\ T_1(s_1) \wedge T_2(s_2) \wedge f(s_1, s_2, t) \wedge t = \bar{t}\}$. The tracing query TQ_2 for provenance in table T_2 is similar.

Notice that in this appendix, we use s (s') and t (t') to denote tuple variables; use $\bar{s}$ ($\bar{s}'$) and $\bar{t}$ ($\bar{t}'$) to denote tuple constants.

Assume that $T'1_1$ and T'_2 are the provenance retrieved by the tracing query. In order to show that they are exactly the defined provenance by Definition 2.1, we only need to show

1. $T'_k \subset T_k$ $(k = 1, 2)$

2. $\bar{t} \in Q(T_1', T_2')$
3. $\forall s_k' \in T_k' : Q(T_1', ..., \{s_k'\}, ..., T_2') \neq \varnothing$
4. T_1', T_2' are the maximal subset of their kinds respectively

First, we show $T_k' \subseteq T_k$. We only show here $T_1' \subseteq T_1$. The other case is similar. To show this, we only need to show $\forall s_1' \in T_1' :\ s_1' \in T_1$.

Since $s_1' \in T_1'$, s_1' is an answer tuple to the the tracing query TQ_1. Therefore, the formula in TQ_1 evaluates to true with the assignment of s_1' to s_1. That is, $\exists s_2, t\ T_1(s_1') \wedge T_2(s_2) \wedge f(s_1', s_2, t) \wedge t = \bar{t}$ evaluates to true. That is to say, $T_1(s_1')$ evaluates to true. Since $T_1(s_1')$ evaluates to true, $s_1' \in T_1$. Therefore, $\forall s_1 \in T_1' :\ s_1 \in T_1$. Thus, $T_1' \subset T_1$.

Second, we show $\bar{t} \in Q(T_1', T_2')$.

Since $\bar{t} \in Q(T_1, T_2)$, then the formula in Q evaluates to true with the assignment of $\bar{t}$ to t. That is, $\exists s_1, s_2\ T_1(s_1) \wedge T_2(s_2) \wedge f(s_1, s_2, \bar{t})$ evaluates to true. Therefore, there exists tuples $\bar{s}_1$ and $\bar{s}_2$ such that $T_1(\bar{s}_1) \wedge T_2(\bar{s}_2) \wedge f(\bar{s}_1, \bar{s}_2, \bar{t})$ evaluates to true. This further means, $\exists s_2, t\ T_1(\bar{s}_1) \wedge T_2(s_2) \wedge f(\bar{s}_1, s_2, t) \wedge t = \bar{t}$ evaluates to true. Since this is just the formula in TQ_1 with $\bar{s}_1$ assigned to s_1, therefore, $\bar{s}_1 \in T_1'$. Similarly, $\bar{s}_2 \in T_2'$. Therefore, the formula $T_1'(\bar{s}_1) \wedge T_2'(\bar{s}_2) \wedge f(\bar{s}_1, \bar{s}_2, t) \wedge t = \bar{t}$ evaluates to true. That is to say, $\exists s_1, s_2\ T_1'(s_1) \wedge T_2'(s_2) \wedge f(s_1, s_2, t) \wedge t = \bar{t}$ evaluates to true. This means, we have found an assignment of $\bar{t}$ to t, which makes the formula $\exists s_1, s_2\ T_1'(s_1) \wedge T_2'(s_2) \wedge f(s_1, s_2, t)$ evaluate to true. Since this is just the formula of Q executed on T_1' and T_2'. Thus, $\bar{t} \in Q(T_1', T_2')$.

Third, we show $\forall s_1' \in T_1' :\ \bar{t} \in Q(\{s_1'\}, T_2')$. The other case is similar.

Since $s_1' \in T_1'$, therefore s_1' is an answer tuple to TQ_1. That is to say, the formula in TQ_1 evaluates to true with the assignment of s_1' to s_1. Thus, $\exists s_2, t\ T_1'(s_1') \wedge T_2'(s_2) \wedge f(s_1', s_2, t) \wedge t = \bar{t}$ evaluates to true. This further means, the formula $\exists s_2, t\ T_2'(s_2) \wedge f(s_1', s_2, t) \wedge t = \bar{t}$ evaluates to true. Therefore, the formula $\exists s_1, s_2, t\ s_1 \in \{s_1'\} \wedge T_2'(s_2) \wedge f(s_1, s_2, t) \wedge t = \bar{t}$ evaluates to true. This is just the formula of Q if executed on $\{s_1'\}, T_2'$. Therefore, $\bar{t} \in Q(\{s_1'\}, T_2')$. Thus, $Q(\{s_1'\}, T_2') \neq \varnothing$.

Fourth, we show T_1' is the maximal subset that satisfies the three conditions above. The other case is similar. We show that by contradiction.

Suppose T_1'' and T_2'' are the provenance of $\bar{t}$ as defined in Definition 2.1, and T_1'' is not a subset of T_1'. This means, there exists a tuple $\bar{s_1}$ that is in T_1'' but not in T_1'. According to the third condition and since T_1'' is the provenance, the formula $\exists s_1, s_2, t\ s_1 \in \{\bar{s}_1\} \wedge T_2''(s_2) \wedge f(s_1, s_2, t)$ evaluates to true with the assignment of $\bar{t}$ to t. That is to say, the formula $\exists s_2, t\ T_2''(s_2) \wedge f(\bar{s}_1, s_2, t) \wedge t = \bar{t}$ evaluates to true. Furthermore, since $\bar{s}_1 \in T_1$ due to the first condition, the formula $\exists s_2, t\ T_1(\bar{s}_1) \wedge T_2''(s_2) \wedge f(\bar{s}_1, s_2, t) \wedge t = \bar{t}$ evaluates to true. Since $T_2'' \subseteq T_2$, the formula $\exists s_2, t\ T_1(\bar{s}_1) \wedge T_2(s_2) \wedge f(\bar{s}_1, s_2, t) \wedge t = \bar{t}$ evaluates to true. This formula is just the formula in TQ_1 with the assignment of $\bar{s_1}$ to s_1. Therefore, $\bar{s_1} \in T_1'$. This contradicts with the assumption that $\bar{s_1}$ is not in T_1'. Therefore, T_1' is the maximal subset that satisfies the three above conditions.

So far, we have show that the tracing queries retrieve the defined provenance when there is no aggregations in the original derivation queries. Now assume there is an aggregate attribute in the target list. Then Q is like $\{t : \langle A_1, ..., A_m, G\ AS\ aggr(A_{m+1})\rangle \mid \exists s_1, s_2\ T_1(s_1) \wedge T_2(s_2) \wedge f(s_1, s_2, t.A)\}$, where G is an aggregate attribute, $aggr$ is an aggregate function and $t.A$ is a short hand for $t.A_1, ..., t.A_m$.

Suppose we have another query Q', which is $\{t : \langle A_1, ..., A_m\rangle \mid \exists s_1, s_2\ T_1(s_1) \wedge T_2(s_2) \wedge f(s_1, s_2, t.A)\}$. The only difference between Q and Q' is that Q' does not have the aggregate attribute G in the target list.

An observation on Q and Q' is that they have the same tracing queries, e.g., both have a TQ_1 being $\{s_1 : \langle B_1, ..., B_n\rangle \mid \exists s_2, t\ T_1(s_1) \wedge T_2(s_2) \wedge f(s_1, s_1, t) \wedge t.A = \bar{t}.A\}$.

Given two tuples, $t_{agg} : \langle a_1, ..., a_m, g\rangle$ from Q and $t : \langle a_1, ..., a_m\rangle$ from Q', if we can show that

1. these two tuples have the same provenance according to Definition 2.1,
2. and they also have the same provenance retrieved by the tracing query in Equation 2,
3. and the provenance of t retrieved by the tracing query matches the provenance of t defined by Definition 2.1,

then we can say that the provenance of t_{agg} retrieved by the tracing query matches the provenance of t_{agg} defined by Definition 2.1.

The third of the above is obvious since Q' is an aggregate-free query. In the case of aggregate-free queries, we have shown that the provenance retrieved by the tracing query is exactly the provenance defined by Definition 2.1.

The second of the above is also obvious since Q and Q' have the same tracing queries.

Thus, we only need to show that the provenance of t_{agg} is the same as that of t according to Definition 2.1. For t_{agg} : $\langle a_1, ..., a_m, g\rangle$, there exists a group of tuples $\langle a_1, ..., a_m, g_1\rangle$, ..., $\langle a_1, ..., a_m, g_k\rangle$ such that g is $aggr(g_1, ..., g_k)$. Since the formula parts of Q and Q' are the same, this same group of tuples are projected into t if Q' is executed. Therefore, according to the transitivity of the defined provenance, the provenance of t_{agg} and t are both the provenance of these group of tuples. Thus, the defined provenance of t_{agg} and t are the same.

Therefore, the provenance retrieved by the tracing query in Equation 2 is the provenance defined in Definition 2.1.

B. AGGREGATIONS IN FORMULA

We only allow aggregate functions/attributes in the target list. In Section 2, we claim that if an aggregate function/attribute appears in the formula part of a query, this query can decomposed into two formulas such that none of them has aggregations in the formula.

We give a simple example first. Assume we have a table T that has two attributes A_1 and A_2. We group on A_1, then compute the average of values in A_2, and finally select the value of A_1 and the average of the group if the average is bigger than 2. Thus, the query is of the form $\{t : \langle A_1, G\ AS\ aggr(A_2)\rangle \mid \exists s\ T(s) \wedge aggr(s.A_2) > 2 \wedge s = t\}$.

This query can be decomposed into two queries. The first query is the original one except for the atomic formula involving aggregations. The second uses the output of the first query and applies the atomic formula that is left out in the first query.

Thus, the first query is $\{t : \langle A_1, G\ AS\ aggr(A_2)\rangle \mid \exists s\ T(s) \wedge s = t\}$. Assume the result is stored in table T'. Then the second query is $\{t : \langle A_1, G\rangle \mid \exists s\ T'(s) \wedge s.G > 2 \wedge s = t\}$. Thus, there are no more aggregate functions in the formulas.

In general, assume a query that has an aggregate function in the formula is of the form $\{t : \langle A_1, ..., A_m\rangle \mid \exists s_1, ..., s_n\ T_1(s_1) \wedge$

$... \wedge T_n(s_n) \wedge f(s_1, ..., s_n, t) \wedge f'(aggr(A_{m+1}))\}$, where $f'(aggr(A_{m+1}))$ is an atomic formula specifying a condition on the aggregate value resulting from the application of the aggregate function $aggr$ to the attribute A_{m+1}.

Then this query can be decomposed into two queries.The first is $\{t : \langle A_1, ..., A_m, G\ AS\ aggr(A_{m+1})\rangle \mid \exists s_1, ..., s_n\ T_1(s_1) \wedge ... \wedge T_n(s_n) \wedge f(s_1, ..., s_n, t)\}$. Assume the result is stored in S. The second is $\{t : \langle A_1, ..., A_m\rangle \mid \exists s\ S(s) \wedge f'(s.G) \wedge s = t\}$.

More complex cases, such as multiple aggregate functions in the formula, can be treated similarly. We do not detail on them here.