PROVENANCE IN MODIFIABLE DATASETS

by

Jing Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2012

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Provenance In Modifiable Datasets

by
Jing Zhang

Chair: H.V. Jagadish

The provenance of derived data, which explains the derivation and retrieves or captures the source data, is valuable information for the data consumers possibly due to different purposes, e.g., audit requirements, error tracing, data reproduction and etc. The provenance of a derived datum should include all the details about how it is derived, including in particular, the source data used in its derivation. The provenance of a derived datum can be recorded during the original derivation process but storing it explicitly can incur very high storage cost. Therefore, techniques have been developed to record only a small amount of information, which can be used later to retrieve the full provenance from the source dataset. Such provenance retrieval relies on the provenance being present in the dataset in order to be retrieved by tracing queries. However, many datasets are subject to modifications, e.g, new experimental data is collected and stored.

In this thesis, we investigate the retrieval of the provenance of a derived datum from a modifiable dataset, specifically we consider the following four questions:

(i). Can we explain what a particular derived datum depends on, even if a value used in its derivation has since been modified.

(ii). Can we determine if a particular derived datum is still valid upon the source dataset modifications without performing full view maintenance but through examining its provenance.

(iii). Can we retrieve part of the provenance of a given datum due to the users' request or the fact that the rest of the provenance is missing.

(iv). Can we retrieve the provenance of a derived datum without predefined granularity in an unstructured dataset.

In this thesis, we provide affirmative answers to the above questions in the form of new techniques that use limited space and computational effort.

CHAPTER I

Introduction

Provenance has conventionally been used to describe the history or origin of a work of art. In the computer science universe, provenance refers to the history or origin of digital objects. These digital objects can be (1) workflows and/or data in a workflow management system, (2) data in a database management system, (3) network meta-data of a datum in a network, etc. In this thesis, we focus on the data provenance in database applications.

Database applications are widespread. The ease of storing, querying, and manipulating data through database management systems has led to an explosion of the usage of databases. Once data comes into being in a database, it can be used to derive more data. The derivations can serve many different purposes, e.g., reformatting, filtering out irrelevant information, aggregating a big volume of data into a small volume for better human comprehension, etc. We are all consumers of derived data when we check hotel ratings, visit price comparison sites, and even when we read blogs that rely on some data or statistics.

A consumer of a derived datum may request the provenance of the derived datum for one or more reasons, e.g., determining the reliability of the derived datum, debugging the derivation process, meeting the audit requirements, etc. The provenance of a derived datum includes “where” the datum comes from and “why” it is produced from the derivation process [5]. This provenance can be computed in an eager manner or a lazy manner [35].

In a lazy manner, the “why” provenance can be retrieved through tracing queries [12, 16]. Thus the provenance computation is decoupled from the original query evaluation and there is a need neither for storage nor for rewriting the original query. However, this approach needs to handle modifications of the source data [40], i.e., insertion, deletion, updates, after the derivation. This is because the modifications after the derivation of a given derived datum may remove the provenance of the given datum or invalidate it.

An eager approach usually couples the provenance computation with the original query evaluation. One type of eager approach rewrites the original query to produce the provenance together with result [15, 16]. Another type of eager approach propagates annotations during the evaluation of the derivation query [18, 14]. A third type of eager approach is targeted at a curated database and records the curation operations as provenance [2]. The storage overhead of eager approaches will become intractable if more and more derived data keep being generated. In [8], provenance factorization is proposed to reduce provenance size.

Studying provenance in the presence of modifications to the source datasets is important since modifying existing datasets is a rather common practice, e.g., new experimental results being inserted, existing figures being corrected, erroneous facts being deleted, etc. A single modification to a dataset can affect part of the dataset or the complete dataset. Multiple modifications can be applied sequentially to the same dataset. In this thesis, we focus on the study of provenance retrieval and usage in modifiable datasets.

Moreover, modifiable datasets can be structured, unstructured or semi-structured. Typical structured datasets include relational databases and object oriented databases. Typical unstructured datasets include plain text documents, audios and videos. Unstructured datasets can be transformed to semi-structured ones, e.g., by attaching tags to plain text documents. Typical semi-structured datasets are XML files. In this thesis, we study the provenance retrieval and usage in modifiable relational databases and plain text documents.

Due to the different characteristics of relational databases and text documents, some provenance questions may be more interesting in one scenario than in the other or vice versa. In particular, given a set of modifications in chronological order that have taken place to the source dataset after the derivation of a derived datum, we need to answer the following provenance related questions.

(i). In case of relational databases, the following questions are applicable.

- (a) What is the provenance of a given derived datum, even when (part of) the provenance is not in the current dataset?
- (b) Is it possible to compute one part of the provenance independent of the rest, which may or may not be in the current dataset? When is it possible and how to compute?
- (c) Is a given derived datum affected by the set of modifications? How is the given derived datum affected?

(ii). In case of plain text documents, the following questions are applicable.

- (a) What modifications in the revision log have actually affected a selected piece of text?
- (b) What is the evolution of a selected piece of text, while the sub-pieces may evolve independently and the ranges of the sub-pieces may change dynamically?

The question ((i)a) requires the retrieval of the lost provenance due to the modifications. The classical tracing queries are adjusted to take into consideration the log of modifications and the archived historical data. This approach is discussed in Chapter II.

The question ((i)b) requires the retrieval of partial provenance independently. The classical tracing queries are rewritten to remove the references to certain source tables when those source tables satisfy some specific requirements. This approach is discussed in Chapter III.

The two questions in ((i)c) require determining the effect of modifications on some selected derived data in the result set. Instead of performing a view maintenance over the entire result set, validating the selected data can be more efficiently done through the usage of pruning predicates. This approach is discussed in Chapter IV.

The questions in ((ii)a) and ((ii)b) require retrieving relevant revisions to the selected text piece in the revision history/log of a text document. Due to the lack of a predefined structure, we can adjust the ranges of the text pieces in the document according to the revision history and according to the user's request, and therefore prune away irrelevant revisions from the provenance of the selected text piece. This approach is discussed in Chapter V.

After the discussion in the above mentioned four chapters, we conclude our work and introduce further work in Chapter VI. Moreover, the background knowledge that is needed to understand the approaches in this thesis is in the appendices.

CHAPTER II

Retrieval Of Lost Source Provenance

2.1 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 can we 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 2.1. Consider a simple source database comprising two tables[1] *Book* and *Price* as shown in Figure 2.1 and Figure 2.2 respectively. A query Q_{bb} is shown in Figure 2.4, 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 2.3. After the query

[1]The attribute named *since* in the tables is reserved for our provenance approach. Its meaning will be explained in Section 2.2.

Q_{bb} is executed, the update U_p (Figure 2.4) on the table *Price* is executed, which increases the price of any book 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 2.1: **Book**

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

Figure 2.2: **Price**

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

Figure 2.3: **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 2.4: Two Database Operations

Taking the tuple ⟨“A Brief History of Time”, 10⟩ as an example. Its source tuple in the table *Price* is $\langle 0553380168, 10\rangle$, which is no longer in the table *Price* after the two operations in Figure 2.4 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 *BargainBook* table requires an understanding of its historical price.

Existing techniques of provenance retrieval, such as tracing queries [11] or propagating the identifiers of the source rows as annotations [18], 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 chapter, 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 chapter is organized as follows: Section 2.2 is the background information about database provenance and historical data; Section 2.3 gives the overview of our approach and introduces the necessary data structures required by our approach; Section 2.4 describes the provenance reconstruction algorithm in detail and analyzes its time and space costs; Section 2.5 shows the experimental results of these costs in a realistic setting; finally, Section 2.6 and Section 2.7 are the related work and the conclusion respectively.

2.2 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.

2.2.1 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 chapter, we only consider conjunctive queries and conjunctive queries with aggregations, i.e., SPJ queries and 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 A.2. In fact, this decomposition corresponds to the decomposition of an aggregate query into canonical segments in [11]. 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.

2.2.2 Provenance of Tuples

The provenance of a given tuple can be defined in many different ways. In this chapter, we adopt the definition from [11]. We restate the definition here.

Definition 2.2. [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 have tables $T'_1, ..., T'_n$, where $T'_1, ..., T'_n$ are the **maximal** subsets of $T_1, ..., T_n$ such that:

(i). $\{t\} = Q(T'_1, ..., T'_n)$

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

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 chapter, we assume set semantics. Under set semantics, it has been proven in [11] that the provenance defined above for a given tuple is unique.

2.2.3 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 2.1, 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 chapter. 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.

2.3 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.

2.3.1 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 greater ID for its corresponding log entry. That is to say, the ID indicates the order of the commission 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 2.1. The provenance log after the execution of the two operations Q_{bb} and U_p is shown in Figure 2.5.

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 2.5: Provenance Log Example

2.3.2 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 a same log entry ID.

Recall Example 2.1. After the update U_p, the table *Price* is like the one shown in Figure 2.7; and its shadow table $Price_{sh}$ is shown in Figure 2.6.

ISBN	Price	begin	end
0553380168	10	0	2

Figure 2.6: Shadow Table $Price_{sh}$ After U_p

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

Figure 2.7: Table *Price* After U_p

2.3.3 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 chapter, we assume that this annotation attribute *since* is already included in each table in the database schema.

2.3.4 Populating Auxiliary Data Structures

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

(i). When a database operation takes place, a new entry is created in the provenance log and a unique ID is assigned 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.

(ii). 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.

(iii). 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.

2.4 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 [11] is able to retrieve the provenance defined in Definition 2.2, 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 [11]. 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.

2.4.1 Tracing Query Revisited

In [11], the tracing queries and the original queries are expressed in a relational algebra extended with aggregations. In this chapter, we use Tuple Relational Calculus (TRC) instead, which can be extended with aggregations as well [25]. Moreover, this extended

TRC is equivalent to the extended relational algebra [25].

Recall the example query Q_{bb} shown in Figure 2.4. The SQL statement of Q_{bb} can be expressed in TRC as follows:

$$\begin{aligned}\Big\{\ & t : \langle Title, Price\rangle \ \mid \\ & \exists s_1 : \langle ISBN, Title\rangle, s_2 : \langle ISBN, Price\rangle \\ & (Book(s_1) \wedge Price(s_2) \\ & \wedge s_1.ISBN = s_2.ISBN \wedge s_2.Price <= 10 \\ & \wedge t.Title = s_1.Title \wedge t.Price = s_2.Price)\ \Big\}\end{aligned}$$

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:

$$\left\{ \begin{array}{l} \underline{s_1 : \langle ISBN, Title\rangle} \mid \\ \exists s_2 : \langle ISBN, Price\rangle, \underline{t : \langle Title, Price\rangle} \\ (Book(s_1) \wedge Price(s_2) \\ \wedge s_1.ISBN = s_2.ISBN \wedge s_2.Price <= 10 \\ \wedge t.Title = s_1.Title \wedge t.Price = s_2.Price \\ \wedge \underline{t.Title = \text{“A Brief History of Time''}} \wedge \underline{t.Price = 10}) \end{array} \right\}$$

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

(i). 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.

(ii). 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

$$\left\{ \begin{array}{l} t : \langle A_1, ..., A_n, G\ AS\ \ agg(A_{n+1})\rangle \mid \\ \exists s_1, ..., s_m \\ (T_1(s_1) \wedge ... \wedge T_m(s_m) \wedge f(s_1, ..., s_m, t)) \end{array} \right\} \tag{2.3}$$

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 $B_1, ..., B_l$,

$$\left\{ \begin{array}{l} s_k : \langle B_1, ..., B_l\rangle \mid \\ \exists t, s_1, ..., s_{k-1}, s_{k+1}, ..., s_m \\ (T_1(s_1) \wedge ... \wedge T_m(s_m) \wedge f(s_1, ..., s_m, t) \\ \wedge t.A_1 = a_1 \wedge ... \wedge t.A_n = a_n) \end{array} \right\} \tag{2.4}$$

We show in Appendix A.1 that this tracing query can retrieve the provenance defined in Definition 2.2.

2.4.2 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 2.4, the extended tracing query need an extra piece of information, i.e., the ID of the provenance log entry that records the the original query.

Recall our discussion in Section 2.3.2. 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 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

$$\begin{aligned}
\Big\{ \quad & s_2 : \langle ISBN, Price\rangle \ \mid \\
& \exists s_1 : \langle ISBN, Title\rangle, t : \langle Title, Price\rangle \\
& (\underline{Book^H(s_1)} \wedge \underline{Price^H(s_2)} \\
& \wedge s_1.ISBN = s_2.ISBN \wedge s_2.Price <= 10 \\
& \wedge s_1.ISBN = t.ISBN \wedge s_2.Price = t.Price \\
& \wedge t.Title = \text{“A Brief History of Time''} \wedge t.Price = 10 \quad \Big\}
\end{aligned}$$

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:

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

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:

(i). the derived tuple

(ii). the original query

(iii). 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 2.3, then the extended tracing query to retrieve provenance in the table T_k is

$$\begin{aligned} \{ \quad & s_k : \langle B_1, ..., B_l \rangle \ \mid \\ & \exists t, s_1, ..., s_{k-1}, s_{k+1}, ..., s_m \\ & (T_1^H(s_1) \wedge ... \wedge T_m^H(s_m) \wedge f(s_1, ..., s_m, t) \\ & \wedge t.A_1 = a_1 \wedge ... \wedge t.A_n = a_n) \quad \} \end{aligned} \tag{2.5}$$

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

$$\begin{aligned} \{ \quad & s_k : \langle B_1, ..., B_l \rangle \ \mid \\ & (T_k(s_k) \wedge s_k.since < id) \vee \\ & \exists s'_k (T_{k\_sh}(s'_k) \wedge s'_k.begin < id \wedge s'_k.end >= id \\ & \wedge s'_k.B_1 = s_k.B_1 \wedge ... \wedge s'_k.B_l = s_k.B_l) \quad \} \end{aligned} \tag{2.6}$$

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 2.6.

2.4.3 Analysis of Efficiency of Extended Tracing Queries

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

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 decreasing the space cost also means an increase 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 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 2.6.

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.

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 2.5. The cause of this is probably 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.

2.5 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 [22] 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.

2.5.1 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.

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 [22]. 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.

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 2.8, 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 2.8, has a corresponding shadow table.

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

Figure 2.8: Reads and Writes of Tables in Transactions

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 2.13).

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.

(i). we take a workload generated by the CEE class in EGen package;

(ii). keep only the transactions of trade_order;

(iii). 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.

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.

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 chapter 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" [28]; or queried via XQuery, e.g., [37]. 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 [24]. 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.

2.5.2 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.

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 2.9. 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 2.10. 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 2.10, 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 2.9: Workloads

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

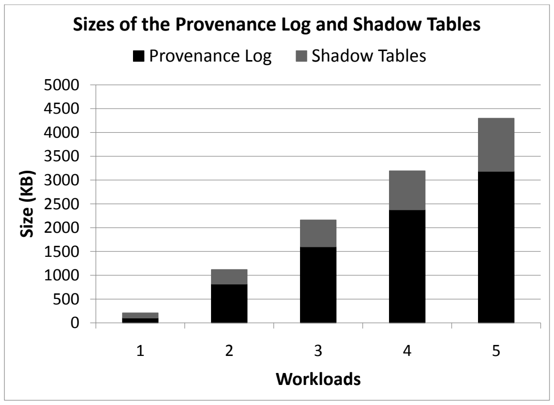


Figure 2.10: Sizes of the Provenance Log and Shadow Tables

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 2.9. In this workload, the query $R^{\text{c-p}}_{\text{CA,HS,LT}}$ 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 2.11. 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.

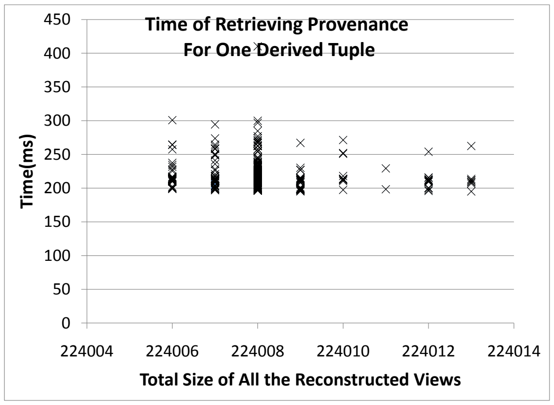


Figure 2.11: Time of Provenance Retrieval For a Derived Tuple

We can see from Figure 2.11 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 2.11 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 2.12.

	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 2.12: 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 2.13. As can be seen from this figure, the executing time of the extended tracing query grows when the ratio of historical tuples increases.

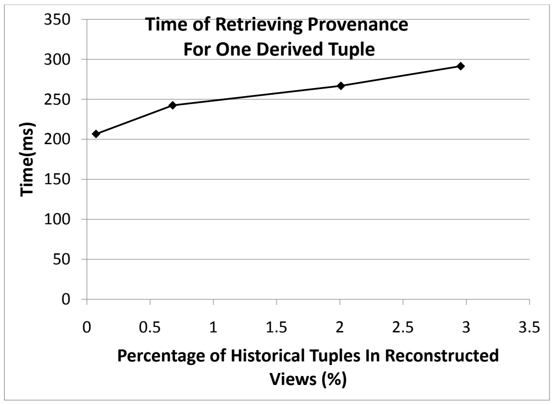


Figure 2.13: Time Cost of Examining a Provenance Log Entry With Fixed Reconstructed Views

2.6 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 [4][12][18][3][9][16][14][2][7][8].

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 [12], 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 [16][18][14][8][3], 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 [2], 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 [2].

Although much work has been done on database provenance, the effect of in-place update on provenance has not attracted attention until recently [4][3][9].

[4] 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 [3]. For example, if a cell of a tuple is updated by a provenance-aware update operation, the color annotation of the tuple and all other cells stay the same while the color annotation of this cell changes.

However, kind-preserving is a very weak condition [3]. 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, [4] does not provide for the retrieval of previous values.

Our work addresses the problem of retrieving the provenance no longer present in the database due to the in-place update with regard to the inversion based approaches. The typical inversion based approaches retrieve the provenance through tracing queries. If the provenance is not currently in the database, the tracing queries are not able to find it. Thus, additional information are needed to recover the proper historical values and the classical tracing queries need to be modified to use those additional information.

Another area of related work to this chapter 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 them 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 [37][31] 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.

2.7 Conclusion

In this chapter, 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.

CHAPTER III

Customization of Provenance

3.1 Introduction

The provenance of a given result tuple intuitively includes all the source tuples that contributed to its derivation. This provenance is usually examined by a human for some purpose, e.g., locating error causes, validating the result, acquiring missing information in the result, etc.

The basic provenance request is to find all the provenance information of a given result tuple (or tuples). However, this one-size-fits-all solution has limitations. When users have a particular purpose for which they want provenance information, the complete provenance can contain a great deal of additional irrelevant information. For example, the user need may be for the intersection of the provenance of two result tuples derived from different queries, rather than for the complete provenance of both. For another example, the user may be interested in finding the provenance of a result tuple within a specific source table with a particular attribute less than some value: provenance information regarding other source tables, or with larger values of this attribute, are not of interest.

These observations lead to the need to define customized provenance, determined based on the user need. The customized provenance of a given result tuple is usually a subset of the complete provenance. That suggests a simple two-stage approach of computing the customized provenance: first computing the complete provenance and then filtering it using the specific criterion of the customized provenance request. However, this basic approach involves unnecessary computation of some (and often, a great deal of) provenance information that is discarded in the second stage. To illustrate this problem consider the following (greatly simplified) example.

Example 3.1. We have three database tables *Orders*, *Categories* and *Ratings* as shown in Figure 3.1a, with the primary keys underlined. We want to find the revenue of every category including only the products that are either iPhones or are positively rated, where a product is considered positively rated if its average rating is higher than 3.

A possible query for this request is shown in Figure 3.1b in both SQL and Datalog. The corresponding predicate dependency graph[1] [38] built from the set of Datalog rules is shown in Figure 3.1d. That query first computes the average rating of each product that has been rated, and filters on the average ratings, and computes the revenue for each category by summing up the orders placed for the products with qualified average ratings and the orders placed for iPhones.

Suppose we want to retrieve the complete provenance of the result tuple {SmartPhone, 90}. We can issue appropriate tracing queries to determine the tuples in the source tables *Categories*, *Orders* and *Ratings* on which this result tuple depends, denoted as $Categories^p$, $Orders^p$ and $Ratings^p$. The method prescribed in [11] is illustrated in Figure 3.2 as rules r_4 through r_8.

The rule set of r_4 through r_8 involves two tracing steps: (1) find the provenance of {SmartPhone, 90} in $Orders^p\_Products^p\_Categories^p$, which can then be projected to the schema of *Orders*, the schema of *Products* and the schema of *Categories* to get $Orders^p$,

[1]For a set of Datalog rules, a predicate dependency graph can be built. In that graph, every node corresponds to a predicate. An edge from node p to node q means q is the head of a rule whose body contains p. Notice that some literature defines the edge in the opposite direction.

$Products^p$, $Categories^p$; and (2) find in $Ratings$ the provenance of $Products^p$.

Now suppose that the user wants to customize the provenance. The user only wants to study the iPhone share in the calculated revenue “90” if there is such a share. In other words, the user wants the orders placed on iPhone and included in the returned revenue “90”. Then, the baseline approach is to execute r_4 and r_5 and filter $Orders^p$ with the customization request, shown as the option #1 in Figure 3.3. However, a quick observation over the derivation of $Products$ will tell us that the predicate $Products(\text{“}iPhone'')$ always evaluates to true. Therefore, given the specific customization of the user, we can get rid of the predicate $Products$ in the rules of the provenance retrieval. Thus, we end up with option #2 in Figure 3.3 as tracing queries. In fact, the baseline approach, i.e., option #1, can be transformed into option #2 through three rewritings as shown in Figure 3.3b.

In summary, we have an opportunity to optimize the retrieval of customized provenance. Some of the potential optimizations may be found by general purpose optimization methods. However, many can not, since they request application-specific knowledge. Our goal in this chapter is to develop a systematic technique to find potential optimizations that are tuned for tracing queries.

A special characteristic of tracing queries is that the desired result, at each step, is a subset of a source or intermediate table. When written as a Datalog program, each rule in the tracing query has as the predicate in the head is a “provenance-restriction” of one of the predicates in the body. The restriction is provided by the other predicates in the body of the rule. Selective provenance requests add additional predicates to the body of one or more rules in the Datalog representation of the tracing query. Our challenge in this chapter is to recognize this special structure of tracing queries and the restriction predicates imposed on them.

We propose an optimization algorithm based on rewriting rules to solve this problem.

In particular, two types of rewriting are used: (1) substitution of a predicate for a rule, (2) removal of a predicate.

The benefits that result from our optimization of the provenance retrieval include: (1) less predicates in a Datalog rule (in other words, less joins in the query evaluation); (2) elimination of unnecessary access to intermediate tables (in other words, either elimination of the recomputation of intermediate tables during retrieval or elimination of storage of intermediate tables); (3) elimination of unnecessary retrieval of provenance inside intermediate tables.

Whereas we developed our techniques for customized provenance, it turns out that most of them are applicable to all tracing query computations. Doing so requires no modifications of the optimization techniques at all, since a regular tracing query is simply a special case of a customization that does nothing. The extent of actual benefit derived depends on the specific tracing query.

The rest of this chapter is organized as follows. In Section 3.2, we review the provenance definition and the classical tracing queries used to retrieve it; and then we formalize the representation of customized provenance as database views defined by modified tracing queries. In Section 3.3, we develop a set of query rewriting rules that are aimed at simplifying the tracing queries. In Section 3.4, we show how these rules are applied to a Datalog program. In Section 3.5, we show experiments on customized provenance retrieval that compare the time costs of baseline tracing query approach with the time costs of the optimized tracing query approach. Finally, in Section 3.6 and in Section 3.7 respectively, we discuss the related work and conclude our work.

3.2 Problem Set Up and Background

3.2.1 Customization of Provenance

Users may have criteria on what provenance should be returned to them. Their specific requests are usually based on their knowledge of the answer, the query and even the underlying database. Recall Example 3.1. The user knows that iPhone belongs to the smart phone category, and he then wonders about the iPhone share in the smart phone revenue returned by the example query.

More complex customization is possible. For example, suppose there are two answer tuples produced through two different queries respectively. If we know one of the tuples is wrong and have been able to locate the erroneous source data inside its provenance, we may request the common provenance of the two answer tuples to check if the other answer tuple used the erroneous data in its derivation. Note that we are not interested in the full provenance for the second tuple, but rather in its intersection with the provenance of the first tuple.

Traditional provenance management systems work hard to compute, and possibly store, provenance information. However, there is not much need for a provenance query facility: a user merely has to click on a data item to get its full provenance. Now, with provenance customization, we would like to give users precisely what they want, rather than the whole thing. But this requires that we know what they want – that is, that the users have expressed their customization desires to the system. In short, we need a provenance customization specification language.

Fortunately, provenance is also just data, and in fact, structured data. Therefore, provenance is easily queried through query languages such as SQL or its equivalent. Thus, the customization of provenance is simple. Using query languages to customize provenance may be difficult for users who do not have familiarity a query language. However, this is a problem that is not unique to provenance customization. Rather, it applies broadly to most stores of structured data. As such, there has been much work done to develop database tools that help users to query data with form-based or keyword-based interfaces. Such tools can also be used to customize provenance. In this chapter, we do not worry about these specification details. Rather, we examine how to evaluate customized provenance efficiently.

SQL is the query language of choice for commercial database products. However, Datalog is often the preferred language for development of complex algebraic techniques. It is straightforward to translate Datalog programs into SQL. For these reasons, in this chapter we use non-recursive Datalog programs consisting of a set of safe and non-negated Datalog rules. A Datalog rule is safe if every variable that appears in the head of rule also appears in some non-negated relational predicate in the body of the rule. A *relational* predicate is a predicate that represents relations/tables or views in the database. Another type of predicates is *arithmetic* predicates, such as $A > 3$ and $mod(A, B, 3)$, which is true if and only if $A\%B = 3$.

3.2.2 Definitions

Before we formally start to discuss the customization of provenance and the rewriting rules to optimize the retrieval of the customized provenance, we need to review the definition of provenance and the classical tracing queries.

Definition 3.2. [SPJ Tuple Provenance] Suppose there is an SPJ query Q as follows:

$$T(Y) :- S_1(X_1), ..., S_m(X_m)$$

Then, given $t \in T$, a table D is the provenance of t according to Q if and only if

(i). D is of the schema $\cup_{i=1}^{i=m} X_i$

(ii). $\forall d \in D\ t.Y = d.Y$

(iii). $\forall d \in D$, $S_i(X_i)[X_i/d.X_i]$ $(i = 1, ..., m)$ evaluates to true, where $S_i(X_i)[X_i/d.X_i]$ represents a ground $S_i(X_i)$ by substituting X_i for the value $d.X_i$

(iv). $\nexists D' \subset D$ such that D' satisfy the above requirements

In Definition 3.2, every d in D uniquely specify a mapping from the variables in the body of Q to values.

Definition 3.3. [ASPJ Tuple Provenance] Suppose there is an ASPJ query Q as follows:

$$T(G, \Sigma(\langle A \rangle)) :- S_1(X_1), ..., S_m(X_m)$$

where the notation $\langle A \rangle$ indicates a grouping, with G being the GROUP-BY attribute, and $\Sigma \in \{sum, count, avg, max, min\}$ being the aggregate function. Then, given $t \in T$, a table $D = \{d_1, ..., d_n\}$ is the provenance of t according to Q if and only if

(i). D is of the schema $\cup_{i=1}^{i=m} X_i$

(ii). $t.G = d_j.G$ for $j = 1, ..., n$

(iii). $t.A = \Sigma(d_1.A, ..., d_n.A)$

(iv). $S_i(X_i)[X_i/d_j.X_i]$ evaluates to true, for $i = 1, ..., m$ and $j = 1, ..., n$

(v). $\nexists D' \subset D$ such that D' satisfies all the above requirements

Definition 3.4. [Query Result Provenance] Suppose there is a query Q and further suppose $T = \{t_1, ..., t_k\}$ is the (sub)set of the result of Q over tables $S_1, ..., S_m$. Then, the provenance of T is the union of the provenances of t_i for $i = 1, ..., k$.

3.2.3 Tracing Queries

Tracing queries are a standard technique for determining the provenance of a desired result. To find the provenance according to an SPJ query, we have:

$$P(\cup_{i=1}^{i=m} X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{t.Y = Y} \tag{3.5}$$

where t is an answer tuple; and more generally

(3.6) $$P(\cup_{i=1}^{i=m} X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{T'(Y)}$$

where T' is the subset of the answer set. Moreover, if we want the provenance inside a table S_i, we have:

(3.7) $$S_i^p(X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{t.Y = Y}$$

and

(3.8) $$S_i^p(X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{T'(Y)}$$

The bold parts in the equations above are predicates related to the provenance retrieval given the answer tuple(s), e.g., r_4 in Figure 3.2.

To find the provenance according to an ASPJ query, we have:

(3.9) $$P(\cup_{i=1}^{i=m} X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{t.G = G}$$

where t is an answer tuple; and more generally

(3.10) $$P(\cup_{i=1}^{i=m} X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{T'(G)}$$

where T' is the subset of the answer set projected on the attribute list G. Moreover, to have the provenance inside S_i, we have

(3.11) $$S_i^p(X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{t.G = G}$$

and

(3.12) $$S_i^p(X_i) :- S_1(X_1), ..., S_m(X_m), \boldsymbol{T'(G)}$$

When a query involves more than one rule, i.e., a predicate in the body of a Datalog rule is the head of another rule, the provenance retrieval is done recursively as discussed in [12].

3.3 Rewriting Rules For Optimization

Our optimization method is rewriting based. Thus, our optimization method consists of a set of rewriting/optimization rules. (When there is a possibility of confusing Datalog rules with rewriting rules, we will use a modifier “Datalog” or “rewriting/optimization” before “rule”. Otherwise, we will use just “rule” and its meaning will be clear from context.)

Each rewriting rule consists of two parts. One part specifies requirements on the predicates in a Datalog rule. The other part specifies a transformation of a Datalog rule.

To apply an optimization rule to a Datalog rule, we check if the predicates in the Datalog rule satisfy the conditions specified in the optimization rule. If they do, then the Datalog rule can be rewritten according to the optimization rule.

In order to list our rewriting rules for optimization, we have to discuss two things first: (1) the types of predicates in the body of a Datalog rule as part of a tracing query, and (2) the types of transformation operations used in transforming the rule body.

3.3.1 Types Of Predicates In Tracing Queries

In general, given a Datalog rule as part of a classical tracing query, we have two types of predicates. If the tracing query is for customized provenance, there is a third type of predicate.

The first type consists of the predicates that appear in the original query, e.g., $Orders$, $Products$ and $Categories$ in r_4.

The second type consists of the predicates related to the provenance retrieval and determined by the given answer tuple(s), e.g., $caid$ = SmartPhone, where the attribute and the value in the comparison are determined by the answer tuple {SmartPhone, 90}.

The provenance related predicates are meant to narrow down the source tuples used in the original query evaluation to only the ones particularly contributing to the given answer tuple(s). The provenance related predicates can be written in the form of a relational predicate $T(Y)$, where T is a temporary table holding the set of selected answer tuple(s) and Y is the attribute list of the answer tuple, e.g., $T(caid, rev) = \{\{\text{SmartPhone}, 90\}\}$ if the provenance of {SmartPhone,90} is requested; or $T(caid, rev)$=$\{\{\text{SmartPhone}, 90\}$, $\{\{\text{NormalPhone}, 100\}\}$ if the provenance of {SmartPhone, 90} and {NormalPhone, 100} is requested. ({NormalPhone, 100} is not in the result of Example 3.1 because of the fraction of data shown in the example.)

When a user specifies the customization of the provenance, we have a third type of predicates in the tracing queries, i.e., the customization related predicates, e.g., pid = iPhone in r_9. The customization related predicates can be relational predicates or arithmetic predicates. For example, $Orders^p_{cust}$:– $Orders^p(oid, pid, amt, prc), amt * prc > 10$. In particular, when the customization related predicate is relational, it is no different than a provenance related predicate in effect. Therefore, from here on, when we refer to provenance related predicates, we actually mean provenance related predicates and relational customization related predicates.

When the provenance/customization related predicates are present in the same rule as the original predicates, the provenance/customization related predicates can sometime render the original predicates redundant by being more restrictive than the original predicates. In Figure 3.3b, $Products^p$ is removed because $pid = iPhone$ is more restrictive than it.

The optimization rule will explore the relative restrictiveness of the provenance/customization related predicates and the original predicates. We will discuss how to determine their relative restrictiveness when we discuss the optimization rules.

3.3.2 Types of Transformation Operations

In Figure 3.3b, the baseline approach, i.e., Option #1, is transformed into Option #2, and Option #2 is obviously simpler and more efficient in terms of computation.

There are two rewriting operations in the transformation from Option #1 to Option #2: (1) substituting a subgoal for the body of a rule whose head is this subgoal, e.g., the first rewriting step in Figure 3.3b; (2) removing a predicate, e.g., the second and the third rewriting steps in Figure 3.3b. Moreover, there is a third operation that is not used in the example: moving arithmetic predicates.

For the first rewriting operation, it can be done if there are no conflicts in the variable naming. In this chapter, we assume that all the variables in a Datalog program are named properly to avoid naming conflicts.

For the second rewriting operation, a predicate can be removed if there is at least one predicate in the same rule body that is more restrictive than it. In the context of tracing queries, a provenance related predicate is usually more restrictive than original predicates, since the purpose of the provenance related predicate is to narrow down the source tuples to only the ones contributing to the given answer tuples. In the following subsection, we will show in the rewriting rule #1 through #3 that when certain requirements on the provenance related predicate are met, we can remove an original predicate.

For the third rewriting operation, an arithmetic predicate should be pushed towards the beginning of a query evaluation process, since the earlier the filtering takes place, the smaller the intermediate results are. This has been studied extensively and is not our focus.

The above rewriting operations are all *atomic* rewriting operations. A proper composition of these rewriting operations will produce a simpler and optimized tracing queries, as illustrated in Figure 3.3b. In the following section, we will show in the rewriting rules #4 through #8, that when certain requirements on the provenance related predicates are met, a combination of predicate substitution and predicate removal can produce a simpler tracing query than the original one.

3.3.3 Rewriting Rules

Given a query $r: R(Y) :- S_1(X_1), ..., S_m(X_m)$ or $r: R(Y, \Sigma(\langle A \rangle)) :- S_1(X_1), ..., S_m(X_m)$, and a set of answer tuples $y_1, ..., y_n$, supposing the tracing query $r_p: S_i^p(X_i) :- S_1(X_1), ...,$ $S_m(X_m), T(Y)$, where $T(Y)$ is the provenance related predicate and $S_i(X_i)$ are the predicates in the original query, we have the following rewriting rules for the tracing query r_p.

Rewriting Rule #1: if $Y \supseteq X_i$, then r_p can be transformed into

$$r'_p: S_i^p(X_i) :- T(Y) \tag{3.13}$$

Intuitively, when all the attributes of S_i are in the schema of the answer tuples (i.e., $Y \supseteq S_i$), the provenance of the answer tuples in S_i is completely included in the answer tuple, and thus there is no need to query the other predicates than the answer tuples (i.e., $T(Y)$).

Formally, we show that the result relations S_i^p of r_p and r'_p are the same, i.e., $\{y_1.X_i, ..., y_n.X_i\}$.

Given the way a tracing query is constructed, $T(Y) = \{y_1, ..., y_n\}$. Since $Y \supseteq X_i$, then the evaluation of r'_p is simply to substitute the proper value in y_k $(k = 1, ..., n)$ into Y and also X_i. Therefore, we will have the result relation of r'_p as $\{y_1.X_i, ..., y_n.X_i\}$.

For every y_k $(k = 1, ..., n)$, we can construct a substitution θ_k such that $\theta_k(Y) = y_k$ and $S_j(X_j)[X_j/\theta_k(X_j)]$ $(j = 1, ..., m)$ evaluates to true. Such θ_k must exist, otherwise, y_k can not be an answer tuple to the original query. Therefore, θ_k $(k = 1, ..., n)$ will produce a fact $y_k.X_i$ in the result relation of r_p. Moreover, any substitution θ that makes the body of r_p evaluate true also makes $T(Y)$ evaluate true. This further means that $\theta(Y) = y_k$ where $k \in \{1, ..., n\}$. Since $Y \supseteq X_i$, $\theta(X_i) = y_k.X_i$. Therefore there will no result tuple other than $y_1.X_i, ..., y_n.X_i$ in the result relation of r_p.

Rewriting Rule #2: if $Y \supseteq PK_{S_i}$, where PK_{S_i} is the primary key of S_i, then r_p can be transformed into

$$r'_p : S_i^p(X_i) :- S_i(X_i), T(Y) \tag{3.14}$$

The intuition behind this rule is similar to that behind the rewriting rule #1. The primary key of S_i can uniquely determine a tuple in S_i. When the primary key of S_i is included in the schema of the answer tuples, i.e., $Y \supseteq PK_{S_i}$, the answer tuples are sufficient to identify their contributing tuples in S_i.

The proof of the rewriting rule #2 is similar to that of the rewriting rule #1. In short, we can construct substitutions based on $y_1, ..., y_n$ and show the result relation of $r_p(r'_p)$ using these substitutions is $\{y_1.X_i, ..., y_n.X_i\}$.

Rewriting Rule #3: if $Y \supseteq PK_{S_j} \wedge (\forall k \in \{1, ..., m\}\ Y \supseteq X_j \cap X_k) \wedge ((\cup_{l=1}^{l=j-1} X_l) \cup (\cup_{l=j+1}^{l=m} X_l) \cup Y \supseteq X_i)$, then

$$S_i^p(X_i) :- S_1(X_1), ..., S_{j-1}(X_{j-1}), S_{j+1}(X_{j+1}), ..., S_m(X_m), T(Y) \tag{3.15}$$

The goal of this rule is to take advantage of the presence of $T(Y)$ to remove $S_j(X_j)$. This can be done if (1) $T(Y)$ is more restrictive than $S_j(X_j)$ and (2) every variable in $X_i \cap X_j$ must also be in some predicates other than X_j in the body, otherwise the rule will become unsafe after the removal of $S_j(X_j)$.

We now show why $T(Y)$ is more restrictive than $S_j(X_j)$ given the requirements in the rewriting rule #3. To show that it is sufficient to show that every substitution that makes $T(Y)$ evaluate to true makes $S_j(X_j)$ evaluate true.

$T(Y)$ is the provenance related predicate. According to the way the tracing query is constructed, $T(Y) = \{y_1, ..., y_n\}$. Thus any substitution θ for r_p that makes $T(Y)$ evaluate true satisfy the condition $\exists k \in \{1, ..., n\}\ \theta(Y) = y_k$. Since $Y \supseteq PK_{S_j}$, therefore, $\theta(X) =$

$y_k.X_j$. Moreover, $S_j(X_j)[X_j/y_k.X_j]$ must evaluate to true as well, otherwise y_k will not be an answer tuple in the first place.

In the bottom right graph in Figure 3.3b, the predicate pid = iPhone can be considered as a temporary table $T(pid) = \{\{\text{iPhone}\}\}$. Then, $Products^p(pid)$ is removed since the conditions in the above rule is satisfied.

The rewriting rules #1 through #3 involve only predicate removal operations. In the following rules, we combine the predicate substitution and predicate removal.

In Figure 3.3b, the first rewriting is in fact two predicate substitutions in a row: substituting $Orders^p$ with the body of r_5, i.e., $Orders^p\_Productes^p\_Categories^p$ and then substituting $Orders^p\_Productes^p\_Categories^p$ with the body of r_4. After these two predicate substitutions, two predicate removals follow. Thus, the final outcome is a query much simpler than the original query.

The reason that we combine the predicate substitution and the predicate removal is to avoid intermediate table access or intermediate provenance retrieval without increasing the number of predicates in the involved Datalog rules.

Suppose we have two rules $r: \ S_i^p(X_i) :- S_1(X_1), ..., S_m(X_m), T(Y)$, and $r': \ R_k(Z_k) :- R_1(Z_1), ..., R_n(Z_n), S_i^p(X_i)$, where $T(Y)$ and $S_i^p(X_i)$ are the provenance related predicates, and $S_i^p(X_i)$ is also intermediate provenance.

Imagine that we substitute S_i^p in r' for the body of r. Then, although we eliminate the reference of intermediate provenance S_i^p in r', the resulting r' has now $n+m+1$ predicates instead of $n + 1$. The elimination of intermediate provenance may not be worth the price of more predicates in r', and we do not really want to apply the predicate substitution. However, after the predicate substitution we may find predicate removals and reduce the number of predicates in r'; this is the situation where we really want to apply the predicate substitution rewritings.

A general procedure of applying predicate substitution is:

(i). trying to reduce the number of predicates in r using rewriting rules #1 through #3;

(ii). substituting $S_i^p(X_i)$ in r' for the body of r;

(iii). trying to reduce the number of predicates in the rewritten r' using rewriting rules #1 through #3;

(iv). checking if the number of predicates in the rewritten r' is greater than $n + 1$, if it is, abort the predicate substitution; else keep it.

There are 4 choices for the first predicate removal operation (the rewriting rule #1 through #3, and the no-op), and the same 4 choices for the second predicate removal operation. Thus there are 16 choices in all. Not all combinations make sense. For example, no-op, sub, no-op will lead to more predicates in the body on account of the substitution. Similarly #3,sub,#3 also leads to more predicates in the rewritten rule than in the original. After eliminating 6 such combinations, we are left with 10 possible combinations that result in a rewritten r' with less predicates (and no S_i^p). These are listed in Figure 3.4.

We now describe a few of the noteworthy rules in Figure 3.4. The rest are similar.

This rewriting rule #4 is a result of applying rewriting rule #1 to r and then a predicate substitution in r'. Since $Y \supseteq X_i$, then according to the rewriting rule #1, $S_i^p(X_i) :- T(Y)$. Furthermore, substituting $S_i^p(X_i)$ for $T(Y)$ into r', we get the equation in the rewriting rule #4.

This rule can avoid the computation of intermediate provenance S_i^p and also avoid the access to the intermediate table S_i. Meanwhile, although the transformed r' has the same number of predicates as the original r', the number of tuples in $T(Y)$ is less than or equal to the number of tuples in S_i^p. Therefore the transformed r' can take less time to compute than the original r'.

As for the rewriting rule #5 in Figure 3.4, we substitute $S_i^p(X_i)$ in r' for the body of r, we get $R_k(Z_k) :- R_1(Z_1), ..., R_n(Z_n), S_1(X_1), ..., S_m(X_m), T(Y)$. Since $Y \supseteq Z_k$, then according to the rewriting rule #1, we get the equation in the rewriting rule #5.

As for the rewriting rule #12 in Figure 3.4, we first apply the rewriting rule #3 to r, and then substituting S_i^p in r' for the body of the transformed r, and finally applying the rewriting rule #2 to r'.

3.4 Applying Rewriting Rules To A Datalog Program

The rewriting rules #1 through #13 can be applied repeatedly. Moreover, the rewriting due to one rule may trigger more opportunities to apply the same or other rules. Therefore, we provide an algorithm to apply the rewriting rules to each of the Datalog rules in a given rule set, and the order in which each Datalog rule is examined is derived from the partial order of the rule dependency graph of these Datalog rules.

When a Datalog rule set (a.k.a. a Datalog program) is given, a rule dependency graph, or dependency graph in short, can be constructed for it. Each node in the rule dependency graph corresponds to a Datalog rule in this set, and the edge from a node to another means the former depends on the latter. In this paper, we assume no recursive queries, and thus the resulting tracing queries from the original query are non-recursive as well. Therefore, the dependency graph is a DAG (directed acyclic graph).

Notice that the rule dependency graph should not be confused with the predicate dependency graph. In rule dependency graphs, nodes are rules and edges are rule dependency; while in predicate dependency graphs, nodes are predicates and edges are inference.

To build the rule dependency graph of a given Datalog rule set, we have Algorithm 1.

Algorithm 1 runs two passes over the rule set. The time complexity of the first pass is linear in the number of nodes, or the number of rules. The time complexity of the second

Algorithm 1: **C**onstruction Of **D**ependency **G**raph Of A Rule Set (CDG)

```
Input: $\mathbb{R}$, which is a Datalog rule set
begin
    Initialize a DAG $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of directed edges
    Initialize a hash map $head\_map$, which maps a predicate name to a rule identifier
    for every rule $r$ in $\mathbb{R}$ do
        $V \leftarrow V \cup \{r\}$
        $T \leftarrow$ the head of $r$
        Insert $(T, r)$ into the hash map $head\_map$
    end
    for every rule $r$ in $\mathbb{R}$ do
        for every predicate $S$ in the body of $r$ do
            if $(r' = head\_map(S)) \neq null$ then
                $E \leftarrow E \cup \{(r, r')\}$
            end
        end
    end
    return $G$
end
```

pass is of the order of the magnitude of the square of the number of predicates.

With the rule dependency graph built, we can optimize the rule set by applying the rewriting rules to each node one by one by in a traversal over the graph. A node is processed only when all the nodes it depends on have been processed. This means we will process the nodes according to a topological sort/ordering of the rule dependency graph.

During the first pass over the entire graph, we process each node with the predicate removal operations if they are applicable. During the second pass over the entire graph, we process each node with the combination of predicate substitution and predicate removal operations. When a predicate substitution is performed, the rule dependency graph needs to be updated accordingly, as shown in the lines 20-25 in Algorithm 2.

We then keep repeating the second pass until there is no rewriting during one pass. The reason we repeat the second pass is because the substitution of a predicate introduces new predicates into the Datalog rule (i.e., new edges into the rule dependency graph), and these new predicates may be substituted further.

When a node is processed, if there is more than one applicable rewriting rule, we choose the rule that will lead to a rewritten rule with the least number of predicates in the body.

Algorithm 2: **T**raversal Of **D**ependency Graph (TDG)

```
   Input: $\mathbb{R}, G = (V, E)$
 1 begin
 2     Initialize a queue node_to_be_processed to a queue of nodes in $V$ in a topological ordering
 3     Initialize a queue node_processed to an empty queue
 4     while node_to_be_processed is not empty do
 5         $r \leftarrow$ node_to_be_processed
 6         Remove $r$ from node_to_be_processed
 7         Add $r$ to node_processed
 8         repeat
 9             Apply an applicable rewriting rule in #1 through #3 to $r$
10         until No rule in the rewriting rules #1 through #3 is applicable
11     end
12     node_to_be_processed $\leftarrow$ node_processed
13     repeat
14         while node_to_be_processed is not empty do
15             $r' \leftarrow$ node_to_be_processed
16             Remove $r'$ from node_to_be_processed
17             Add $r'$ to node_processed
18             for every $(r', r) \in E$ do
19                 Apply an applicable rewriting rules in #4 through #13 to $(r', r)$
20                 if a predicate substitution is preformed then
21                     Remove $(r', r)$ from $E$
22                     for every new predicate $r''$ that is introduced into $r'$ by the predicate substitution
                       do
23                         Add an edge $(r', r'')$ to $E$
24                     end
25                 end
26             end
27         end
28         node_to_be_processed $\leftarrow$ node_processed
29     until The last pass does not make any rewriting
30     return $\mathbb{R}$
31 end
```

Theorem 3.16. *Algorithm 2 produces a set of rewritten Datalog rules such that none of the rewriting rules (#1 to #13) can be applied further to decrease the number of predicates in any of these Datalog rules.*

From G, multiple topological orderings can be derived. For our Algorithm 2, we can use any of these topological orderings, and the result is a set of rewritten rules satisfying Theorem 3.16.

3.4.1 Optional Recording Of The Values Of Special Attributes

The rewriting of Datalog rules discussed above makes use of provenance related predicates on special attributes, i.e., primary keys in the source tables. When the primary keys are in the output result, the provenance related predicates are most restrictive and thus most useful in the rewriting. However, primary keys are not always in the output. In fact, we can rewrite the original query to include the primary keys in the output. In [15], rewriting techniques have been introduced to rewrite the original query to include all the attributes from the source tables in the output. It can be easily adapted to include only the primary keys instead of all the attributes. Thus, the extra space cost due to additional attributes in the output will be greatly reduced. By paying for the extra space cost of including the primary keys in the output, we can maximize the applicability of the rewriting rules introduced in this chapter.

Moreover, if the user requests to exclude the usage of an arbitrarily selected source table from the retrieval of provenance. A similar approach can be used. That is, we rewrite the original query using the techniques introduced in [15] to include all the attributes from the selected source table in the output. Thus, for each of the attributes in the selected source table, we can add a provenance related predicate to the provenance retrieval query and this predicate checks the equality of the attribute and its output values. These predicates will then remove the reference(s) to the selected source table in the provenance retrieval query according to our rewriting rules, since they are most restrictive regarding the attributes in the selected source table. Therefore, we can retrieve provenance without referring to the source table arbitrarily selected by the user.

As a summary, by taking the advantage of the presence of provenance related predicates in the rules, our optimization rules have three direct or indirect benefits:

(i). fewer predicates in the rule body (in other words, fewer joins in the query evaluation), (ii). the elimination of unnecessary access to intermediate tables during provenance retrieval,

(iii). the elimination of unnecessary retrieval of intermediate provenance, which is the provenance inside the intermediate tables, e.g., the retrieval of $Products^p$.

3.5 Experimental Evaluation

We conducted experiments to evaluate the benefits obtained by applying the rewriting rules developed in this chapter. In particular, given a customized provenance request, we compare the time cost of computing it using the optimized tracing queries with the time cost of computing it using the baseline tracing queries.

3.5.1 Database And Queries Used

We use the database and queries specified in TPC-H benchmark. There are 8 source tables, *lineitem*, *orders*, *customer*, *part*, *supplier*, *nation*, *region*, *partsupp*. Our instances of the 8 tables are generated through DBGen with the scaling factor of 1.

There are 22 query templates in TPC-H. Our method is applicable to multiple-block ASPJ queries with set-oriented subqueries. Therefore we exclude queries from these 22 query templates that can not be transformed into this desired form. This leaves us with 6 single-block (A)SPJ query templates and 2 two-block nested (A)SPJ query template. The single-block query templates are No.1, No.3, No.5, No.6, No.10 and No.19, as numbered in TPC-H benchmark. The two-block query templates are No.7 and No.9. One instance query was produced for each query template using QGen.

3.5.2 Platform And Measurement Description

We ran all of our experiments on a machine with the following configuration: (i). Intel Core i7 CPU with 7.7GB RAM, (ii). Ubuntu 11.10 64bit, (iii). PostgreSQL 8.4.4 64bit, with “share_buffer” set to be 16MB and “effective_cache_size” 128MB.

All measurements were taken as follows:

(i). every time cost shown in the figures is an average of three measurements;

(ii). the time measurement (in seconds) is measured by using “pg_statement” module in PostgreSQL;

(iii). the system cache and PostgreSQL cache are cleaned before the evaluation of each query by “echo 3 > /proc/sys/vm/drop_caches”.

3.5.3 Choice of Customization

Our experiments include the following steps.

(i). For each query Q, we choose an answer tuple t from the answer set. (We conveniently choose the first answer in the answer set.)

(ii). We construct the tracing query Q^{trace} to retrieve the provenance of t. Q^{trace} is determined by t and Q.

(iii). We set a customized provenance requests Q^{cust}.

(iv). We compute the customized provenance using

  (a) baseline tracing queries, i.e., $Q^{cust}(Q^{trace}(D))$

  (b) optimized tracing queries, i.e., $Q^{opt}(D)$, where Q^{opt} is the result of optimizing $Q^{cust} \circ Q^{trace}$.

Unfortunately, we do not have any gold standard for customized provenance requests. The TPC-H benchmark, of course, does not provide any. Furthermore, since this is a benchmark, rather than a real database, we cannot even use subjective expert knowledge. In short, there is no way for us to use real, or even “realistic”, customization.

In light of this, we decided to be minimalist. Rather than create complex customizations, we decided to restrict our choice of Q_{cust} to a simple conjunction of equality conditions on one or more attributes in one of the source tables followed by a projection. Furthermore, we decided to write these simple customization functions such that precisely one rewriting rule is fired by each $Q_i^{cust} \circ Q^{trace}$.

We choose to test 4 representative rewriting rules rather than all of them. By observing the rewriting rules #1 through #13, we can see 4 patterns of the rewritten rules. After rewriting, the number of predicates in a rule body either remains the same (#4), or becomes 1 (#1 and #5 through #8), or becomes 2 (#2 and #9 through #12), or decreases by 1(#3 and #13).

Recall that we have both single-block queries and double-block nested queries in our query set. The scope for optimization is somewhat different in the two. For these reasons, we decided to test one rewriting rule for single-block queries and another rule from the same pattern family for nested queries. We choose to experiment with the last two pattern families.

Given a single-block query Q and an answer tuple t, we construct Q_1^{cust} to make the rewriting rule #2 applicable to $Q_1^{cust} \circ Q^{trace}$, and construct Q_2^{cust} to make the rewriting rule #3 applicable to $Q_2^{cust} \circ Q^{trace}$.

Correspondingly, given a two-block query Q and an answer tuple t, we construct Q_1^{cust} to make the rewriting rule #9 applicable to $Q_1^{cust} \circ Q^{trace}$, and construct Q_2^{cust} to make the combination of rewriting rule #13 applicable to $Q_2^{cust} \circ Q^{trace}$.

3.5.4 Experiments On Time Costs

In Figure 3.5, we show the time costs of tracing customized provenance by using the baseline tracing queries and the optimized tracing queries, which are optimized by the rewriting rule #2 for the single-block queries and by the rewriting rule #9 for the two-block queries.

We experiment on 8 queries and thus have 8 pairs of time costs. The left 6 queries are single-blocked, and the time costs are measured in seconds. The optimized tracing queries always run faster. The right 2 queries are two-blocked. The optimized tracing queries always run faster too. Moreover, since the time difference between the baseline approach and the optimized approach is as large as not being at the same magnitude, we show the time costs in natural logarithms. The time difference is large because the optimized approach does not access the intermediate tables while the baseline approach does and has to recompute them.

In Figure 3.6, we show the time costs of retrieving the customized provenance using baseline tracing queries and optimized tracing queries, which are optimized by the rewriting rule #3 for single-block queries or by the rewriting rule #13 for the two-block queries.

Queries No.1 and No.6 have only one source table, therefore rule #3 is not applicable. Thus, Figure 3.6 shows only queries No.3, No.5, No.10, No.19, No.7 and No.9. The trend shown in Figure 3.6 is similar to that in Figure 3.5. The optimized queries always run faster. The elimination of access to intermediate tables, such as in query No.7 and No.9, has the most significant improvement in terms of computational time.

3.6 Related Work

Provenance in the database setting has been addressed through different points of view. One approach is in [12] for (A)SPJ queries. It is later extended by [16] for queries with provnestsubq. In both definitions, provenance includes the contributing source tuples from different derivations of the result tuple. Another approach to provenance is in [18] for SPJ queries. In [18], the provenance is represented as a polynomial in the semi-ring algebra, and each term in the polynomial is a derivation. Moreover, in [5], the why provenance includes all the alternative derivations; and the where provenance has a finer granularity, which captures the mapping from an attribute in the source tuples to an attribute in the result tuple, if applied to the relational database setting. Finally, in [10], data provenance is viewed as data dependency. The result tuple in general depends on the source tuples used to generate it.

Provenance can be computed in either a lazy way or an eager way [35]. In a lazy approach, provenance is computed by tracing queries by request [12]. In an eager approach, provenance is generated with the result either through the query rewriting [15] or through annotation propagation [18, 14] or through recording data operation in a curated database [2] or through input from users. The lazy approach may implicitly require the availability of the database at a historical time point [40]. The eager approach may require an efficient storage mechanism [8].

An optimization technique used typically in database systems can be based on query rewriting rules or based on evaluation cost estimation. For a rewriting rule based optimization method, the goal is to convert a query plan corresponding to a query into an equivalent query plan using the rewriting rules, such that the new plan can be executed more efficiently. In our work, we come up with rewriting rules that are applicable for tracing queries in the context of provenance retrieval; these rules are then used for rewriting the tracing queries into more efficient queries.

Our work explores the opportunities of optimizing the tracing queries. Those opportunities are due to the provenance related predicates present in the tracing queries. The provenance related predicates are often more restrictive than the relational predicates also in the tracing queries, as has been argued in this chapter. Being more restrictive, those provenance related predicates are able to eliminate the less restrictive predicates from the tracing queries and make the tracing queries faster to evaluate.

3.7 Conclusion

Users are often not interested in the complete provenance of given answer tuples, but rather in a specific question that may only need part of the complete provenance. Baseline approaches first compute the complete provenance and then select the requested part. It is obvious that unnecessary computation on the unrequested part is involved. In this chapter, we suggested to customize the provenance using SQL language or other equivalent languages and proposed an approach to computing the customized provenance efficiently by applying rule-based optimization techniques to the tracing queries that retrieve the customized provenance. These optimization techniques take advantage of the provenance related predicates in the tracing queries and simplify the tracing queries by removing redundant original predicates. We show through experiments that the optimizations can reduce the time cost of provenance retrieval significantly.

Orders

oid[a]	pid[b]	amt[c]	prc[d]
1	Nexus	5	20
1	iPhone	3	10
2	Galaxy	4	15

Ratings

pid	cust[e]	rt[f]
Nexus	Cu1	3
iPhone	Cu2	4
Galaxy	Cu1	4
Galaxy	Cu2	3.5

Categories

pid	caid[g]
Nexus	SmartPhone
iPhone	SmartPhone
Galaxy	SmartPhone

[a] *oid* is the order ID.
[b] *pid* is the product ID.
[c] *amt* is the amount.
[d] *prc* is the price.
[e] *cust* is the customer ID.
[f] *rt* is the rating of the product.
[g] *caid* is the category ID.

(a) Database

In SQL:
SELECT $C_1.caid$, $sum(O_1.prc * O_1.amt)$ AS *rev*
INTO *Revenue*
FROM *Orders* AS O_1
INNER JOIN *Categories* AS C_1
INNER JOIN ((SELECT $R_1.pid$ FROM *Ratings* AS R_1 GROUP BY $R_1.pid$ HAVING AVG($R_1.rt$)>3)
UNION
(SELECT "iPhone" AS *pid*)) AS *Products*
GROUP BY $C_1.caid$

In Datalog:
r_1: $AvgRatings(pid, avg(\langle rt \rangle)\ as\ avgrt)$:– $Ratings(pid, cust, rt)$
r_2: $Products(pid)$:– $AvgRatings(pid, avgrt), avgrt > 3$
r_2': $Products$("iPhone") :–
r_3: $Revenue(caid, sum(\langle amt * prc \rangle)\ as\ rev)$:– $Products(pid), Orders(oid, pid, amt, prc), Categories(caid, pid)$

(b) Query

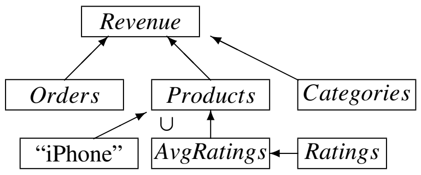
Revenue

caid	rev
SmartPhone	90

Products

pid
iPhone
Galaxy

(c) Query Result



(d) Predicate Dependency Graph

Figure 3.1: Example Database, Query And Its Result

Retrieval Step 1:

r_4: $Orders^p\_Products^p\_Categories^p(Orders\_oid, Orders\_pid, Orders\_amt, Orders\_prc,$
$Products\_pid, Categories\_caid, Categories\_pid),$
$:\!-\ Products(pid), Orders(oid, pid, amt, prc), Categories(caid, pid), caid = \text{SmartPhone}$

r_5: $Orders^p(oid, pid, amt, prc)$
$:\!-\ Orders^p\_Products^p\_Categories^p(Orders\_oid, Orders\_pid, rders\_amt, Orders\_prc,$
$Products\_pid, Categories\_caid, Categories\_pid),$
$Orders\_oid = oid, Orders\_pid = pid, Orders\_amt = amt, Orders\_prc = prc$

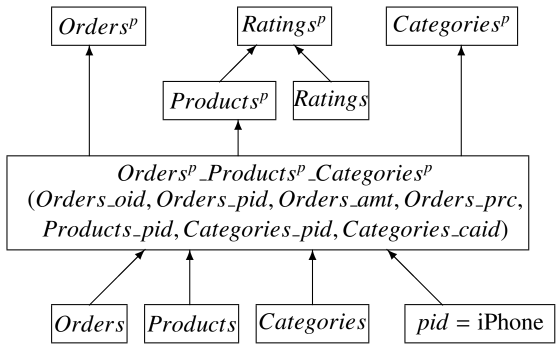
r_6: $Products^p(pid)$
$:\!-\ Orders^p\_Products^p\_Categories^p(Orders\_oid, Orders\_pid, Orders\_amt, Orders\_prc,$
$Products\_pid, Categories\_caid, Categories\_pid),$
$Products\_pid = pid$

r_7: $Categories^p(caid, pid)$
$:\!-\ Orders^p\_Products^p\_Categories^p(Orders\_oid, Orders\_pid, Orders\_amt, Orders\_prc,$
$Products\_pid, Categories\_caid, Categories\_pid),$
$Categories\_caid = caid, Categories\_pid = pid$

Retrieval Step 2:

r_8: $Ratings^p(pid, cust, rt)$
$:\!-\ Ratings(pid, cust, rt), Products^p(pid)$

(a) Retrieval Of Complete Provenance of {SmartPhone, 90}


$Orders^p$
$Ratings^p$
$Categories^p$
$Products^p$
$Ratings$
$Orders^p\_Products^p\_Categories^p$
$(Orders\_oid, Orders\_pid, Orders\_amt, Orders\_prc,$
$Products\_pid, Categories\_pid, Categories\_caid)$
$Orders$
$Products$
$Categories$
pid = iPhone

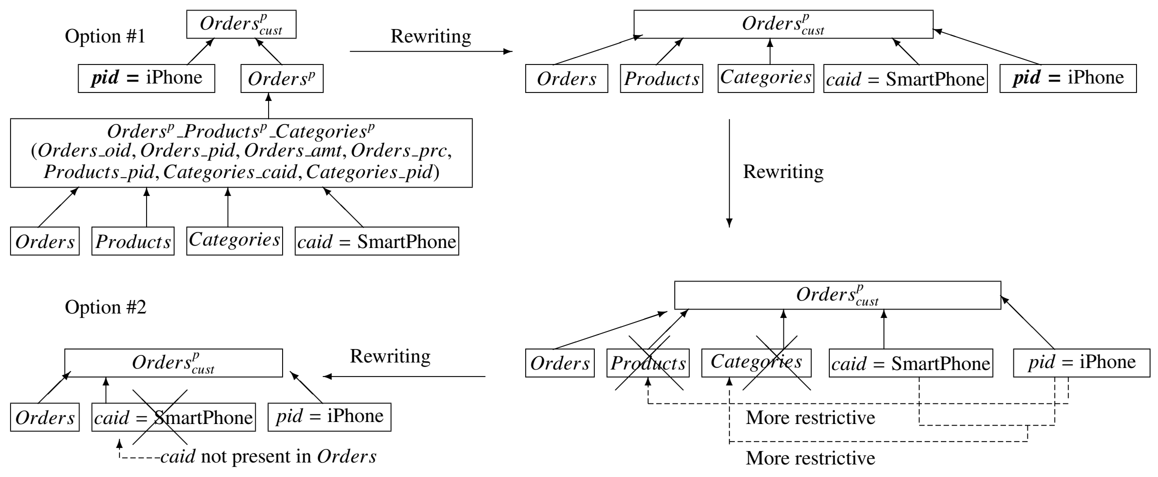
(b) Illustration Of Rules In Figure 3.2a

Figure 3.2: Retrieval Of Complete Provenance Of {SmartPhone,90}

Option #1: baseline approach
r_4 & r_5 in Figure 3.2
r_9: $Orders^p_{cust} : -Orders^p, pid = \text{iPhone}$

Option #2: optimized approach
r'_5: $Orders^p_{cust}(oid, pid, amt, prc) :- Orders(oid, pid, amt, prc), pid = \text{iPhone}$

(a) Retrieval Of Customized Provenance Related To “iPhone”



(b) Optimization Applied To Rules In Baseline Approach Shown In Figure 3.3

Figure 3.3: Retrieval Of Customized Provenance Of Orders Related To “iPhone” Inside The Provenance Of {SmartPhone,90}

Rule	Requirements	Rewritten r'	Operation Sequence		
#4	$Y \supseteq X_i$	$R_k(Z_k) :- R_1(Z_1), ..., R_n(Z_n), T(Y)$	#1	sub[a]	no-op[b]
#5	$Y \supseteq Z_k$	$R_k(Z_k) :- T(Y)$	no-op	sub	#1
#6	$Y \supseteq Z_k \wedge Y \supseteq X_i$		#1	sub	#1
#7	$Y \supseteq Z_k \wedge Y \supseteq PK_{S_i}$		#2	sub	#1
#8	$Y \supseteq Z_k \wedge Y \supseteq PK_{S_j} \wedge (\forall l \in \{1, ..., m\}\ Y \supseteq X_j \cap X_l)$ $\wedge((\cup_{l=1}^{l=j-1} X_l) \cup (\cup_{l=j+1}^{l=m} X_l) \cup Y \supseteq X_i)$		#3	sub	#1
#9	$Y \supseteq PK_{R_k}$	$R_k(Z_k) :- R_k(Z_k), T(Y)$	no-op	sub	#2
#10	$Y \supseteq PK_{R_k} \wedge Y \supseteq X_i$		#1	sub	#2
#11	$Y \supseteq PK_{R_k} \wedge Y \supseteq PK_{S_i}$		#2	sub	#2
#12	$Y \supseteq PK_{R_k} \wedge Y \supseteq PK_{S_j} \wedge (\forall k \in \{1, ..., m\}\ Y \supseteq X_j \cap X_k)$ $\wedge((\cup_{l=1}^{l=j-1} X_l) \cup (\cup_{l=j+1}^{l=m} X_l) \cup Y \supseteq X_i)$		#3	sub	#2
#13	$Y \supseteq X_i \wedge Y \supseteq PK_{R_j} \wedge (\forall l \in \{1, ..., m\}\ Y \supseteq Z_j \cap Z_l)$ $\wedge((\cup_{l=1}^{l=j-1} Z_l) \cup (\cup_{l=j+1}^{l=m} Z_l) \cup Y \supseteq Z_i)$	$R_k(Z_k) :- R_1(Z_1), ..., R_{j-1}(Z_{j-1}),$ $R_{j+1}(Z_{j+1}), ..., R_k(Z_k), T(Y)$	#1	sub	#3

[a] sub means predicate substitution in r'
[b] no-op means no operation is applied to either r or r'

Figure 3.4: Compositions Of Predicate Substitution And Predicate Removal

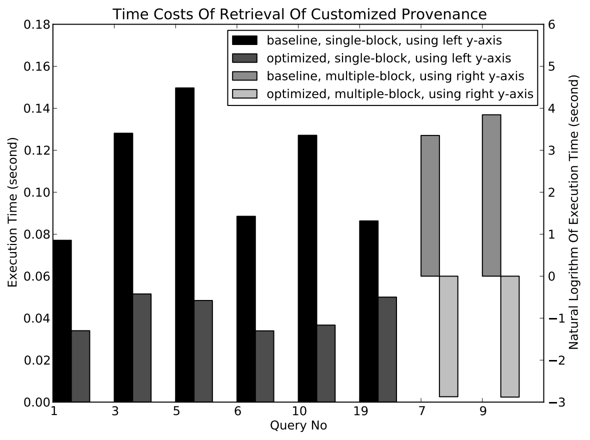


Figure 3.5: Baseline Tracing Queries vs. Optimized Tracing Queries Using Rewriting Rule #2 or #9

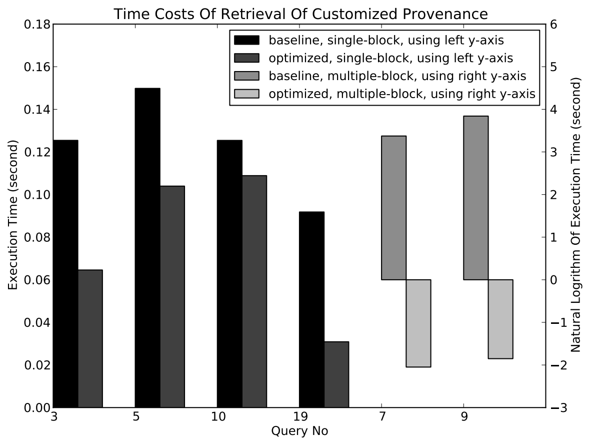


Figure 3.6: Baseline Tracing Queries vs. Optimized Tracing Queries Using Rewriting Rule #3 or #13

CHAPTER IV

Validation Of Derived Data

4.1 Introduction

In a modern scientific project, there frequently is a huge body of raw data collected from experiments. Usually, this data is stored in a database, and processed by SQL queries to make it ready for further analysis. This derived data is vital for the final scientific conclusions the scientists draw from the experiments. When the raw data changes, e.g., due to a re-collection or a curation of the raw data, in the form of database inserts, deletes and updates, it is important to know whether previously derived data and results are still valid or derivable.

Previously derived data can be validated by incrementally maintaining the derived data set with regard to the updated database. However, scientists are often interested in only some particular portion of the derived data set, possibly even a single tuple. For example, this may be a specific result quoted in some publication or used in follow-on work. In such cases, one desires a more efficient way to validate the part in question without refreshing the whole derived data set, especially when the derived data set is large.

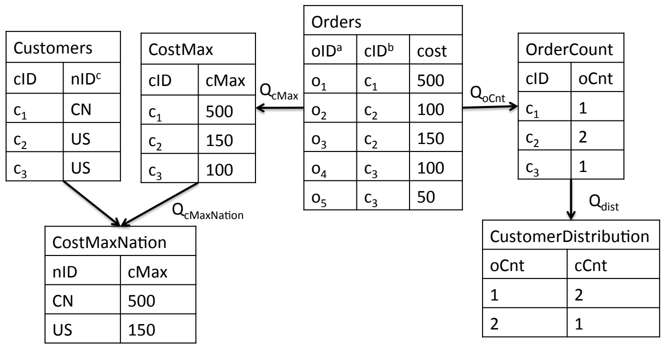
In this chapter, we propose an approach to validating the selected answer tuples derived from a nested query in case of modifications to the source database, and provide an explanation of the invalidation of any of these tuples that is invalidated. For the former part, we base our approach on the incremental evaluation of materialized views enhanced with pruning predicates derived from the selected tuples and tailored for both positive and negative tuples[1] in delta tables; for the latter part, we treat the invalidated tuples as negative tuples in the delta result table and retrieve their provenance as a set of both positive and negative tuples within original and/or delta source tables.

Consider the following illustrative scenario, which we have designed using customers and orders, as is so common in the database literature. We use this as our running example, to make it accessible to the reader without requiring domain knowledge in any scientific discipline.

Example 4.1. Assume we have two simple tables *Orders* and *Customers* as shown in Figure 4.1. Every order in *Orders* consists of a unique order ID, a customer ID and the cost of the order. Every customer in *Customers* consists of a unique customer ID and a nation ID. There are four simple ASPJ queries Q_{cMax}, Q_{oCnt}, Q_{dist} and $Q_{cMaxNation}$ as shown in Figure 4.2. Q_{cMax} computes the the maximum cost of a single order for each customer; Q_{oCnt} computes the order count for each customer; $Q_{oCnt} \circ Q_{dist}$ computes the distribution of customers for each count of orders; $Q_{cMax} \circ Q_{cMaxNation}$ computes the maximum cost of a single order for each nation. The derived tables are *CostMax*, *OrderCount*, *CustomerDistribution* and *CostMaxNation* are also shown in Figure 4.1.

Suppose we have updates to *Orders* table as $\Delta Orders$, shown in Figure 4.3. The *CNT* attribute in $\Delta Orders$ is the number of derivations of each tuple. The precise meaning of *CNT* will be specified in Section 4.3 as preliminaries, and it is sufficient for now to consider only the sign of *CNT*. Tuples with positive *CNT* are to-be-inserted tuples and tuples with negative *CNT* are to-be-removed tuples. $\Delta Orders$ leads to the update $\Delta CostMax$ to the result table *CostMax*. For example, $(o_6, c_3, 150) \in \Delta Orders$ is inserted into the

[1]Tuples in delta tables can have positive counts or negative counts [20]. We call tuples with positive counts positive tuples, and tuples with negative counts negative tuples. See Section 4.3.2 for a brief review of delta tables and incremental view maintenance.



Customers

cID	nID[c]
c_1	CN
c_2	US
c_3	US

CostMax

cID	cMax
c_1	500
c_2	150
c_3	100

Orders

oID[a]	cID[b]	cost
o_1	c_1	500
o_2	c_2	100
o_3	c_2	150
o_4	c_3	100
o_5	c_3	50

OrderCount

cID	oCnt
c_1	1
c_2	2
c_3	1

CostMaxNation

nID	cMax
CN	500
US	150

CustomerDistribution

oCnt	cCnt
1	2
2	1

[a]: *oID* is a unique ID for each order.
[b]: *cID* is a unique ID for each customer.
[c]: *nID* is a unique ID for each nation.

Figure 4.1: Source Table And Derived Tables

source table $Orders$, and then $(c_3, 100) \in CostMax$ is replaced with $(c_3, 150)$. We say that $(o_6, c_3, 150)$ contradicts the previous answer $(c_3, 100)$, and $(o_6, c_3, 150)$ serves as an explanation of the invalidation of $(c_3, 150)$ from $CostMax$.

Note that upon the insertion of $(o_6, c_3, 150)$, the derivation that produced $(c_3, 100)$ is still in $Orders$. However, $(c_3, 100)$ is no longer an answer in $CostMax$. Thus, the existence of contributory derivations is not sufficient to form an answer. Moreover, it is obvious that there is more than one way to contradict an answer. For example, $(o_7, c_3, 200)$ can contradict $(c_3, 100)$ as well. On the other hand, the removal of contributory source tuples, e.g., $(o_4, c_3, 100)$ can invalidate $(c_3, 100)$ too.

In general, an answer's validity can be changed by the insertion of contradictory source tuples or by the removal of contributory source tuples.

The contributions of this chapter are as follow.

(i). Given a set of answer tuples to a simple or nested (A)SPJ query over a source database, we validate it by incrementally evaluating the query with pruning attributes tailored for both positive tuples (i.e., existing or inserted tuples) and negative tuples (i.e., removed tuples). These pruning predicates can reduce computational cost by avoiding evaluation over irrelevant source tuples. These pruning predicates can be

Q_{oCnt}:
SELECT cID, $count(oID)$ as $oCnt$
FROM $Orders$ WHERE $cost >= 100$
GROUP BY cID
r_{oCnt}:
$OrderCount(cID, count(\langle oID \rangle)\ AS\ \ oCnt) :- Orders(oID, cID, cost)$
$? - OrderCount(cID, oCnt)$
Q_{dist}:
SELECT $oCnt$, $count(cID)$ as $cCnt$
FROM $OrderCount$ GROUP BY $oCnt$
r_{dist}:
$CustomerDistribution(oCnt, count(\langle cID \rangle)\ AS\ \ cCnt) :- OrderCount(cID, oCnt)$
$? - CustomerDistribution(oCnt, cCnt)$
Q_{cMax}:
SELECT cID, $max(cost)$ as $cMax$
FROM $Orders$
GROUP BY cID
r_{cMax}:
$CostMax(cID, max(\langle cost \rangle)\ AS\ \ cMax) :- Orders(oID, cID, cost)$
$? - CostMax(cID, cMax)$
$Q_{cMaxNation}$:
SELECT nID, $max(cMax)$ as $cMaxNation$
FROM $CostMax$, $Customers$
WHERE $CostMax.cID = Customers.cID$
GROUP BY nID
$r_{cMaxNation}$:
$CostMaxNation(nID, max(\langle cMax \rangle)\ AS\ \ cMaxNation) :- CostMax(cID, cMax), Customers(cID, nID)$
$? - CostMaxNation(nID, cMaxNation)$

Figure 4.2: Example Queries

propagated through subqueries in the nested (A)SQPJ query.

(ii). In case of the answers being invalidated, we explain their invalidation by finding contradictory derivations contained in the delta source tables. These contradictory derivations can include both removed tuples and inserted tuples.

This chapter is organized as follows. In Section 4.2, we briefly review the related work and its relationship to our work. In Section 4.3, we briefly review the concepts of Datalog and incremental view maintenance, which we will need in the discussion of our approach. In Section 4.4, we discuss how to construct pruning predicates and use them with incremental evaluation to validate selected answer tuples. In Section 4.5, we define the contributory and contradictory derivations of answer tuples, and explain the invalidation of them with their contradictory derivations. In Section 4.6, we report the

Orders

oID^a	cID^b	$cost$	CNT
o_1	c_1	500	1
o_2	c_2	100	1
o_3	c_2	150	1
o_4	c_3	100	1

$\Delta Orders$

oID^a	cID^b	$cost$	CNT
o_6	c_3	150	1
o_1	c_1	500	-1

$\Delta OrderCount$

cID	$oCnt$	CNT
c_1	1	-1
c_3	1	-1
c_3	2	1

$\Delta CustomerDistribution$

$oCnt$	$cCnt$	CNT
2	1	-1
1	2	-1
2	2	1

$\Delta CostMax$

cID	$cMax$	CNT
c_1	500	-1
c_3	100	-1
c_3	150	1

Figure 4.3: (Delta) Tables Extended With CNT

time cost of incrementally evaluating (nested) queries with pruning predicates. Finally, we conclude in Section 4.7.

4.2 Related Work

Several research problems are closely related to our validation problem, though each is different in crucial ways. We briefly review view maintenance, schema mapping, provenance, causality, explanation of (non-)answers, and aggregate selections.

View maintenance techniques update a view upon source data change. In [20], the count of derivations of every view tuple is tracked. The count of derivations can be either positive or negative. Negative counts track the number of derivations removed due to changes to the base tables. Given changes to the base tables as delta base tables, [20] evaluates a set of delta rules to determine the changes to this count and further decide the presence of every view tuple. Our problem can be considered as a specialization of this work where our interest is in only selected tuples in the result view.

Although view maintenance techniques can figure out if given answers are invalidated by comparing the updated view and the original view, they require too much work in our scenario, where the view is much larger than the few selected answers.

Our validation approach relies on incremental view maintenance and applies additional pruning determined by the selected answers we are trying to validate. The goal of pruning is to avoid evaluating over irrelevant tuples, i.e., the tuples that are not capable of changing the selected answers. This intuition is similar to the one leading to aggregate selection [33] or magic sets [1]. The criterion they use to determine what tuples are irrelevant only applies to positive tuples. Our pruning predicates applies to both positive and negative tuples.

Provenance (also known as lineage) and view maintenance are closely related areas. Intuitively, provenance consists of the source tuples that contribute to the derivation of the view tuple or an answer tuple [12, 15]. These contributing source tuples explains “why” [5] the answer is in the derived set. A finer grained provenance, “where” [5] provenance, can explain how a particular value in the answer is derived [14, 10, 29]. In particular, [14, 10] applies to a wider application scenario than just relational databases.

Provenance can be defined as a polynomial expression [18]. The number of terms in the polynomial is the number of derivations and the factors in each term correspond to the joined source tuples in each derivation.

Provenance can be viewed as dependency [10]. Given part of an answer, its provenance is “data slices” in the source database that the part of the answer may depend on, although minimal slices are not computable.

Causality [30] is another way of looking at the relationship between answers and the source tuples. The causes of an answer can be counterfactual or actual. The removal of counterfactual causes can invalidate an answer, while actual causes can invalidate an answer if a certain set of source tuples are removed as well. Thus, causes of an answer are the tuples that contribute to the answer’s derivation.

Provenance can be computed in either a lazy way or an eager way [35]. In a lazy approach, provenance is computed by tracing queries [12]. In an eager approach, provenance is generated with the result either through query rewriting [15], through annotation propagation [18, 14] or through recording data operation in a curated database [2]. The lazy approach may implicitly require the availability of the database at a historical time point [40]. The eager approach may require an efficient storage mechanism [8].

In [26], absorption provenance is used to efficiently determine the effect of the deletion-induced updates on a distributed and recursive view. Absorption provenance is based on the semi-ring provenance introduced in [18]. In [26], the update to the aggregate results due to the input stream data is based on incremental view maintenance and aggregate selections.

View maintenance and provenance are also closely related to schema mapping. [17] sets up an application scenario where a set of data sources exchange data with each other through schema mappings. Their model of data sharing incorporates the semi-ring provenance to enforce trust policy when propagating the update from one source to the other.

Provenance can be used to explain the derivation of answers. As a counterpart, the explanation of non-answers or missing answers [7, 29, 23, 21] is actively explored too. The explanations of non-answers or missing answers can be query-based or instance-based [21]. The former explanation consists of the operations in the derivation process that eliminate the production of expected but missing answers [7]. The latter explanation consists of the missing source tuples that can otherwise produce the missing answers according to the derivation process [23, 29, 21].

Our explanation of invalidation of answers relies on provenance generalized to negative tuples. While the instance-based explanation of non-answers aims at describing the modifications to the source that can add the non-answers into the result set, our validation problem aims at describing the modifications to source that have removed specified answers, which include insertion of new source tuples and/or removal of existing source tuples. This information is something we would like, in our problem setting.

4.3 Preliminaries

In this chapter, we represent queries in Datalog rules [6] and enhance each database tuple with a count of its derivations [20]. We very briefly revisit related syntax and semantics in this section. More details are in [6, 20].

4.3.1 Datalog

Datalog rules are of the form $R(X) :- S_1(Y_1), ..., S_n(Y_n), C(\cup_{i=1}^{i=n} Y_i)$. $R(X)$ and $S_1(Y_1), ...,$ $S_n(Y_n)$ are relational predicates, which correspond to relational tables. $C(\cup_{i=1}^{i=n} Y_i)$ is a disjunction of conjunctive predicates on the variables $\cup_{i=1}^{i=n} Y_i$ (sometimes written as $\cup_i Y_i$ for short, when the range of i is clear from context). A conjunctive predicate is a conjunction of arithmetic predicates, which are either comparisons or built-in functions. In this chapter, we consider only comparisons, e.g., $cost \geq 100$ in r_{oCnt}.

A negated predicate is a predicate modified by a negation, e.g., $\neg S_1(Y_1)$. $\neg S_1(Y_1)$ evaluates to true if and only if $S_1(Y_1)$ evaluates to false. A Datalog rule $R(X) :- S_1(Y_1), ..., S_n(Y_n),$ $C(\cup_i Y_i)$ is safe if and only if $\forall x \in X\ \exists k \in \{1, ..., m\}$ such that $x \in Y_k$ and S_k is a relational predicate and is not a negated predicate. We only consider safe rules in this chapter.

A substitution maps each variable in the rule to a constant value or a variable. A valuation is a substitution that maps each variable in the rule to a constant value. For simplicity of notation, we abuse the attribute names in a relational table as variables in the corresponding relational predicate. Given a rule $R(X) :- S_1(Y_1), ..., S_m(Y_m), C(\cup_i Y_i)$, a valuation μ of this rule is a mapping $\mu : \cup_i Y_i \rightarrow \mathbb{D}(\cup_i Y_i)$, where $\mathbb{D}(\cup_i Y_i)$ is the domain of $\cup_i Y_i$. $S_i(Y_i)[Y_i/\mu(Y_i)]$ represents a ground term resulting from substituting each variable y in Y_i with $\mu(y)$, and it evaluates to true if $\mu(Y_i)$ is a tuple in the relational table S_i. Given μ, the goal evaluates to true if and only if all the subgoals evaluate to true.

We categorize Datalog rules into aggregation-free rules (or simple rules) and aggregate rules. An aggregation-free rule is of form $R(X)$:− $S_1(Y_1), ..., S_m(Y_m), C(\cup_i Y_i)$. An aggregation rule is of the form $T(G, \Sigma(\langle A \rangle)\ AS\ \hat{A})$:− $R(Y), C(Y)$, where $G \subset Y$ and $A \subset Y$. G is the GROUPBY attribute. A is the to-be-aggregated attribute. $\hat{A}$ is the aggregate attribute and $\Sigma \in \{sum, count, avg, max, min\}$. Sometimes, we also use Σ in a form as $\Sigma(a_1, ..., a_k)$, where a_i $(i = 1, ..., k)$ are values of A and are aggregated.

In a nested query of interest to us in this chapter, the inner query's result table is used as the source table to its outer query. Represented in SQL, the inner query appears in the FROM clause of the outer query. Represented in Datalog, the goal of a rule is a subgoal of another rule. We will not consider other types of nesting, such as in the WHERE clause, following the lead of previous work, such as [12, 18].

Moreover, multiple references to the same table in a query are treated as independent instances of the table.

4.3.2 Incremental Evaluation

Given a table, its delta table describes the changes made to that table. Delta tables can contain both to-be-removed and to-be-inserted tuples [20]. Each tuple is attached with a special count, denoted as CNT, which can be positive or negative. If positive, CNT represents the number of derivations of an existing or to-be-inserted tuple; if negative, CNT represents the number of derivations to be removed for an existing tuple. In Figure 4.3, we show an example of the delta source/result tables for the example in Figure 4.1.

We assume every tuple has the count CNT. We attach CNT to a tuple using $*$ as in [20], e.g., $(o_1, c_1, 500) * -1$ and $t * -1$. Strictly speaking, CNT is not part of the table schema, i.e., not an attribute. Therefore, the comparison of tuples does not compare their CNT values. However, for easy notation, we access the CNT value of t by $t.CNT$ as if it was an attribute. We call tuples with positive CNT positive tuples, and tuples with negative CNT negative tuples.

The incremental evaluation of a query, say $r : R(X) :- S_1(Y_1), ..., S_m(Y_m), C(\cup_i Y_i)$, involves a set of delta rule. That is, $\Delta_i r : \Delta_i R(X) :- S_1 \uplus \Delta S_1, ..., \Delta S_i, ..., S_m$. Then, the updated result set is $R \uplus \Delta_1 R \uplus ... \uplus \Delta_m R$. $\uplus$ is the union of two sets of tuples with CNT. $s \in S_1(X) \uplus S_2(X)$ with a count c, if (1) $s \in S_1(X) \wedge s \notin S_2(X)$ with a count c or (2) $s \in S_2(X) \wedge s \notin S_1(X)$ with a count c or (3) $s \in S_1(X)$ with a count c_1 and $s \in S_2(X)$ with a count c_2 and $c = c_1 + c_2$.

4.4 Validation Of Answers

Given a query, a set of selected answer and a set of delta source tables, the validation of the answers and the explanation in case of invalidation can be done in the following steps:

Step 1 compute the delta result table by incrementally evaluating the (nested) query with pruning predicates;

Step 2 check the delta result table against the original result table to see if the given answers are invalidated, and explain the invalidation with the positive and/or negative tuples in the delta source tables (and possibly tuples in the original source tuples).

We discuss Step 1 in this section and discuss Step 2 in the next section. In Step 1, the key point is the construction of pruning predicates. The goal is to prune irrelevant source tuples in the (delta) source tables. The source tuples that can not possibly affect the selected answer(s) are considered irrelevant. Note that the view update results computed from the incremental evaluation with and without pruning predicates are possibly different, since the former does not care for updating answers other than the selected ones.

We first define pruning predicates for aggregation-free Datalog rules in Definition 4.2 and for aggregate rules in Definition 4.3.

Definition 4.2. [Pruning Predicate For A Single Answer Tuple According To An Aggregation-Free Rule]

Given

- a query $r : R(Y) \coloneq S_1(X_1), ..., S_m(X_m), C(\cup_i X_i)$
- an answer $t \in R(Y)$
- a set of delta tables $\Delta S_1, ..., \Delta S_m$

For $k = 1, ..., m$, let

- $\Delta_k R(Y) \coloneq S_1(X_1), ..., \Delta S_k(X_k), ..., S_m(X_m), C(\cup_i X_i)$
- $\Delta_k R^p(Y) \coloneq S_1(X_1) \uplus \Delta S_1(X_1), ..., \Delta S_k(X_k), ..., S_m(X_m), C(\cup_i X_i), C^p(\cup_i X_i)$

Then $C^p(\cup_i X_i)$ is a pruning predicate for t according to r if and only if

- $t \in \Delta_k R$ implies $t \in \Delta_k R^p$, and vice versa

Definition 4.3. [Pruning Predicate For A Single Answer Tuple According To An Aggregate Rule]

Given

- a query $r_\alpha : T(G, \Sigma(\langle A \rangle)\ AS\ \hat{A}) \coloneq R(Y)$, where $G \subset Y$ and $A \subset Y$
- an answer $t \in T(G, \hat{A})$
- a delta table ΔR

Let

- $\Delta R^p(Y) \coloneq \Delta R(Y), C^p(Y)$
- $T^p_\uplus \coloneq R \uplus \Delta R^p$ and $\Delta T^p = T^p_\uplus - T$
- $T_\uplus \coloneq R \uplus \Delta R$ and $\Delta T = T_\uplus - T$

Then $C^p(Y)$ is a pruning predicate for t according to r_α if and only if

- $t \in \Delta T$ implies $t \in \Delta T^p$, and vice versa

Definition 4.2 and Definition 4.3 define a pruning predicate that restricts the view maintenance to a given answer t. The vital requirement of C^p is that it should not affect the view update with regard to t. The pruning predicate is not unique for a given answer. Obviously, the more restrictive a pruning predicate is, the better it is. C^p being *true* is a trivial pruning predicate that does not prune any source tuples.

For example, given an answer $(c_2, 150) \in CostMax$, a possible choice of C^p is $(cost > 150 \wedge CNT > 0) \vee (cost = 150 \wedge CNT < 0)$. C^p states two cases where t could be affected by $\Delta Orders$. One case is the insertion of orders of the customer c_2 with a new maximum cost; the other case is the removal of orders of c_2 that have the current maximum cost. Another pruning predicate C^p_1 could be $(cost > 100) \wedge CNT > 0) \vee (cost = 150) \wedge CNT < 0)$. Since C^p is more restrictive than (or tighter than) C^p_1, and thus C^p is better.

Although the source tuples that do not satisfy these pruning predicates can not possibly change the given answer, the source tuples that do satisfy the pruning predicates do not necessarily change the given answer. Take $(c_2, 150) \in CostMax$ as an example again. A negative tuple $(o_2, c_2, 150) * -1 \in \Delta Orders$, which satisfies $cost = 150 \wedge CNT < 0$, has the potential to change $(c_2, 150)$, but won't if there is another order of c_2 whose cost is also 150.

Definition 4.4. [Pruning Predicate For A Set Of Answer Tuples] Given a query r that produces R, and a set of answer $t_1, ..., t_n \in R$, let C^p_k be a pruning predicate for t_k according to r $(k = 1, ..., n)$, then $C^p_1 \vee ... \vee C^p_n$ is a pruning predicate for $\{t_1, ..., t_n\}$.

Since the pruning predicates for a set of answers can be constructed from pruning predicates for each answer in the set, we will focus on constructing pruning predicates for single answers.

The construction of pruning predicates for a given answer is fairly intuitive for single Datalog rules (i.e., single-block queries) but more complicated for a set of Datalog rules (i.e., nested queries). We will first discuss enhancing incremental evaluation of a single Datalog rule with pruning predicates for a given answer, and then doing the same for a set of Datalog rules.

4.4.1 Single Aggregation-Free Datalog Rules

Algorithm 3: Incremental Evaluation Of An SPJ Query With Regard To A Given Answer Tuple

```
  Input: $r_{conj} : T(Y) :- S_1(X_1), ..., S_m(X_m), C(\cup_{i=1}^{i=m} X_i)$
  Input: $\Delta S_1, ..., \Delta S_m$
  Input: $t \in T(Y)$
  Output: $\Delta T(Y)$ relevant to $t$
1 begin
2     $\Delta T(Y) \leftarrow \emptyset$
      /* pruning predicate $C^p$                                              */
3     $C^p(Y) \leftarrow (Y = r.Y)$
      /* incremental evaluation with pruning predicate                        */
4     $\Delta T \leftarrow \emptyset$
5     for $k \leftarrow 1$ to $m$ do
6         Evaluate $\Delta_k r_{conj} : \Delta_k T(Y) :- S_1(X_1) \uplus \Delta S_1(X_1), ..., S_{k-1}(X_{k-1}) \uplus$
          $\Delta S_{k-1}(X_{k-1}), \Delta S_k(X_k), S_{k+1}(X_{k+1}, ...., S_m(X_m), C(\cup_{i=1}^{i=m} X_i), C^p(Y)$ as indicated in [20]
          $\Delta T(Y) \leftarrow \Delta T(Y) \uplus \Delta_k T(Y)$
7     end
8     return $\Delta T(Y)$
9 end
```

C^p in Algorithm 3 is a pruning predicate. In case of aggregation-free rules, the pruning logic is the same for positive tuples and negative tuples. In fact, $Y = t.Y$ is a classic way of filtering contributing source tuples with regard to a given answer tuple as used in the provenance tracing queries [12].

4.4.2 Single Aggregate Datalog Rules

In Algorithm 4, we show the construction of pruning predicates for incremental evaluation of a single aggregation. The pruning predicate C^p equals to $C^p_{groupby} \wedge C^p_{\alpha}$, and the first term is the pruning on GROUPBY attribute G and the second term is on the aggregated attribute A. $C^p_{groupby}$ being $G = t.G$ excludes the source tuples in groups other than the one producing t. For *max* and *min*, C^p_{α} specifies two cases of affecting t: the insertion of new maximums (minimums) or the removal of current maximums (minimums). For *sum*, C^p_{α}

Algorithm 4: Incremental Evaluation Of An Aggregation With Regard To A Given Answer Tuple

```
Input: $r_\alpha : T(G, \Sigma(\langle A \rangle)\ AS\ \hat{A}) :- R(G, A)$
Input: $\Delta R$
Input: $t \in T(G, \hat{A})$
Output: $\Delta T(G, \hat{A})$ with regard to $r$
begin
    /* pruning predicate $C^p_{groupby} \wedge C^p_\alpha$ */
    $C^p_{groupby}(G) \leftarrow (G = t.G)$
    if $\Sigma$ is max then
        $C^p_\alpha(A) \leftarrow (A > t.\hat{A} \wedge CNT > 0 \vee A = t.\hat{A} \wedge CNT < 0)$
    end
    if $\Sigma$ is min then
        $C^p_\alpha(A) \leftarrow (A < t.\hat{A} \wedge CNT > 0 \vee A = t.\hat{A} \wedge CNT < 0)$
    end
    if $\Sigma$ is sum then
        $C^p_\alpha(A) \leftarrow (A \neq 0)$
    end
    if $\Sigma$ is count then
        $C^p_\alpha(A) \leftarrow true$
    end
    $C^p(G, A) = C^p_{groupby}(G) \wedge C^p_\alpha(A)$
    /* incremental evaluation of aggregation */
    $\Delta R^p(Y) \leftarrow \Delta R(Y), C^p(G, A)$
    $T_\uplus \leftarrow T$
    for every $r \in \Delta R^p$ do
        if exists $t \in T_\uplus$ and $t.G = r.G$ then
            $T_\uplus \leftarrow T_\uplus \uplus \delta(t, r)$
        else
            $T_\uplus \leftarrow T_\uplus \uplus \{(r.G, \Sigma(r.A)) * r.CNT\}$
        end
    end
    /* in case that the previous maximum or minimum is removed */
    if $\Sigma$ is max or min and exists $t \in T_\uplus$ and $t.CNT < 0$ then
        Remove $t$ from $T_\uplus$
        Add to $T_\uplus$ new maximum or minimum from $R \uplus \Delta R$ with GROUPBY value being $t.G$
    end
    $\Delta T \leftarrow \emptyset$
    for every $t \in T_\uplus$ do
        if exists $t' \in T$ and $t.G = t'.G$ and $t.\hat{A} \neq t'.\hat{A}$ then
            $\Delta T \leftarrow \Delta T \uplus \{(t'.G, t'.A) * -1, (t.G, t.A) * 1\}$
        else
            $\Delta T \leftarrow \Delta T \uplus \{t\}$;
        end
    end
    return $\Delta T(G, \hat{A})$
end
```

prunes the source tuples with zero values on the aggregated attribute A. For *count*, we do not have a non-trivial pruning predicate. In other words, C^p_α is trivially true if the aggregation is *count*. This is because any source tuple that is in the group $G = t.G$ can change the count of of this group and thus affect t.

In Algorithm 4, note that C^p_α is different from the aggregate selection used in [33], since it deals with both positive and negative tuples and has different pruning logic for them. The predicate for the positive tuples in C^p_α is the same as the aggregate selection.

The aggregate selection introduced in [33] deals with recursive Datalog programs. The evaluation of a recursive program keeps adding facts into the database but never removes tuples from the already computed set of facts. Therefore, aggregate selection does not need to deal with the effect that the removal of facts might have on the given answer.

In Algorithm 4, $\delta(t, r)$ is

- if Σ is *sum*, $\{(t.G, t.\hat{A} + r.A \times r.CNT) * 1, t * -1\}$
- if Σ is *count*, $\{(t.G, t.\hat{A} + r.CNT) * 1, t * -1\}$
- if Σ is *max* or *min*
  - if $r.CNT > 0$, $\{(t.G, max(t.\hat{A}, r.A)) * 1, t * -1\}$ or $\{(t.G, min(t.\hat{A}, r.A)) * 1, t * -1\}$
  - if $r.CNT < 0$
    - $*$ if $r.A < t.\hat{A}$, $\{\}$
    - $*$ if $r.A = t.\hat{A}$, $\{t * (-t.CNT - 1)\}$

We summarize in Figure 4.4 the pruning predicates in Algorithm 3 and Algorithm 4. In particular, we formalize C^p into $C^+ \vee C^-$. C^+ in Figure 4.4 applies to positive tuples and C^- applies to negative tuples. Moreover, if we have C^p for two different attributes respectively, then the C^p for these two attributes together is the conjunction of the separate C^p, e.g., $C^p(G, A)$ is $C^p(G) \wedge C^p(A)$.

C^+ (C^-) for the aggregated attribute depends on the aggregation type. Meanwhile, C^+ (C^-) for non-aggregated attributes, such as GROUPBY attributes or output attributes of aggregation-free rules, does not depend on the aggregation type. In Figure 4.4, Σ being null means C^+ (C^-) for non-aggregated attributes.

Σ	$C^+(A)$ or $C^+(Y)$ or $C^+(G)$	$C^-(A)$ or $C^-(Y)$ or $C^-(G)$
max	$A > t.\hat{A}$	$A = t.\hat{A}$
min	$A < t.\hat{A}$	$A = t.\hat{A}$
sum	$A \neq 0$	$A \neq 0$
count	true	true
null	$G = t.G$ or $Y = t.Y$	$G = t.G$ or $Y = t.Y$

Figure 4.4: Pruning Predicates For A Single Aggregation

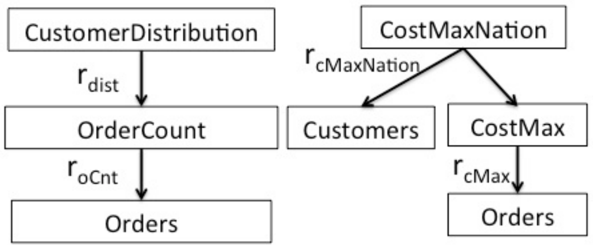
4.4.3 Set Of Datalog Rules

Consider a non-recursive negation-free Datalog program (a set of Datalog rules) where the goal of each rule appears as a subgoal in another rules except for one rule. This one rule produces our final result. This type of programs correspond to the type of nested queries specified in Section 4.3. In general, this type of Datalog programs can be translated into query trees, and we illustrate two such Datalog programs in Figure 4.5a.

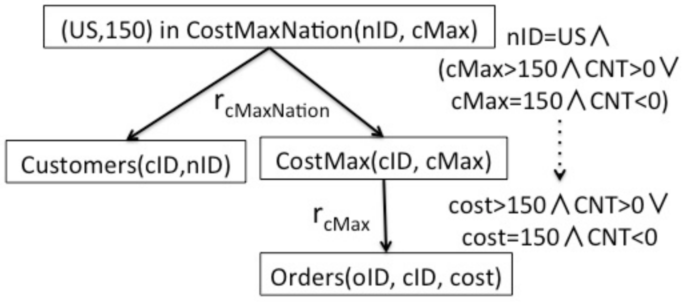
The rule in a program can be either aggregation-free or aggregate. Incremental evaluation of a Datalog program starts from the leaves in the query tree, and when every subgoal in a rule has been updated, the goal of the rule can be updated.

In a Datalog program, every rule can have its own pruning predicate. In Figure 4.5b, we show an answer $(US, 150)$ produced from $\{r_{cMax}, r_{cMaxNation}\}$, and two pruning predicates that can be applied to r_{cMax} and $r_{cMaxNation}$ respectively. There are two things that need to be noted. First, the two pruning predicates are in fact related; and second, the earlier the pruning takes place, the better it is.

In Figure 4.5b, given the answer $(US, 150)$ produced by $r_{cMaxNation}$, we can construct the pruning predicate applicable to the delta rules of $r_{cMaxNation}$ as shown in Figure 4.4.



(a) Datalog Program Examples


(US,150) in CostMaxNation(nID, cMax)
r_cMaxNation
Customers(cID,nID)
CostMax(cID, cMax)
r_cMax
Orders(oID, cID, cost)
nID=US ∧ (cMax>150 ∧ CNT>0 ∨ cMax=150 ∧ CNT<0)
cost>150 ∧ CNT>0 ∨ cost=150 ∧ CNT<0

(b) Pruning Predicates Inference Example

Furthermore, since the subgoal in $r_{cMaxNation}$ is produced from r_{cMax}, we intuitively want to try some early pruning during the incremental evaluation of r_{cMax}, which should achieve similar results as the pruning predicate for $r_{cMaxNation}$ does and thus is inferred from the one for $r_{cMaxNation}$.

Suppose $\Delta Orders$ is as shown in Figure 4.3. When computing $\Delta CostMax$ from it, the pruning predicate for r_{cMax} in Figure 4.5b causes $\Delta CostMax$ to be empty. Thus, without incrementally evaluating $r_{cMaxNation}$, we know that $(US, 150)$ will not be invalidated. This shows the advantage of earlier pruning in the incremental evaluation of a Datalog program.

In general, given a Datalog program, suppose r is the rule at the root of the corresponding query tree, we start from r and construct the pruning predicates applicable to its delta rules as shown in Figure 4.4. Suppose r has a subgoal that is the goal of r', we infer the pruning predicate applied to r' from the pruning predicate applied to r and then recursively infer pruning predicates for the subgoals in r'. We repeat this process for all other subgoals in r that are goals of other rules.

In order to construct pruning predicates by inference, we need to know how attributes propagate through the query. For example,

- r_{oCnt}:

  (i). $Orders.cID \rightarrow OrderCount.cID$

  (ii). $Orders.oID \xrightarrow{count} OrderCount.oCnt$

- r_{oCnt}, r_{dist}:

  (i). $Orders.cID \rightarrow OrderCount.cID$

  $\xrightarrow{count} CustomerDistribution.cCnt$

  (ii). $Orders.oID \xrightarrow{count} OrderCount.oCnt$

  $\rightarrow CustomerDistribution.oCnt$

- r_{cMax}:

  (i). $Orders.cID \rightarrow CostMax.cID$

  (ii). $Orders.cost \xrightarrow{max} CostMax.cMax$

- $r_{cMax}, r_{cMaxNation}$:

  (i). $Orders.cID \rightarrow CostMax.cID \rightarrow \emptyset$

  (ii). $Orders.cost \xrightarrow{max} CostMax.cMax$

  $\xrightarrow{max} CostMaxNation.cMaxNation$

  (iii). $Customer.cID \rightarrow \emptyset$

  (iv). $Customer.nID \rightarrow CostMaxNation.nID$

The mapping path of an attribute starts with an attribute in the source table and ends at an attribute in the final result. This path illustrates the dependency of the attribute in the final result on the attribute in the source table. It gives us useful clues about how to prune the source tuples with regard to this attribute using attribute values in the result tuples. Algorithm 5 shows how to construct the mapping path of every attribute in the final result.

Algorithm 5: Construction of attribute mapping paths

```
Input: A Datalog rule set: $r_1, ..., r_n$
Output: $G = (V, E)$, where $V$ is a set of attributes, $E$ is a set of edges representing mapping from an
        attribute in goal to an attribute in the subgoal of the same rule
1  begin
2      Initialize $V$ and $E$ be empty
3      for $i \in \{1, ..., n\}$ do
4          for every attribute A appearing in the goal R of $r_i$ do
5              Add $A$ to $V$
6              for every subgoal $S$ in $r_i$ do
7                  if $A$ appears in $S$ then
8                      Add $(R.A \xrightarrow{\Sigma} S.A)$ to $E$, where $\Sigma$ is null if $r_i$ is an aggregation-free rule, or the
                       aggregation of $r_i$ if $r_i$ is an aggregate rule
9                  end
10             end
11         end
12     end
13 end
```

Suppose we have a mapping path of an attribute $T_1.A_1 \xrightarrow{r_1} T_2.A_2 \xrightarrow{r_2} ... \xrightarrow{r_{n-1}} T_n.A_n$. r_i has T_i as a subgoal and T_{i+1} as the goal. It can be an aggregation-free rule or an aggregate rule. We infer the pruning predicates backward from the end of the path. First, given an answer $t \in T_n$ and r_{n-1}, we construct $C^p(A_{n-1})$ as in the single Datalog rule case. Then, given $C^p(A_k)$ and r_{k-1}, we can infer $C^p(A_{k-1})$. In particular, the inference process is as follows:

(i). $C^p(A_k)$ is decomposed into $C^+(A_k)$ and $C^-(A_k)$;

(ii). from $C^+(A_k)$, we infer $C_1^+(A_{k-1})$ and $C_1^-(A_{k-1})$;

(iii). from $C^-(A_k)$, we infer $C_2^+(A_{k-1})$ and $C_2^-(A_{k-1})$;

(iv). $C^+(A_{k-1})$ is $C_1^+(A_{k-1}) \vee C_2^+(A_{k-1})$;

(v). $C^-(A_{k-1})$ is $C_1^-(A_{k-1}) \vee C_2^-(A_{k-1})$;

(vi). $C^p(A_{k-1})$ is $C^+(A_{k-1}) \vee C^-(A_{k-1})$.

We show the inference for the aggregation *max* in Figure 4.5a and Figure 4.5b, for the aggregation *min* in Figure 4.6a and Figure 4.6b.

$C^+(A_{k+1})$	$C_1^+(A_k)$	$C_1^-(A_k)$
$A_{k+1} = c$	$A_k = c$	$A_k > c$
$A_{k+1} > c$	$A_k > c$	$A_k > c$
$A_{k+1} < c$	$A_k < c$	true[a]
$A_{k+1} \neq c$	$A_k \neq c$	true

[a]: true means no pruning

(a) Inference From $C^+(A_{k+1})$

$C^-(A_{k+1})$	$C_2^+(A_k)$	$C_2^-(A_k)$
$A_{k+1} = c$	$A_k > c$	$A_k = c$
$A_{k+1} > c$	$A_k > c$	$A_k > c$
$A_{k+1} < c$	$A_k < c$	$A_k < c$
$A_{k+1} \neq c$	true	true

(b) Inference From $C^-(A_{k+1})$

$C^+(A_{k+1})$	$C_1^+(A_k)$			$C_1^-(A_k)$		
	①	②	③	①	②	③
$A_{k+1} = c$	$A_k = c$	false[b]	$A_k = c$	false	$A_k > c$	false
$A_{k+1} > c$	$A_k > c$	$A_k > c$	$A_k > c$	false	$A_k > c$	false
$A_{k+1} < c$	false	false	$A_k < c$	$A_k = c$	$A_k > c$	$A_k < c$
$A_{k+1} \neq c$	$A_k > c$	$A_k > c$	$A_k \neq c$	$A_k = c$	$A_k > c$	$A_k < c$

[b]: false means all input tuples are pruned

(c) Explanation Of Inference From $C^+(A_{k+1})$

$C^-(A_{k+1})$	$C_2^+(A_k)$			$C_2^-(A_k)$		
	①	②	③	①	②	③
$A_{k+1} = c$	$A_k > c$	false	false	$A_k = c$	false	false
$A_{k+1} > c$	false	$A_k > c$	false	false	$A_k > c$	false
$A_{k+1} < c$	false	false	$A_k < c$	false	false	$A_k < c$
$A_{k+1} \neq c$	false	$A_k > c$	true	$A_k = c$	$A_k > c$	$A_k < c$

(d) Explanation Of Inference From $C^-(A_{k+1})$

①:	c is equal to current maximum of A_k
②:	c is less than current maximum of A_k
③:	c is greater than current maximum of A_k

Figure 4.5: Inference Of Pruning Predicates (*max*)

Every row in Figure 4.5a shows three predicates $C^+(A_{k+1})$, $C_1^+(A_k)$ and $C_1^-(A_k)$. Given $C^+(A_{k+1})$, we try to find $C_1^+(A_k) \vee C_1^-(A_k)$ such that the tuples in ΔT_k that satisfy $C^+(A_{k+1})$ will still in ΔT_k after $C^+(A_k) \vee C^-(A_k)$ are applied to the delta rules of r_k. We now explain the first row in Figure 4.5a as an example. Since A_{k+1} is the current maximum of A_k, the predicate $A_{k+1} = c$ means that the new maximum is c. Therefore, we have three cases to consider: c is equal to or greater than or less than the current maximum of A_k.

(i). when c is equal to current maximum of A_k

- the new tuples inserted into T_k that satisfy $A_k = c$ can possibly generate a new maximum being c (① under $C_1^+(A_k)$ column in Figure 4.5c)
- the removal of existing tuples from T_k does not have the potential to produce

a new maximum equal to c, and thus $C_1^-(A_k)$ being $false$ (① under $C_1^-(A_k)$ in Figure 4.5c)

(ii). when c is less than current maximum of A_k

- the new tuples inserted into T_k can not change the current maximum to a smaller value, and thus $C_1^+(A_k)$ being $false$ (② under C_1^+k in Figure 4.5c)
- the removal of tuples from T_k with $A_k > c$ can potentially produce a new maximum equal to c (② under $C_1^-(A_k)$ in Figure 4.5c)

(iii). when c is greater than the current maximum of A_k

- the inserted new tuples in T_k should satisfy $A_k = c$ (③ under $C_1^+(A_k)$ in Figure 4.5c)
- the removal of tuples from T_k can not produce a greater maximum value, and thus $C_1^-(A_k)$ being $false$ (③ under $C_1^-(A_k)$ in Figure 4.5c)

Then, $C_1^+(A_k)$ ($C_1^-(A_k)$) is the disjunction of the predicates from the three cases. Therefore, $C_1^+(A_k)$ is $(A_k = c) \vee (A_k = c) \vee (A_k = c)$ and thus is normalized to $A_k = c$; $C_1^-(A_k)$ is $false \vee A_k > c \vee false$ and thus is normalized to $A_k > c$.

For all the other rows in Figure 4.5a and Figure 4.5b, the detailed inference is similar, as shown in Figure 4.5c and Figure 4.5d respectively.

In Figure 4.5a, Figure 4.5b, Figure 4.6a and Figure 4.6b, the aggregations are either *max* or *min*. The inference of pruning predicates when the aggregation is *sum* or *count* is quite different. Whether an input source tuple to *sum* or *count* can potentially contribute to the aggregate value for a given group of source tuples is constrained by that source tuple's value on GROUPBY attribute. As a contrast, it is not constrained by that source tuple's value on the to-be-aggregated attribute. Therefore, there is no pruning on the to-be-aggregated attribute of source tuples as an input to *sum* or *count*. That is to say, given

$C^p(A_{k+1})$ and the aggregation in r_k being *count* or *sum*, $C^p(A_k)$ is trivially *true*, i.e., no pruning.

Moreover, given $C^p(A_{k+1})$ and r_k being aggregation-free, $C^p(A_k)$ is the same as $C^p(A_{k+1})$ with the attribute A_{k+1} replaced with A_k.

$C^+(A_{k+1})$	$C_1^+(A_k)$	$C_1^-(A_k)$
$A_{k+1} = c$	$A_k = c$	$A_k < c$
$A_{k+1} > c$	$A_k > c$	true
$A_{k+1} < c$	$A_k < c$	$A_k < c$
$A_{k+1} \neq c$	$A_k \neq c$	true

(a) Inference From $C^+(A_{k+1})$

$C^-(A_{k+1})$	$C_2^+(A_k)$	$C_2^-(A_k)$
$A_{k+1} = c$	$A_k < c$	$A_k = c$
$A_{k+1} > c$	true	$A_k > c$
$A_{k+1} < c$	$A_k < c$	$A_k < c$
$A_{k+1} \neq c$	$A_k \neq c$	true

(b) Inference From $C^-(A_{k+1})$

$C^+(A_{k+1})$	$C_1^+(A_k)$			$C_1^-(A_k)$		
	①	②	③	①	②	③
$A_{k+1} = c$	false	$A_k = c$	false	false	false	$A_k < c$
$A_{k+1} > c$	false	$A_k > c$	false	$A_k = c$	$A_k > c$	$A_k < c$
$A_{k+1} < c$	$A_k < c$	$A_k < c$	$A_k < c$	false	false	$A_k < c$
$A_{k+1} \neq c$	$A_k < c$	$A_k > c$	$A_k < c$	$A_k = c$	$A_k > c$	$A_k < c$

(c) Explanation Of Inference From $C^+(A_{k+1})$

$C^-(A_{k+1})$	$C_2^+(A_k)$			$C_2^-(A_k)$		
	①	②	③	①	②	③
$A_{k+1} = c$	$A_k < c$	false	false	$A_k = c$	false	false
$A_{k+1} > c$	false	true	false	false	$A_k > c$	false
$A_{k+1} < c$	false	false	$A_k < c$	false	false	$A_k < c$
$A_{k+1} \neq c$	false	$A_k \neq c$	$A_k < c$	$A_k = c$	$A_k > c$	$A_k < c$

(d) Explanation Of Inference From $C^-(A_{k+1})$

①:	c is equal to current maximum of A_k
②:	c is less than current maximum of A_k
③:	c is greater than current maximum of A_k

Figure 4.6: Inference Of Pruning Predicates (*min*)

We have shown all the necessary components in the procedure of incrementally evaluating a Datalog program with pruning predicates, and now summarize below the procedure given a set of selected answers.

Construction of attribute mapping paths For every attribute in the final result, we build a mapping path for it as shown in Algorithm 5;

Construction of pruning predicates For the selected set of answers, we build a pruning predicate from this set of answers according to the rule at the root of the corresponding query tree as shown in Figure 4.4;

Inference of pruning predicates For all rules other than the root, we process in a breadth-first traversal order, and infer its pruning predicate from the pruning predicate of its parent rule;

Evaluation Of Delta Rules For every delta rule, if a subgoal $T(X)$ is in form of $\Delta T(X)$, then for every attribute A in X, we add the predicate $C^p(A)$ to the delta rule's body; if a subgoal $T(X)$ is in the form of $T(X)$ or $T(X) \uplus \Delta T(X)$, for every attribute A in T, we add the predicate $C^+(A)$ to the delta rule's body. Then, the delta rules with pruning predicates can be evaluated as indicated in [20].

4.5 Explanation of Invalidated Answers

Given a Datalog program consisting of rules $r_1, ..., r_n$, and an answer t in the result set T, after the incremental evaluation with proper pruning predicates, we have a delta table ΔT. If $t' \in \Delta T$ with $t'.CNT + t.CNT \leq 0$ and $t = t'$, we know that t is invalidated; otherwise, t is still valid and $t.CNT$ is also updated to $t'.CNT + t.CNT$. If the answer $t \in T$ is invalidated, we need to find the explanation of its being invalidated.

The explanation of the invalidation can consist of (1) the positive tuples (to-be-inserted tuples) in the delta source tables, (2) negative tuples (to-be-removed tuples) in the delta source tables, (3) positive tuples (existing tuples) in the original source tables. Any such tuples both *contradict* the previous answer t as a positive tuple in T and *contribute* to the invalidated answer t' as a negative tuple in ΔT. In other words, the contributory derivations of the invalidated answer t' are the contradictory derivations of the answer t. Moreover, from the viewpoint of the answer t, it has contributory derivations and contradictory derivations, while the former can derive the answer (i.e., traditional provenance) and the latter can invalidate the answer.

Recall the example in Figure 4.3, the delta source table $\Delta Orders$ leads to the delta tables $\Delta OrderCount$, $\Delta CustomerDistribution$, $\Delta CostMax$ and $\Delta CostMaxNation$, which can be computed by the incremental view maintenance technique, e.g., [20].

The tuples in $\Delta OrderCount$, $\Delta CustomerDistribution$ and $\Delta CostMax$ with negative CNT are the answers that are invalidated by $\Delta Orders$. For example, $(c_1, 1, 1) \in OrderCount$ is invalidated because of $(o_1, c_1, 500, -1) \in \Delta Orders$ and correspondingly $(c_1, 1, -1) \in \Delta OrderCount$ is produced because of $(o_1, c_1, 500, -1) \in \Delta Orders$. In other words, $(o_1, c_1, 500, -1) \in \Delta Orders$ *contributes* to $(c_1, 1, -1)$ in $\Delta OrderCount$, and *contradicts* $(c_1, 1, 1)$ in $OrderCount$.

As another example, $(c_3, 100, 1) \in CostMax$ is invalidated by $(o_6, c_3, 150, 1) \in \Delta Orders$ and its replacement $(c_3, 150, 1) \in \Delta CostMax$ (its invalidated version $(c_3, 100, -1) \in \Delta CostMax$) is produced by $(o_6, c_3, 150, 1) \in \Delta Orders$.

This explanation of invalidated answers can be interpreted as derivations of the invalidated answers if we define derivations in a way such that

(i). both positive tuples, i.e., valid answers in the (delta) result tables tuples, and negative tuples, i.e., invalidated answers in the delta result tables, can have derivations;

(ii). the derivations can consist of positive tuples, i.e., inserted tuples in the delta source tables or existing tuples in the source tables, and negative tuples, i.e., removed tuples in the delta source tables.

4.5.1 Definitions Of Contributory And Contradictory Derivations

Definition 4.5. [Contributory Derivation (SPJ)]

Given

- $r_Q : R(Y) :- S_1(X_1), ..., S_m(X_m), C(\cup_{i=1}^{i=m} X_i)$ where
  - X_i and Y are lists of variables or constants

- $Y \subseteq \cup_{i=1}^{i=m} X_i$
- C is a disjunction of conjunctive predicate

For $t \in R$, suppose there is a valuation $\mu : \cup_{i=1}^{i=m} X_i \rightarrow \mathbb{D}(\cup_{i=1}^{i=m} X_i)$ such that

- $\mu(Y) = t.Y$
- $S_i(X_i)[X_i/\mu(X_i)]$ evaluates to true, where $i = 1, ..., m$[2]
- $C(\cup_{i=1}^{i=m} X_i)[\cup_{i=1}^{i=m} X_i/\mu(\cup_i X_i)]$ evaluates to true

Then, $\mu(X_1), ..., \mu(X_m)$ constitute a contributory derivation of t according to or with regard to r_Q within $S_1, ..., S_m$.

In Definition 4.5, the derivation μ maps the variable in $\cup_{i=1}^{i=m} X_i$ to a constant value. Thus, $\mu(X_i)$ is a tuple in S_i, and $\mu(X_1), ..., \mu(X_m)$ can derive t. We assume set semantics in the evaluation of queries, therefore, given an answer, the valuation μ is not unique.

The contributory derivations of a set of answers $t_1, ..., t_n$ are the union of the contributory derivations of each answer in the set. Suppose there is a set of answers $t_1, ..., t_n$, and μ_i is a contributory derivation of t_i according to r_Q within $S_1, ..., S_m$, then a contributory derivation of $t_1, ..., t_n$ is $\mu_1(X_1), ..., \mu_1(X_m), , ..., \mu_n(X_1), ..., \mu_n(X_m)$ according to r_Q within $S_1, ..., S_m$. If there are duplicate tuples in that set of tuples, the duplicates can be simply removed.

Definition 4.6. [Contributory Derivation (Aggregation)]

Given

- $r_Q \ : \ T(G, \Sigma(\langle A \rangle) \ AS \ \ \hat{A}) \ :- \ R(Y)$, where G is the GROUPBY attribute, $\hat{A}$ is the resulting aggregate attribute and $\Sigma \in \{sum, count, avg, max, min\}$

For an answer $t \in T$, suppose that there exists a set of valuations $\mu_1, ..., \mu_n : X \rightarrow \mathbb{D}(Y)$ $(n \geq 1)$ such that

[2]$[X_i/\mu(X_i)]$ means the substitution of variables in X with the corresponding values in $p.X_i$

(i). $\mu_i(G) = t.G$ where $i = 1, ..., n$

(ii). $R(Y)[Y/\mu_i(Y)]$ evaluates to true, where $i = 1, ..., n$

(iii). $t.\hat{A} = \Sigma(\mu_i(A), ..., \mu_n(t_n))$

(iv). $\nexists \mu_{n+1}$ such that $\mu_1, ..., \mu_{n+1}$ satisfy the above conditions.

Then, $\mu_1(Y), ..., \mu_n(Y)$ constitute a contributory derivation of t according to or with regard to r_Q within R.

In Definition 4.6, every valuation μ_i ($i = 1, ..., n$) will produce a tuple in R and the values of A in these produced tuples are aggregated within each group by the values of G.

The contributory derivation of an answer produced by an aggregation is unique. Take *count* as an example, all the valuations that map G to $t.G$ constitute the contributory derivation; for *max*, all substitutions that map G to $t.G$ and map A to $t.A$ constitute the contributory derivation.

The contributory derivation of a set of answers produced by an aggregation is the union of the contributory derivations of each answer in the set.

Since contributory derivations are transitive, given a general ASPJ query, we can always decompose it into SPJ blocks and aggregations.

Definition 4.5 and Definition 4.6 are classical definitions as introduced in [12]. They apply to tuples produced through normal queries. During the incremental evaluation of a query, the source delta tables and the result delta table can contain negative tuples, and the following definitions precisely specify the contributory derivations of the negative tuples in the delta result table, which is also the contradictory derivations of an existing answer of the original query.

Definition 4.7. [Contradictory Derivation (SPJ)]

Given

- $r_Q : R(Y) :- S_1(X_1), ..., S_m(X_m), C(\cup_{i=1}^{i=m} X_i)$
- $\Delta S_1, ..., \Delta S_m$

Let

- $\Delta_k r_Q : \Delta_k R(Y) :- S_1(X_1) \uplus \Delta S_1(X_1), ..., \Delta_k S_k(X_k), ..., S_m(X_m), C(\cup_{i=1}^{i=m} X_i)$, where $k = 1, ..., m$

Then, for an answer $t \in R$, if there exists $t_k \in \Delta_k R$ with $t_k.CNT < 0$, and $t_k.Y = t.Y$ ($k = 1, ..., m$), then a contributory derivation of t_k according to $\Delta_k r_Q$ within $S_1 \uplus \Delta S_1, ..., \Delta_k S_k, ..., S_m$ is a contradictory derivation of t according to r_Q within $S_1 \uplus \Delta S_1, ..., \Delta_k S_k, ..., S_m$.

In Definition 4.7, $\Delta_k r_Q$ ($k = 1, ..., n$) are the delta rules in the incremental evaluation of r_Q. These delta rules compute how many derivations of an existing answer are removed due to the delta source tables, and how many derivations of an existing answer are added due to the delta source tables, and also what new answers are produced due to the delta source tables. A negative tuple $t' \in \Delta_k R$ means there is a corresponding existing answer t and some or all of its derivations are removed due to the delta source tables. For the existing answer t, if $t.CNT + \Sigma_{k=1}^{k=m} t_k.CNT \leq 0$, then t is invalidated; otherwise, t is still valid but have less derivations, i.e., $t.CNT + \Sigma_{k=1}^{k=m} t_k.CNT$ derivations instead of $t.CNT$ derivations after the insertion of delta source tables. Since the negative tuple t_k ($k = 1, ..., m$) is produced through delta rules, we can trace its contributory derivations according to the delta rules and its contributory derivations contradict the existing answer t.

Definition 4.8. [Contradictory Derivation (Aggregation)]

Given

- $r_Q : T(G, \Sigma(A)\ AS\ \hat{A}) :- R(Y)$
- $\Delta R(Y)$

Let

- $r'_Q : T'(G, \Sigma(A)\ AS\ \hat{A}) :- R(Y) \uplus \Delta R(Y)$
- $\Delta r_Q : \Delta T(G, \Sigma(A)\ AS\ \hat{A}) :- \Delta R(Y)$

For an answer $t \in T$ and $t \notin T'$, there must exist $t' \in \Delta T$ with $t'.G = t.G$ and $t'.CNT < 0$. Thus, the contributory derivation of t' according to Δr_Q within $\Delta R(Y)$ is the contradictory derivation of t.

In Definition 4.8, the tuple t is invalidated due to the positive or negative tuples in ΔR that have the same value of GROUPBY attribute as t does. Therefore, t' in Definition 4.8 must exist and the contributory derivation of t' consists of all the tuples (positive or negative) that have the same value of GROUPBY attribute as t does and thus is the contradictory derivation of t.

Since the contradictory derivation of an answer tuple in the result table is converted to a contributory derivation of a negative tuple in the delta result table, the properties of contributory derivations are transferred to contradictory derivations, in particular, the union of derivations and the transitivity of derivations.

The contributory derivations can be unioned. If $\mu_1, ..., \mu_n$ constitute a contributory derivation of t according to r_Q and $\sigma_1, ..., \sigma_m$ constitute a contributory derivation of s according to r_Q, then $\mu_1, ..., \mu_n, \sigma_1, ..., \sigma_m$ constitute a contributory derivation of $\{t, s\}$ according to r_Q. If some μ_i and some σ_j are identical, we simply remove one of them from the union.

The contributory derivations are transitive. Suppose $r_{Q1} : R(Y) :- S_1(X_1), ..., S_m(X_m), C_1(\cup_{i=1}^{i=m} X_i)$ and $r_{Q2} : S_1(X_1) :- T_1(Z_1), ..., T_n(Z_n), C_2(\cup_{i=1}^{i=n} Z_i)$. If $\mu_1, ..., \mu_{k_1}$ constitute a contributory derivation of r according to r_{Q1}, and $\sigma_1, ..., \sigma_{k_2}$ constitute a contributory derivation of $\mu_1.X_1$ according to r_{Q2} then $\tau_{i,j}$ ($i = 1, ..., k_1$ and $j = 1, ..., k_2$) constitute a contributory derivation of t according to $r_{Q2} \circ r_{Q1}$, where $\tau_{i,j}$ is μ_i with replacement of the mappings of X_1 with mappings of $\cup_{i=1}^{i=k_2} Z_i$.

4.5.2 Explanation Of Invalidation As Contradictory Provenance

The contributory derivations of the negative t within the intermediate tables and the source tables can measure the tightness of the pruning predicates. As mentioned before, the pruning predicates are not unique. Even if we construct pruning predicates to try to prune as many irrelevant source tuples as possible, there may remain always irrelevant source tuples that are not pruned by the pruning predicates. Intuitively, the tighter the pruning predicates are, the better they are. All the relevant source tuples are included in the derivation of t, and thus can be considered as the ideal result of pruning. Therefore, the tightness of a pruning predicate can be measured by comparing the set of source tuples retained by the pruning predicate and the derivation.

Definition 4.9. [Tightness Of Pruning Predicates] Suppose $r_Q : R(Y) :- S_1(X_1), ..., S_m(X_m), C(\cup_i X_i)$ and $t \in R$. Further suppose $r_{Q'} : R'(Y) :- S_1 \uplus \Delta S_1, ..., S_m \uplus \Delta S_m, C(\cup_i X_i)$ and $t \in \Delta^- R(Y)$, where $\Delta^- R(Y) = R(Y) - R'(Y)$. If the contributory derivations of t according to $r_{Q'}$ are $\mu_1, ..., \mu_n$, then the tightness of a pruning predicate $C^p(\cup_i X_i)$ is defined as

- $|R^{pruned}(\cup_i X_i)|/|R^{prov}(\cup_i X_i)|$

where

- $R^{pruned}(\cup_i X_i) :- S_1 \uplus \Delta S_1(X_1), ..., S_m \uplus \Delta S_m(X_m), C(\cup_i X_i), C^p(\cup_i X_i)$ and
- $R^{prov} = \cup_{j=1}^{j=n} \{\mu_j(\cup_i X_i)\}$

The ideal pruning predicate has a tightness equal to 1. Less tight pruning predicates have tightness measurements greater than 1.

For single-block (A)SPJ queries, the pruning predicates constructed according to Figure 4.4 are always ideal, i.e. have tightness measurements equal to 1. For nested (A)SPJ queries, given an answer t, we have the following cases where the pruning predicates constructed according to Figure 4.4 and inference are known to have tightness measurements equal to 1:

(i). Every attribute in t has a mapping path free of aggregations;

(ii). Every attribute in t has a mapping path involving at most one aggregation at the very beginning of the path.

Suppose one rule in the nested query is $r : T(Y) :- S_1(X_1), ..., S_m(X_m), C(\cup_i X_i)$. Further suppose $T' \subseteq \Delta T$. Let $\Delta_k r : \Delta_k T(Y) :- S_1(X_1) \uplus \Delta S_1(X_1), ..., \Delta S_k(X_k), ..., S_m(X_m), C(\cup_i X_i), C^p$.

For every $t \in T'$, if $t \in \Delta_k T$, we retrieve the provenance of t within S_i using classical tracing rules [12]

- $\Delta_k^{trace} r : \Delta_k^{trace} S_i(X_i) :- S_1(X_1) \uplus \Delta S_1(X_1), ..., \Delta S_k(X_k), ..., S_m(X_m), C(\cup_i X_i), C^p, Y = t.Y$, if r is a conjunctive rule;
- $\Delta_k^{trace} r : \Delta_k^{trace} S_i(X_i) :- S_1(X_1) \uplus \Delta S_1(X_1), ..., \Delta S_k(X_k), ..., S_m(X_m), C(\cup_i X_i), C^p, G = t.G$, where $G \subset Y$ is the GROUPBY attributes, if r is an aggregation rule.

Then, the contributory derivations of T' within $S_1, ..., S_m$ are $\uplus_{k=1}^{k=m} \Delta_k^{trace} S_i$. And if S_i is a goal of another rule, we further trace the provenance of $\uplus_{k=1}^{k=m} \Delta_k^{trace} S_i$ according to that rule.

The retrieval of explanations can be made more efficient if we store (delta) intermediate tables during the validation with pruning predicates. During the validation with pruning predicates, the sizes of delta intermediate tables are reduced to some degree depending on how tight the pruning predicates are. Storing these (delta) intermediate tables will avoid their re-computation during the retrieval of provenance and also reduce the input table sizes on which the delta rules are evaluated. After the explanation of the invalidated answer is found, these stored delta intermediate tables can be removed if desired.

4.6 Experiments

In this section, we experimentally evaluate the construction of pruning predicates and the time cost of validating a set of selected answers with the pruning predicates.

4.6.1 Experiment Set Up

Data Set

In our experiments, we use the database schema and query templates described in the TPC-H benchmark. The relational database consists of 8 tables, *part*, *supplier*, *partsupp*, *orders*, *customer*, *lineitem*, *nation*, *region*. The data for these tables are generated by DBGen package with scaling factor 1 for the data set #1, #2 and #3, with scaling factor 10 for the data set #4. The delta tables $\Delta lineitem$ and $\Delta orders$ are also generated by DBGen and targeting at one hundred thousandth of their respective source tables for the data set #1, and one thousandth for the data set #2 and #4, and five thousandth for the data set #3. The information of these resulting data sets is specified in Section 4.6.1.

TPC-H specifies 22 ASPJ query templates. We eliminate the ones with sublinks (nested subqueries in WHERE clause) and the ones with keywords like "case", "when", etc. to get 9 remaining templates. One query is generated from each of these query templates. We refer to them by their query IDs assigned by QGen package in TPC-H, i.e, Q_1, Q_3, Q_5, Q_6, Q_7, Q_9, Q_{10}, Q_{13} and Q_{19}. Since these queries do not have a *max* or *min* aggregation, we change the *sum* aggregation in Q_5 to *max*. Their basic information is shown in Figure 4.7. In the number of result tuples columns in this figure, there are two numbers shown, separated by a comma, the first one is on the data set #1, #2 and #3, and the other is on the data set #4.

QueryID	# of Source Tables	# of Total Blocks	# of ASPJ Blocks	Aggregation Type	# of Result Tuples	# of C^+, C^-
Q_1	1	1	1	*sum*,*avg*,*count*	4,4	5
Q_3	3	1	1	*sum*	11620,114003	4
Q_5	6	1	1	*max*	5,5	2
Q_6	1	1	1	*sum*	114160,1139264	1
Q_7	6	2	1	*sum*	4,4	4,4
Q_9	6	2	1	*sum*	175,175	3,3
Q_{10}	4	1	1	*sum*	37967,381150	8
Q_{13}	1	2	2	*count*	42,46	1,0
Q_{19}	2	1	1	*sum*	121,1134	1

Figure 4.7: Basic Information of Experimental Queries And The Number Of Pruning Predicates

Performance Metrics

As for the construction of pruning predicates, we report the number of atomic comparison predicate in the pruning predicate for every query block in the query. This number depends on the query only, and is not affected by the selected answers.

As for the validation with pruning predicates, we experiment on four data sets and four different sizes of the selected answers. The source tables in the data set #1, #2 and #3 are the same, whose size is around 1GB. The delta tables in the data set #1, #2 and #3 are increasing in size, being 24KB, 1MB, 5MB respectively. The source tables' size in data set #4 is around 10GB, and the delta tables' size in data set #4 is around 10MB.

For each data set and each experimental query, we experiment on four different sizes of the selected answers of the query. The proportion of the selected answers out of the entire result set affects the comparison of validation with pruning predicates and the incremental view maintenance. Given a query, we try 4 different portions of selected answers, i.e., (1) one tuple in its result set, (2) 10% of its result set, (3) 50% of its result set and (4) the whole result set. Thus, we can observe how the time cost of validation with pruning predicates grows with the proportion.

Experimental Parameters And Methods

Our system configuration is: (1) Intel Core i7 CPU; (2) Ubuntu Release 11.10; (3) Java version 1.6.0_23; (4) PostgreSQL 8.4.4 64-bit, with "share_buffer" set to be 16MB and "effective_cache_size" set to be 128MB. The "shared_buffer" parameter determines how much memory is dedicated to PostgreSQL use for caching data [19]. The "effective_cache_size" is an estimate of how much memory is available for disk caching by the operating system and within the database itself, after taking into account what's used by the OS itself and other applications [19]. This value is used by the PostgreSQL query planner to figure out whether plans it's considering would be expected to fit in RAM or not [19].

All our measurements in this section are taken under the following assumptions: (1) indexes are build on primary keys, foreign keys and grouping attributes in the source tables, delta tables, and the result tables; (2) "echo 3 > /proc/sys/vm/drop_caches" is run between two executions of the same query to clean the system cache and the PostgreSQL cache; (3) the execution time (in seconds) of a query is measured by the module "pg_stat_statements" in PostgreSQL, and the reported time is always an average of 3 runs; (4) due to the large variance of the time/space cost of different queries, all the figures reported in the following experiments are *natural logarithm* of the actual costs.

4.6.2 Experimental Results

Experiments On Constructing Pruning Predicates

The nine queries are divided into two groups: queries without inline views and queries with them. Q_1, Q_3, Q_5, Q_6, Q_{10}, Q_{19} are queries without inline views (i.e., single-block queries) and Q_7, Q_9, Q_{13} are queries with inline views (i.e., nested queries).

In Figure 4.7, under the table column C^+/C^-, we show the number of atomic comparison predicates in the pruning predicate of each query block in an experimental query.

In general, the numbers of predicates for positive tuples and negative tuples are not necessarily the same except for some particular cases. However, if the query consists of a single block, then these numbers are the same. If the query consists of multiple blocks, these numbers for inner blocks depend on the inference shown in Figure 4.5a, Figure 4.5b, Figure 4.6a and Figure 4.6b. In Figure 4.7, the numbers happen to be the same for our experimental queries.

Experiments On Single-Block Queries

For each query without inline views and each data set, we do the following steps.

(i). Execute each query on the tables $customer$, $supplier$, $part$, $nation$, $region$, $partsupp$, $orders$, $lineitem$.

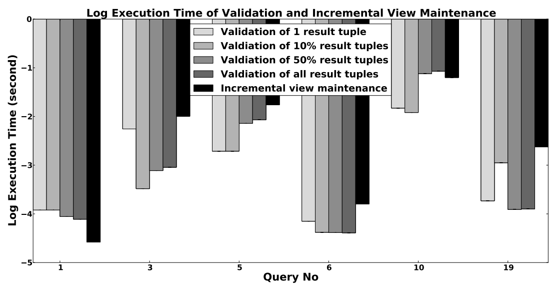
(ii). Pick four subsets of randomly selected derived tuples from the query result set: a subset consisting of one derived tuple, a subset consisting of around 10% of the result set, a subset of around 50% of the result set, and a subset that is in fact the entire result set. When the 10% or 50% size is less than 1 tuple, we round it to 1.

(iii). Incrementally evaluate each query with the pruning predicates.

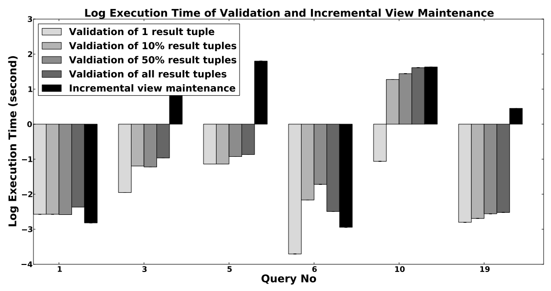
(iv). As a comparison, incrementally evaluate the complete result set without pruning predicates.

The time costs of the (4) and (5) for comparison are reported in Figure 4.8a, Figure 4.8b, Figure 4.8c and Figure 4.8d for the three different data sets respectively.

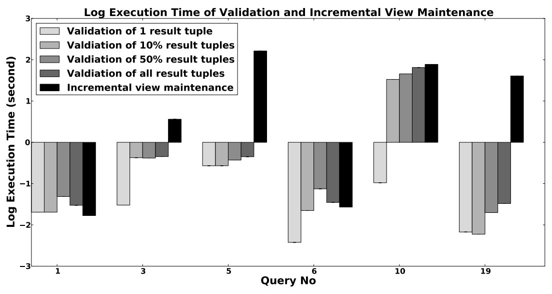
For Q_3, Q_5, Q_{10}, Q_{19}, the time costs of evaluation with pruning predicates are significantly smaller than the time cost of evaluation without pruning predicates (note that the time cost is shown in natural logarithm), even when the selected answers constitute the entire result set. These queries also show a pattern that the time cost grows with the portion of selected answers in Figure 4.8b, Figure 4.8c and Figure 4.8d. However, in Figure 4.8a,



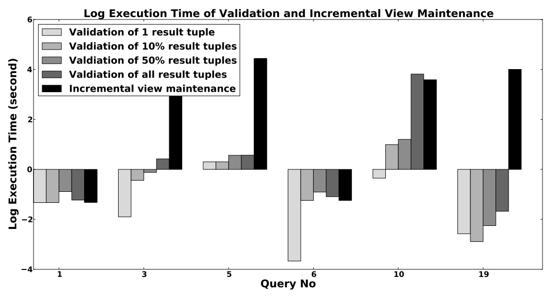
(a) Experiments On The Data Set #1



(b) Experiments On The Data Set #2



(c) Experiments On The Data Set #3



(d) Experiments On The Data Set #4

Figure 4.8: Experiments On Queries Without Inline Views

when the delta tables are very small compared to the source tables, the growth pattern is less obvious for these queries, but the time costs of validating with pruning predicates are still significantly smaller than the incremental view maintenance in most cases of the selected answers.

For Q_1, it actually shows the overhead of the pruning predicates due to the fact that it involves a single source table. When the number of selected answers is small, the comparison predicates in the pruning predicate for these answers can be explicitly written out. When the selected answers consist of hundreds of answers, the equality comparisons are implemented as natural joins. Thus, the validation with pruning predicates has one more join than the incremental view maintenance. Joins are expensive operators, and therefore may cause the performance of evaluation with pruning predicates inferior to the performance of evaluation without pruning predicates, especially when the number of joins in the original queries is small or zero, e.g., Q_1. Therefore, Q_1 demonstrates the overhead of the pruning predicates.

For Q_6, the validation with pruning predicates is much more efficient than the incremental view maintenance when the selected answers' size is less than 50% of the entire result set on all the experimental data sets. Moreover, regarding the data set #2, #3 and #4, when the portion of selected answers is greater than 50%, it is more efficient to validate the selected answers by updating the complete result set through incremental view maintenance. The presence of this critical point may be due to two facts: (1) Q_6 has one source table and (2) Q_6 has a large result set. Because of the first fact, similar to Q_1, the overhead of implementing pruning predicates as a join dominates. However, because of the second fact, when the portion of selected answers is very low, the difference in size of the selected answers and the entire result set is large and the resulting pruning outweights the overhead of the extra join. As a comparison, Q_1 has only four result tuples.

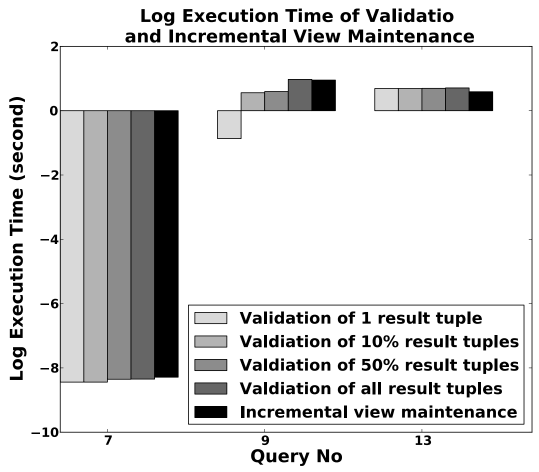
In general, the advantage of validation with pruning predicates over the incremental view maintenance becomes more obvious when the result set's size becomes larger and the number of source tables becomes larger.

Moreover, for each query in $Q_1, Q_3, Q_5, Q_6, Q_10, Q_{19}$, its behavior according to different sizes of the selected answers is consistent across Figure 4.8b and Figure 4.8c and Figure 4.8d. This means that this behavior is determined more by the query itself but less by the data sets.

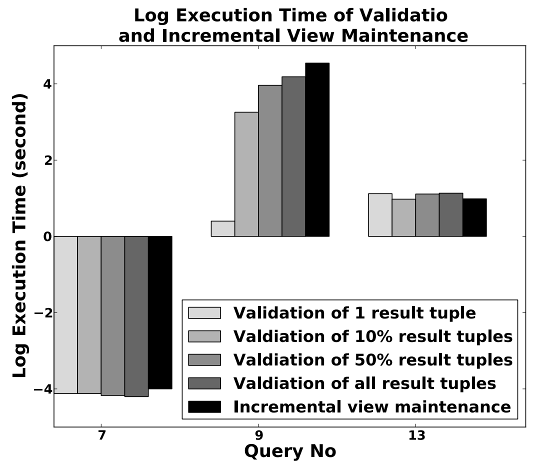
Experiments On Two-Block Queries

For each query with inline views, we follow the same steps for their experiment as we did for the experiment on queries without inline views. The results are reported in Figure 4.9a, Figure 4.9b, Figure 4.9c and Figure 4.9d for the four different data sets respectively.

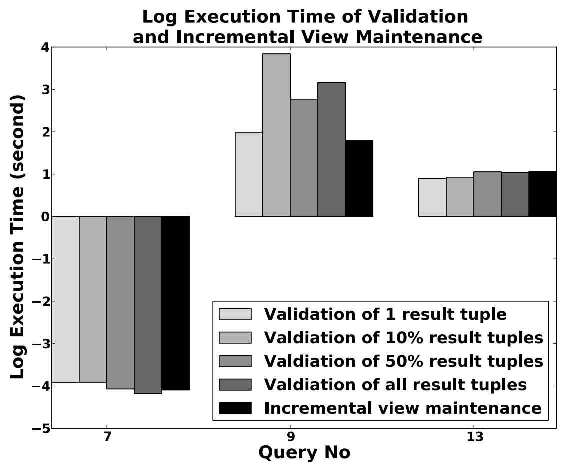
For Q_7, the time costs of the validation with pruning predicates are roughly the same



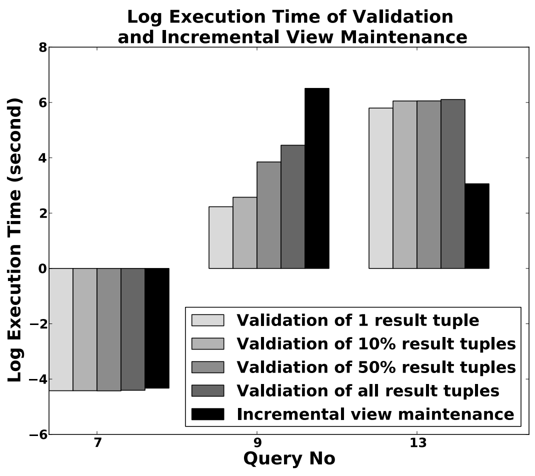
(a) Experiments On The Data Set #1



(b) Experiments On The Data Set #2



(c) Experiments On The Data Set #3



(d) Experiments On The Data Set #4

Figure 4.9: Experiments On Queries With Inline Views

as those of the incremental view maintenance. There are two causes for this: (1) Q_7 has only four result tuples, and thus the selected answers are almost the entire result set; (2) the particular delta tables don't provide much pruning.

For Q_9, on the data set #1, #2 and #4, the validation with pruning predicates is obviously more efficient than incremental view maintenance (note that the time cost is presented in natural logarithm) and has a clear pattern of growth in the time cost with the increased size of selected answers. However, on the data set #3, the anomaly in performance may be due to the particular data set instance.

For Q_{13}, the validation with pruning predicates is similar to the view maintenance due to three reasons. First, it involves only one source table. Second, it has a relatively small result set. Third, the outer block's GROUPBY attribute is the aggregate attribute produced by the inner query and both aggregations are *count*, and thus no non-trivial pruning predicate can be found for the inner block.

4.7 Conclusion

The validity of derived tuples or answers may change due to the source data set updates. View maintenance can track the validity of every answer in the result set by updating the complete result set in case of source data set updates. However, under an application scenario where only a small fraction of all the answers in the result set are concerned, the validity of this small fraction of answers can be checked by view maintenance enhanced with pruning predicates. Pruning predicates can prune irrelevant source tuples that can not possibly affect the selected answers; pruning predicates for the outermost query block (in a nested query) depend on the values of the selected answers and pruning predicates for each inner query block can be inferred from the pruning predicates for its parent query block.

If an answer is invalidated, the explanation of its invalidation consists of the tuples in the source data set updates and possibly tuples in the original source data set. Moreover, those tuples in the source data set updates can be either inserted tuples (positive tuples) or removed tuples (negative tuples). If we extend the provenance definition such that (1) both positive tuples and negative tuples can have provenance and (2) the provenance can consist of positive tuples and negative tuples, then the explanation is the provenance of the invalidated answer.

CHAPTER V

Provenance in Asynchronous Collaboration Of Text Documents

5.1 Introduction

Collaborative editing allows multiple users to edit the same text document. *Asynchronous* multiple-user document collaboration allows users working on separate copies of the same document and merges the diverged copies. An example is the revision control system, e.g., CVS (stand-alone), or Microsoft Word (with integrated version control).

In this chapter, we focus on asynchronous multiple-user collaboration of a *text* document without any predefined structures. In this asynchronous document collaboration application, the revision history of a text document can be represented as a version tree, e.g., the one shown in the left part of Figure 5.1.

Every node in the version tree represents a version of the document. The version tree of a document captures all the edits to the document. With the version tree, the document at any previous time point can be reconstructed. A typical revision control system allows the user to restore any previous version. (Some may support additional functionalities, such as search over the version tree for specified keywords.) However, what if the user wants to view the complete revision history related to a selected piece of text[1] instead of the whole document?

A document may have different parts modified at different times. If a user is interested

[1]In this chapter, when we refer to a piece of text, we mean a continuous range of text if not explicitly stating otherwise.

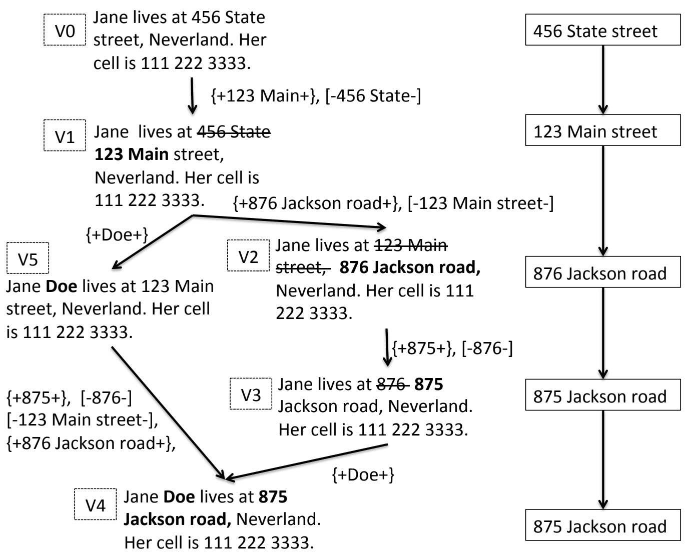

V0
Jane lives at 456 State street, Neverland. Her cell is 111 222 3333.
{+123 Main+}, [-456 State-]
V1
Jane lives at ~~456 State~~ **123 Main** street, Neverland. Her cell is 111 222 3333.
{+876 Jackson road+}, [-123 Main street-]
{+Doe+}
V5
Jane **Doe** lives at 123 Main street, Neverland. Her cell is 111 222 3333.
V2
Jane lives at ~~123 Main street,~~ **876 Jackson road,** Neverland. Her cell is 111 222 3333.
{+875+}, [-876-]
V3
Jane lives at ~~876~~ **875** Jackson road, Neverland. Her cell is 111 222 3333.
{+875+}, [-876-] [-123 Main street-], {+876 Jackson road+},
{+Doe+}
V4
Jane **Doe** lives at **875 Jackson road,** Neverland. Her cell is 111 222 3333.
456 State street
123 Main street
876 Jackson road
875 Jackson road
875 Jackson road

Figure 5.1: Version Tree And Text Piece Provenance

in the provenance of a particular part of the document, edits to the other parts of the document are irrelevant. In this case, the version tree of the document is not desirable as the provenance requested by the user, since it includes all edits, including the ones outside the part of interest. Moreover, when the document is quite large or has many versions, its version tree contains a great deal of information, likely to overwhelm the user, and potentially even strain system resources when being retrieved. It would be much preferred if we can define a notion of provenance for a selected piece of text that contains only relevant revision history.

Since the text document does not have a predefined data structure, any piece of text in the document can be viewed as a temporary data object. A piece of text may be semantically meaningful or arbitrarily chosen. It may have been specified/tagged by the user or inferred from the revision history. How exactly it is identified is beyond the scope of this chapter: we want to build our technique making no assumption about the beginning and end points of an arbitrary piece of text.

The desired relevant revision history of a selected piece of text can not be acquired by simply searching the version tree with keywords. First, the same keyword may appear in different parts in the document, while the selected piece of text actually has two inherent characteristics: the content and the position. Second, the insertion of the selected piece at a particular position does not necessarily serve as the very beginning of the revision history of the selected piece as shown by the example in Figure 5.1.

In Figure 5.1, if we trace back the selected text '875 Jackson road', we can find that it results from correcting '876 Jackson road', and '876 Jackson road' further results from replacing '123 Main street', and '123 Main street' in turn results from replacing '456 State street'. Then, should we include '123 Main street' and '456 State street' in the *complete* history of the selected text '875 Jackson road', even if they don't contain any part of the selected text? We would like to argue that we should for two reasons. First, the replaced texts, although do not contain the selected text, explain why the selected text is where it is since the selected text enters the document to replace them. Second, the replaced texts provide hints about the semantic meaning of the selected text, e.g., in Figure 5.1, all the text pieces shown in the right part are part of an address. Although the semantic meaning of text pieces is beyond the scope of this chapter, the replaced texts are worthwhile to be included in the revision history for their potential relationship with the selected text in terms of their semantical meanings.

The goal of this chapter is to define and retrieve provenance for a piece of text in a collaborative document that is edited asynchronously. The technical challenges of computing provenance of this selected piece of text include:

(i). Identify data objects contained in the selected text. Since we do not consider semantics of the text, data objects can be defined according to the revision history related criterion, e.g., a collection of continuous words that are always edited together according to the revision history so far. We call these identified data objects *revi-*

sion units. The identified revision units change dynamically when the version tree changes.

(ii). Retrieve the dependency relationship among the revision units in all the versions of the document. Given a specific version in the version tree and a selected piece of text in this version, its provenance consists of all the revision units contained in the selected text piece and all the revision units from the previous versions on which the revision units in the selected text piece depend.

The rest of this chapter is organized as follows. In Section 5.2, we review preliminaries necessary for the discussion of revision history and provenance in this chapter. In Section 5.3, we review related work. In Section 5.4, we show how to identify revision units in a piece of text according to the version tree, how to construct the dependency graph of revision units, and how to maintain the graph upon the version tree update. In Section 5.5, we define and retrieve provenance of revision units, and show how to construct the provenance of a selected piece of text from the provenance of related revision units. In Section 5.6, we experiment with the retrieval algorithm on Wikipedia page history data. In Section 5.7, we conclude our work on the provenance of selected text pieces in a collaborative document edited asynchronously by multiple users.

5.2 Preliminaries

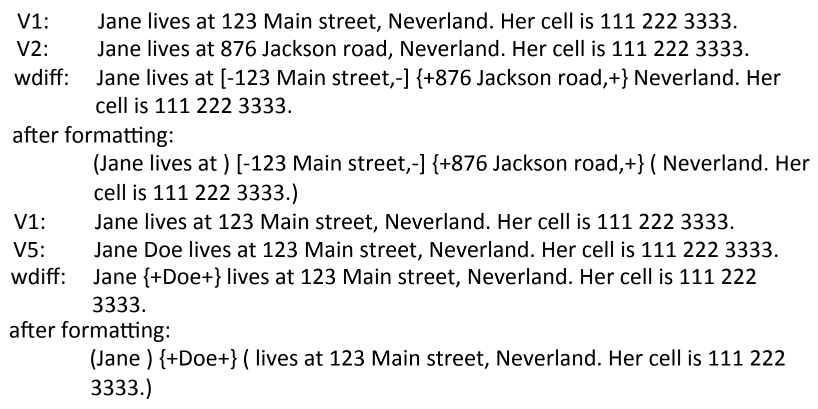
The revision history of a text document is captured as a version tree in general, which is in fact a directed acyclic graph, e.g., Figure 5.1. It becomes a linear sequence of versions if concurrent revisions are not allowed. We denote a version tree as $G = \{V, E\}$. $V = \{v_1, ..., v_n\}$ and is the set of all the versions in G. E is the set of all the edges in G. If there exists $e \in E$ and $e = (v_i, v_j)$, where $v_i, v_j \in V$, then v_j is derived from v_i, and v_j depends on v_i.

Given $(v_i, v_j) \in E$, v_j is derived from v_i through one or more revision operations, such as inserts, deletes, or through merging v_i and other versions. The difference between v_i and v_j is based on *longest common subsequence*. We make use of $wdiff$ package (GNU wdiff 0.6.5) written by Francois Pinard to calculate the difference between v_i and v_j.

$wdiff(v_i, v_j)$ compares two documents v_i and v_j, and outputs inserted text pieces and removed text pieces and inherited pieces with regard to v_i. A removed text piece is a text piece in v_i but not in v_j. An inserted piece is a text piece in v_j but not in v_i. An inherited text piece is in both v_i and v_j. Examples are shown in Figure 5.1, where "{+Doe+}" is an inserted piece, "[-123 Main street-]" is a removed piece, and "Jane lives at" is an inherited piece. The set of all removed text pieces is denoted as $v_i - v_j$, the set of all inserted text pieces is denoted as $v_j - v_i$, and the set of all inherited text pieces is denoted as $v_i \cap v_j$. Note that the text pieces are all of different lengths: this is typical.

The output of $wdiff$ is parsed and formatted into a sequence of text pieces (strings) as shown in Figure 5.2. In the rest of this chapter, we assume the output of $wdiff$ is always formatted that way. In this formatted output, the text document is divided into text pieces embraced with either '(' and ')', or '{+' and '+}', or '[-' and '-]'. The text pieces inside '(' and ')' are inherited from the previous version to the next version. The text pieces inside '{+' and '+}' are inserted. The text pieces inside '[-' and '-]' are removed. Moreover, two adjacent text pieces can not be embraced with the same type of braces, since they can be merged into one text piece.

$wdiff$ does not identify whether an inserted text piece and a removed piece are related, e.g., the former replacing the latter. Strictly speaking, unless explicitly specified, we can not say for sure an inserted text piece is the replacement of a removed text piece. However, inference of a possible replacement is useful. For example, when computing the revision history of a text piece, the indication of the possibility of this text piece as a replacement



```
 V1:       Jane lives at 123 Main street, Neverland. Her cell is 111 222 3333.
 V2:       Jane lives at 876 Jackson road, Neverland. Her cell is 111 222 3333.
wdiff:     Jane lives at [-123 Main street,-] {+876 Jackson road,+} Neverland. Her
           cell is 111 222 3333.
after formatting:
           (Jane lives at ) [-123 Main street,-] {+876 Jackson road,+} ( Neverland. Her
           cell is 111 222 3333.)
 V1:       Jane lives at 123 Main street, Neverland. Her cell is 111 222 3333.
 V5:       Jane Doe lives at 123 Main street, Neverland. Her cell is 111 222 3333.
wdiff:     Jane {+Doe+} lives at 123 Main street, Neverland. Her cell is 111 222
           3333.
after formatting:
           (Jane ) {+Doe+} ( lives at 123 Main street, Neverland. Her cell is 111 222
           3333.)
```

Figure 5.2: Examples Of $wdiff$ Output

to a removed text piece can help the user to explore the possible reasons of the presence of this text piece. In this chapter, the purpose of inferring the possible replacement is more about the exploration of possible explanations of the selected text piece, and less about the accuracy of the inference or how close the inference is to the fact.

In this chapter, we treat versions and text pieces as strings and assume some common string functions for concise notation.

- $s.indexOf(t)$ represents the starting position of t in s
- $s.length()$ represents the length of s
- $s.substring(pos, len)$ represents a substring of s starting at pos with the length len, which means that the substring ends just before $pos + len$

The position in a string is zero-based. Moreover, when we say two strings are identical or one string is equal to another, we mean their contents are the same.

5.3 Related Work

Provenance has been studied extensively for database applications and scientific workflow applications. The original meaning of provenance is history and origin. The actual meaning of provenance depends on the specific interpretations with regard to the specific applications. Provenance information in different types of applications are usually quite different.

With regard to scientific workflow management systems, the basic provenance of the output of a workflow run consists of all the intermediate results and the dependency among them [13]. With regard to database management systems, the basic provenance of derived data consists of source data in the database that are used by the queries expressed in a declarative data query language [12, 18, 10, 30] or consists of the operations expressed in a data modification language [3, 8].

When applying the concept of provenance to text document editing, the provenance is about the revision history of a particular version of a text document or a particular text piece in the version. For collaborative documents, such as Wikipedia pages, the revision history is usually structured as a version tree [34], where the nodes in the tree are versions of the document.

Revision histories are very useful since they can be used to infer some very useful information about the documents, such as trustworthiness [39], error patterns [27], linguistic information [32], trend of revision choices [36], etc. The inference calls for an analysis of the differences between versions of the document. To model those differences, the reasonable granularities include words [32], sentences [39, 36, 32], paragraphs [32] and lines for plain text documents, and method calls for code [27]. The granularity is often picked to suit the purpose of the application in question. Usually, it is chosen to be something of a domain-specific meaning with regard to the application, e.g., linguistic units (words, sentence, paragraphs) for plain text documents, and method calls for code.

In our work, our goal is to find the revision provenance of an arbitrary text piece. Thus, we would like to choose a granularity related to revisions such that it can be fine enough to separate adjacent texts affected by different revisions and coarse enough to merge adjacent texts affected by the same revision. In other words, the granularity should allow us put finer granules to where revisions involve small text pieces and put coarser granules to where revisions involve monolithic large text pieces. Moreover, a single revision on a particular version should have the potential to affect the granules formed in other versions in the revision history, because there is certain dependency among revisions.

5.4 Revision Units And Its Dependency Graph

Text documents don't have predefined structures. Any continuous substring of a text document can be considered as an integrated unit for revision purpose. In fact, different granularities have been adopted for the computation of file difference, e.g., at sentence level, at word level, at line level. Some choices are due to semantical structures, such as sentences or words; some aren't, such as lines. In this chapter, we propose an adjustable granularity which is determined by the revision history. We call it *revision units*.

Definition 5.1. [Revision Unit] Given a version tree (V, E) and $v \in V$, suppose r is a tuple of schema $(version, position, str)$, where $r.position$ is a position in the version $r.version$ and $r.str$ is a string starting at $r.position$, then $r = (v, p, s)$ is a revision unit if and only if it satisfies one of the following conditions:

(i). $\exists(v, v') \in E$ such that $s \in v - v'$ and s equals $v.substring(p, s.length())$

(ii). $\exists(v', v) \in E$ such that $s \in v - v'$ and s equals $v.substring(p, s.length())$

(iii). $\exists(v, v') \in E$ such that $s \in v \cap v'$ and s equals $v.substring(p, s.length())$

(iv). $\exists(v', v) \in E$ such that $s \in v \cap v'$ and s equals $v.substring(p, s.length())$

(v). s is an empty string

(vi). s is a piece of text selected by the user from v and s equals $v.substring(p, s.length())$

In Definition 5.1, the tuple (v, p, s) uniquely identifies a piece of text in the revision history. s is a string that is either removed or inserted or inherited. The first two conditions in Definition 5.1 specify the removed or inserted units. The third and fourth conditions in Definition 5.1 specify the inherited units. The reason that the first and the second conditions (the third and the fourth conditions) can not be merged into one condition is because some version may not have a parent version or have a child version.

Moreover, r can also be an empty revision unit when s is an empty string. This type of units are dummy units that are used when constructing the mapping between units, which will be shown later in this chapter. Also, if a user selects a text piece in some version, the selected piece becomes a unit.

From a version tree, we can uniquely identify a dependency graph of revision units, as shown in Figure 5.3. In Figure 5.3, every node (in the box) is a revision unit. Inside each node, we indicate the version and the text of the revision unit. In Figure 5.3, every edge indicates the transition of a revision unit in one version to another revision unit in the next version. The revision unit at the end of a directed edge depends on the revision unit at the beginning of the directed edge. There are four types of dependency as indicated in Figure 5.3, which will be explained in details later in this section. Moreover, the dependency graph in Figure 5.3 has several disconnected components. For example, The revision units containing the text “Jane” form a disconnected component. The edges in that component are all marked as “inherited”, which indicates that the text piece “Jane” has never been modified throughout the version tree. As another example, the address part constitutes another disconnected component. If we are looking for the provenance of a particular revision unit, it must be in the same component as the revision unit.

In the rest of this section, we are going to show how to construct the dependency graph of revision units from a version tree. In the next section, we are going to use this depen-

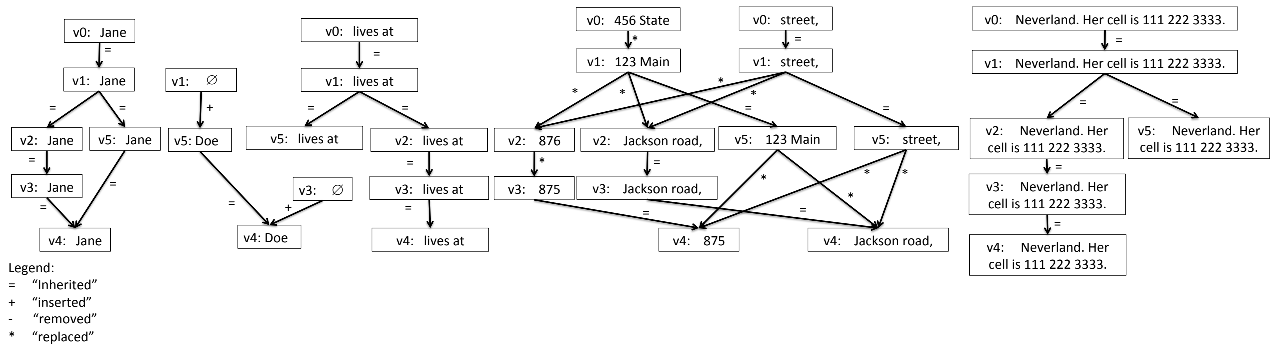


Figure 5.3: Dependency Graph Of Revision Units Derived From The Version Tree In Figure 5.1

dency graph to retrieve provenance of arbitrary revision units and arbitrary text pieces.

We first give an overall description of the procedure that constructs the dependency graph and then discuss in details each step in the procedure. Suppose we have a version tree $G = (V, E)$, we denote the dependency graph derived from it as $D = (U, M)$, where U is the set of all revision units, and M is the set of edges among revision units. We follow the steps below to construct D.

Sorting We sort the nodes in V into a topological ordering of G, and then sort the edges in E such that the starting nodes of the sorted edges follow the topological ordering.

Incremental Construction Initialize $D = (U, M)$ to be an empty graph and grow it every time an edge in E is processed.

Consecutive Versions For each edge $(v, v') \in E$, we call Algorithm 6 to convert v and v' into two sequences of revision units respectively, denoted as $U(v)$ and $U(v')$; and find the dependency relationship between the revision units in $U(v)$ and the revision units in $U(v')$, denoted as $f : U(v) \to U(v')$.

Updating Graph $U(v)$, $U(v')$ and f are used to update the existing U and M respectively.

(i). If the existing U already contains a sequence of revision units of v, say $U'(v)$, we need to merge $U(v)$ and $U'(v)$ by calling Algorithm 7 and Algorithm 8 since every version is only allowed to be converted to one sequence of revision units. Otherwise, $U(v)$ is simply added to U.

(ii). $U(v')$ is processed in the same way as $U(v)$.

(iii). In either of the above two steps, if the merging takes place, $f : U(v) \rightarrow U(v')$ will be adjusted according by Algorithm 7 and 8, and then added to M. Otherwise, it is directly added to M.

We are going to discuss the incremental construction of the dependency graph first since it is the major step in the above procedure, and then discuss the sorting when summarizing with Algorithm 9.

5.4.1 Identifying Revision Units And Their Relationship From Two Consecutive Versions

The incremental construction of the dependency graph consists of two steps: processing two adjacent versions and updating the dependency graph from the output of the processing.

We first look into the step that processes two consecutive versions. Given any two consecutive versions of a text document, according to the revisions between them we can convert these two versions to two sequences of revision units respectively. Moreover, there is a one-to-one relationship between these two sequences of revision units as shown in Figure 5.4. In Figure 5.4, “Doe” is inserted and thus the revision unit “(Doe)” in v_5 is paired with an empty revision unit in v_1. In general, given two consecutive versions or an edge in the version tree, we (1) convert either version into a sequence of revision units and (2) find the one-to-one dependency relationship of revision units in these two sequences as shown in Algorithm 6.

The one-to-one relationship of revision units in two consecutive versions, computed in

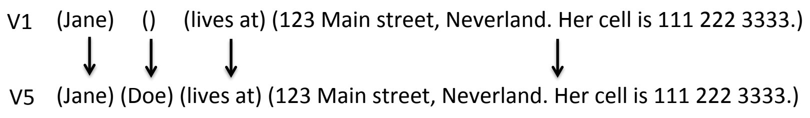

V1 (Jane) () (lives at) (123 Main street, Neverland. Her cell is 111 222 3333.)
V5 (Jane) (Doe) (lives at) (123 Main street, Neverland. Her cell is 111 222 3333.)

Figure 5.4: Revision Units In Consecutive Versions And Their Correspondence

Algorithm 6, has four different types.

(i). If an empty revision unit in v_i corresponds to a non-empty revision unit in v_j, then the latter is inserted during the transition from the version v_i to v_j.

(ii). If a non-empty revision unit in v_i corresponds to an empty revision unit in v_j, the former unit is removed during the transition from v_i to v_j.

(iii). If a non-empty revision unit in v_i corresponds to a non-empty revision unit in v_j, and those two contain different texts, then the former is replaced by the latter during the transition from v_i to v_j.

(iv). If a non-empty revision unit in v_i corresponds to a non-empty revision unit in v_j, and those two contain identical texts, then the former is kept during the transition from v_i to v_j and becomes the latter.

The insertion, deletion and inheritance are easy to infer from the output of $wdiff$. For the replacement, we use a simple heuristics to infer it. When a removed text piece is next to (i.e., immediately before or after) an inserted text piece, we say that the latter replaces the former. This simple heuristics will need to false replacement relationship between tow revision units. We prefer it to missing actual replacement relationship. That is because the user who later examines the provenance can determine for himself whether he would like to discard the replacement relationship, but can not possibly find out the actual replacement relationship if it is not contained in the provenance.

In a version tree, most versions have more than one edge connected to it, including both incoming edges and outgoing edges, e.g., (v_0, v_1) and (v_1, v_5) or (v_5, v_4) and (v_3, v_4) in

Figure 5.1. Suppose we called Algorithm 6 on (v_0, v_1) first and produced $U(v_0)$, $U(v_1)$ and $f : U(v_0) \rightarrow U(v_1)$. We then added $U(v_0)$ and $U(v_1)$ to U, and added $f : U(v_0) \rightarrow U(v_1)$ to M. After that, we moved on to (v_1, v_5) and called Algorithm 6 on it. Thus, we produced a different $U(v_1)$ and find that the existing U already contains a sequence of revision units for v_1. Then, we need to merge that existing sequence and the new $U(v_1)$ into one.

5.4.2 Merging Two Sequences Of Revision Units Of The Same Version

In Figure 5.5, we illustrate how to merge two different sequences of revision units of the same version. In Figure 5.5, the first two sequences are $U(v_0)$ and $U(v_1)$ produced by calling Algorithm 6 on (v_0, v_1). Then, the second two sequences are $U'(v_1)$ and $U(v_5)$ produced by calling Algorithm 6 on (v_1, v_5), where $U'(v_1)$ is a sequence of revision units for v_1 but different from $U(v_1)$.

In order to merge $U(v_1)$ and $U'(v_1)$, we first align them, which is feasible since the underlying documents are the same, i.e., v_1. Then, if the starting position of a unit in one sequence is inside some revision unit in the other sequence, we split the latter unit, as shown with dashed arrows in Figure 5.5. After we split the units in $U(v_1)$ and $U(v_1')$, these two sequences will be consistent and we get a single sequence of revision units for v_1 shown as $U_s(v_1)$ below the block arrow in Figure 5.5.

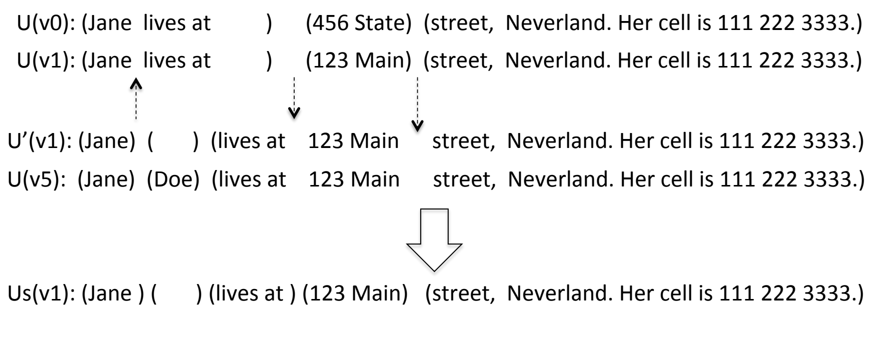

U(v0): (Jane lives at) (456 State) (street, Neverland. Her cell is 111 222 3333.)
U(v1): (Jane lives at) (123 Main) (street, Neverland. Her cell is 111 222 3333.)
U'(v1): (Jane) ( ) (lives at 123 Main street, Neverland. Her cell is 111 222 3333.)
U(v5): (Jane) (Doe) (lives at 123 Main street, Neverland. Her cell is 111 222 3333.)
Us(v1): (Jane) ( ) (lives at) (123 Main) (street, Neverland. Her cell is 111 222 3333.)

Figure 5.5: Example Of Splitting Units

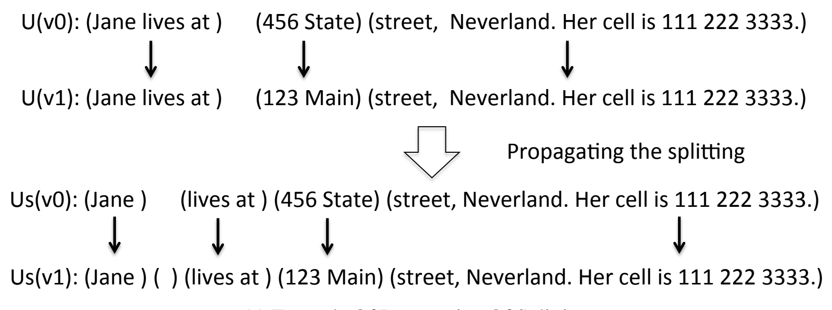
After we get $U_s(v_1)$ after merging $U(v_1)$ and $U'(v_1)$, we will use $U_s(v_1)$ to replace $U(v_1)$, which is currently in U. This replacement will lead to another problem: what should we do about the dependency relationship between $U(v_1)$ and $U(v_0)$ that is currently in M, since $U(v_1)$ is replaced by $U_s(v_1)$. Certainly, any dependency involve the units in $U(v_1)$ should be removed from M when $U(v_1)$ is removed from U. Then, how should we construct the new dependency between the units in $U_s(v_1)$, the replacement to $U(v_1)$, and the units in $U(v_0)$?

We have two options shown in Figure 5.6a and in Figure 5.6b: propagating the splittings in the units of v_1 or not. These two options will result in different updated U and M.

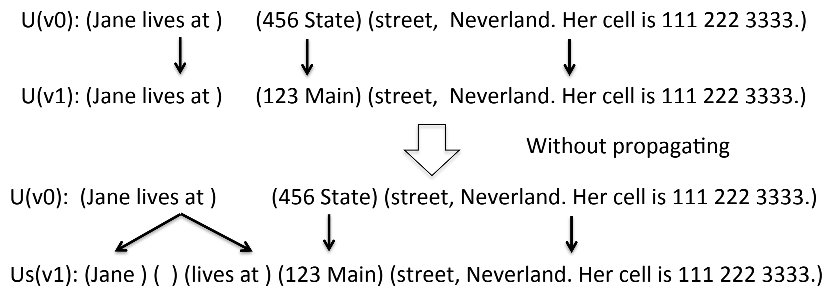
In Figure 5.6a, we show how to propagate the splitting in a revision unit. When the first unit in $U(v_1)$ is split between "Jane" and "Doe", this unit's corresponding unit in v_0 is split accordingly at the corresponding position as indicated with the dashed arrows. If we propagate the splittings, the dependency relationship between the units of v_0 and v_1 keeps being one-to-one as indicated with the solid arrows. Moreover, since the revision units of v_0 get split too, the splitting can propagate further from the splittings in v_0.

In Figure 5.6b, we show an alternative choice to propagating the splitting of units. If we don't propagate the splittings in the revision units of v_1 to the units of v_0, the dependency relationship between the units of v_0 and v_1 is adjusted accordingly and is no longer one-to-one as indicated by the two sequences below the block arrow in Figure 5.6b.

In general, we prefer propagating the splitting as far as possible, since a one-to-one relationship is a more precise and concise description of the dependency among revision units. However, propagating is not always possible. In fact, it is only feasible when the dependency relationship is of the type "inherited". In Figure 5.6a, the unit "Jane lives at" is inherited from v_0 to v_1, and therefore the splitting position in the unit of v_1 has an exact match in the corresponding unit of v_0, which enables us to split the latter as well. Otherwise, when the dependency relationship being "replaced", or "inserted" or "removed", the splitting can not be propagated. Depending on the dependency relationship, the splitting in a single revision unit v can potentially lead to a splitting in any revision unit that is reachable from v or can reach v.


U(v0): (Jane lives at) (456 State) (street, Neverland. Her cell is 111 222 3333.)
U(v1): (Jane lives at) (123 Main) (street, Neverland. Her cell is 111 222 3333.)
Propagating the splitting
Us(v0): (Jane) (lives at) (456 State) (street, Neverland. Her cell is 111 222 3333.)
Us(v1): (Jane) ( ) (lives at) (123 Main) (street, Neverland. Her cell is 111 222 3333.)

(a) Example Of Propagation Of Splitting


U(v0): (Jane lives at) (456 State) (street, Neverland. Her cell is 111 222 3333.)
U(v1): (Jane lives at) (123 Main) (street, Neverland. Her cell is 111 222 3333.)
Without propagating
U(v0): (Jane lives at) (456 State) (street, Neverland. Her cell is 111 222 3333.)
Us(v1): (Jane) ( ) (lives at) (123 Main) (street, Neverland. Her cell is 111 222 3333.)

(b) Example Of No Propagation Of Splitting

Figure 5.6: Cascading Effect Of Splitting

Given two sequences of revision units of the same version, say $U(v)$ and $U'(v)$, Algorithm 7 finds (1) a new sequence $U_s(v)$ that is the result of merging $U(v)$ and $U'(v)$, and (2) the mapping from each unit in $U(v)$ or $U'(v)$ to one or more units in $U_s(v)$. Then, according to the mapping found, Algorithm 8 replaces a revision unit in $U(v)$ or $U'(v)$ with the units in $U_s(v)$ it is mapped to.

In Algorithm 7, the output g (g') captures the correspondence mapping between a unit in $U(v)$ ($U'(v)$) and one or more units in $U_s(v)$. Given a revision unit r in $U(v)$ ($U'(v)$), suppose it is mapped to $r_1, ..., r_m$ in $U_s(v)$ according to g (g'), then, we can call Algorithm 8 to replace r with $r_1, ..., r_m$.

In Algorithm 8, a revision unit r is replaced by $r_1, ..., r_m$. The dependency between r and another existing unit r' will be modified to dependency between r_i ($i = 1, ..., m$) and

r' or the split units out of r'. If the latter case is true, Algorithm 8 is called recursively to replace r' with those split units of r'.

5.4.3 Construction Of The Dependency Graph Of Revision Units From A Version Tree

As a summary of the incremental construction of the dependency graph of revision units from a version tree, we show Algorithm 9 below. It makes use of Algorithm 6, Algorithm 7 and Algorithm 8. It is in fact the same as the textual description of constructing the dependency graph at the beginning of this section, but put in the algorithm form.

In Algorithm 9, the first instruction is to find a topological ordering of the versions in the version tree. A version tree is a directed acyclic graph (DAG), and therefore it defines a partial order of all the versions. Suppose one of the topological ordering of this partial order is $v_1, ..., v_n$, then v_i only depends on $v_1, ..., v_{i-1}$. The topological ordering can be found through breadth-first-search over the version tree. With the topological ordering of nodes, we can sort the edges as $e_1, ..., e_m$ such that if e_i is before e_j, then the starting node of e_i is either the same as the starting node of e_j or is before it in the topological ordering of all the nodes. This sorted sequence of edges can be easily found as: for each node in v_1 through v_n, list all the edges connected to it.

5.4.4 Maintenance Of Dependency Graph Upon Version Tree Update

The dependency graph of revision units computed from a version tree needs to be maintained when the version tree is updated, e.g., new versions are added to the version tree. Suppose $G = (V, E)$ is the existing version tree, and a new version v' is added to V and (v, v') is added to E. Further suppose $D = (U, M)$ is the existing graph of revision units derived from G. Then, the following steps are taken to update D.

(i). Call Algorithm 6 on (v, v') to get $U(v)$, $U(v')$ and $f : U(v) \rightarrow U(v')$.

(ii). Call Algorithm 7 on $U(v)$ and the existing sequence of v, say $U'(v)$, to get $U_s(v)$,

$g : U(v) \rightarrow U_s(v)$ and $g : U'(v) \rightarrow U_s(v)$

(iii). Call Algorithm 8 to update U by replacing $U'(v)$ with $U_s(v)$, and adjust $U(v')$ and f accordingly before adding them to U and M respectively

5.5 Provenance Of Revision Units And Arbitrary Text Pieces

In the following, we first define and retrieve provenance for a revision unit. Then, we define the provenance for a selected text piece and show how to construct it by combining the provenance of revision units related to this text piece.

5.5.1 Provenance Of Revision Units

Definition 5.2. [Provenance Of Revision Units] Given a version tree $G = (V, E)$, the dependency graph of revision units $D = (U, M)$ derived from G, a revision unit $r \in U$, the provenance of r includes (1) any r' such that there exists at least one path from r' to r in D and (2) any edge in M such that there exists at least one path from r' to r with that edge in it.

According to Definition 5.2, the retrieval of the provenance of a given revision unit can be thought of finding a sub-DAG in D. Suppose $D = (U, M)$ is the dependency graph of revision units, given a revision unit r, its provenance is represented as $D(r) = (U(r), M(r))$, where $D(r)$ is a sub-DAG of D and $U(r) \subseteq U$ and $M(r) \subseteq M$. One approach to find $D(r)$ is a two-step procedure: first, find all the "reachable" units which constitute $U(r)$; second, for every edge in M, if both of its end units are in $U(r)$, it is in $M(r)$.

For the first step, all the "reachable" units can be found by searching depth-first along the *reversed* directions of the edges in M until reaching revision units that do not have incoming edges in M. Note that we need to use the reversed directions of the edges since we are actually searching from units in newer versions towards units in older versions.

In Algorithm 10, we show how to retrieve the provenance of a given revision unit.

5.5.2 Provenance Of Selected Text Pieces

If a user selects a continuous piece of text, it may be contained within one revision unit or contain one or more consecutive revision units. For each revision unit fully covered by the selected piece, its revision history is then part of the revision history of the selected piece. For each revision unit partially covered by the selected piece, it should be further refined into two or three revision units such that one of the refined revision units is fully contained in the selected piece. Thus, the provenance of the selected continuous text piece can be built upon the provenance of those revision units fully contained in the text piece.

Given t as a selected text piece in v, if t covers a sequence of consecutive revision units, then the first and last revision units may only be partially contained in t. If so, the partially covered unit is refined by being split into two units such that one of them is fully contained in t. Similarly, if t is contained in a single revision unit, there may exist margins on both ends of t inside the revision unit. Then, the revision unit needs to be split into three units and the middle one is fully covered by t. This splitting of a revision unit may propagate throughout the entire dependency graph of revision units. The splitting and the possible propagation of splitting are handled by Algorithm 8 given the original unit and its replacement units.

Suppose there is a revision unit (v, p, s) and a text piece t completely contained in s. Then, the replacement units for (v, p, s) are

- $(v, p, s.substring(0, s.indexOf(t)))$
- $(v, p + s.indexOf(t), t)$
- $(v, p + s.indexOf(t) + t.length(), s.substring(p + s.indexOf(t) + t.length(), s.length() - t.length() - s.indexOf(t)))$

Suppose there is a revision unit (v, p, s) and a text piece t. If t is a substring of s and matches s from the beginning to somewhere in the middle, then the replacement units for (v, p, s) are

- (v, p, t)
- $(v, p + t.length(),\ s.substring(s.length() - t.length()))$

If t is a substring of s and matches s from somewhere in the middle to the end, then the replacement units for (v, p, s) are

- $(v, p, s.substring(0, s.indexOf(t)))$
- $(v, p + s.indexOf(t), t)$

After the proper splitting of revision units using Algorithm 8, the selected continuous piece of text can exactly cover a sequence of consecutive revision units. Then, we can find the provenance of each contained revision unit as shown in Algorithm 10. Finally, we can assemble the provenance of the text piece from the provenance of those revision units. It is straightforward as set unions. Since the provenance of a revision unit is a set of versions and a set of edges, the union of the provenance of multiple revision units consists of the union of multiple sets of versions and the union of multiple sets of edges.

If the user selects a non-continuous text piece, which consists of several continuous text pieces, then the provenance of the non-continuous text piece is the union of the provenance of each continuous text piece contained in it.

5.6 Experiments

The main benefit obtained from determining provenance with revision units for specified pieces of text is simplicity – the user sees a small, focused result without distraction from much irrelevant revision information. While the size of the resulting provenance is not the whole story, it is a large part of it, and is much easier to measure than user satisfaction. As such, we conducted experiments to measure the size of provenance for selected text snippets and compared this to the size of provenance for the document as a whole.

5.6.1 Experimental Datasets

We use Wikipedia pages as our experimental data. For each page, Wikipedia keeps all its historical versions as its page history. The page history of a Wikipedia page can be exported, and a maximum number of 1000 versions can be included in the history.

We first picked a set of Wikipedia pages whose titles fall into a randomly selected range; then discarded pages containing non-UTF8 characters, e.g., Asian language characters, and pages containing delimiters reserved for the $wdiff$ output, i.e., "(",")","{+","+}", "[-","-]"; finally we got 198 pages.

For each of the 198 pages, we exported its page history as a linear sequence of historical versions, which serves as the page's version tree. Then, we called Algorithm 9 to construct the dependency graph of revision units from the page's version tree. After that, we chose a random revision unit whose length is not zero form the latest version of the page. Finally, we retrieved the provenance of this revision unit with Algorithm 10.

Our code is in Java, and runs on an system that uses Intel Core i7 CPU with 1GB memory and runs Ubuntu Release 11.10 and Java version 1.6.0.

5.6.2 Experimental Results

The provenance size of a selected revision unit is potentially related to (1) the size of the selected unit, (2) the size of the version containing the unit, (3) the total size of all the versions, (4) the number of revision units in the version containing the selected unit and (5) the total number of all revision units. Our goal is to demonstrate these relationships through experiments, and confirm the sensibility of choosing revision units as a proper granularity for revision provenance. That means, the representation of provenance using revision units is more precise, and is neither too coarse as including irrelevant information nor too fine as having many fractional pieces of information. This advantage of using revision units will be reflected in effectively cutting down the size of revision information considered to be relevant to the selected revision unit, as compared to the entire version tree.

Suppose we have a version tree $G = (V, E)$ and a corresponding dependency graph of revision units $D = (U, M)$. Given a revision unit r of the schema $(version, position, str)$[2], suppose its provenance is $D(r) = (U(r), M(r))$, we have the following variables to observe

(i). the size[3] of the revision unit r, i.e., $r.str.length()$

(ii). the size of the version containing r, i.e., $r.version.length()$

(iii). the size of the entire dependency graph, i.e., $\sum_{u \in U} u.str.length()$

(iv). the size of the provenance of r, i.e., $\sum_{u \in U(r)} u.str.length()$

(v). the number of units in the version containing r, i.e., $|U(r.version)|$

(vi). the number of units in the provenance of r, i.e., $|U(r)|$

(vii). the total number of units in the entire dependency graph, i.e., $|U|$

We are going to make plots about the ratios of pairs of the above variables:

(i). $r.str.length()/r.version.length()$ vs. $\sum_{u \in U(r)} u.str.length()/ \sum_{u \in U} u.str.length()$

(ii). $|U(r.version)|/1$ vs. $|U|/|U(r)|$

In Figure 5.7, x-axis is the natural logarithm of $r.str.lenght()/r.version.length()$. We took the natural logarithm of the ratio to have more evenly distributed dots since the ratio varies greatly and is not evenly distributed over the range. In Figure 5.7, y-axis is the natural logarithm of $\sum_{u \in U(r)} u.str.length()/ \sum_{u \in U} u.str.length()$. Most of the dots in Figure 5.7

[2]See Definition 5.1 for the meaning of this revision unit representation and see Section 5.2 for the meaning of the assumed string functions.

[3]In this chapter, we only count the non-space character.

settle around the line $x = y$. That means these two ratios are pretty close. It reflects the fact that relevant revision provenance is linearly proportional to the revision unit in size. In other words, the provenance's size grows linearly with the selected text piece's size and if the selected text piece is the entire document, the provenance will become the entire dependency graph. It indicates that the choice of revision units as granularity is sensible in terms of effectively reducing the size of revision information considered to be relevant to the selected revision unit.

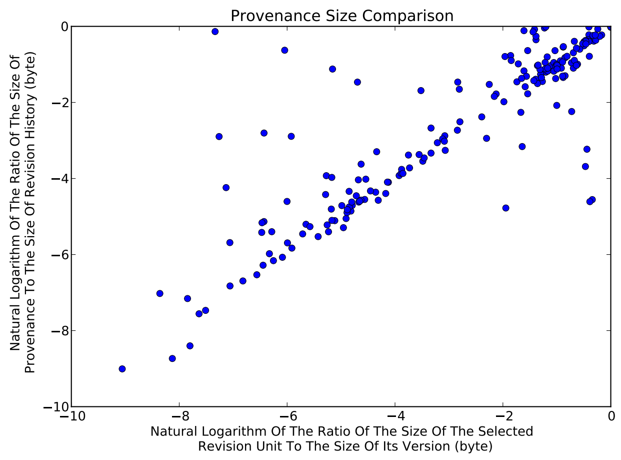


Figure 5.7: Provenance Size In Terms Of Bytes

As a comparison, in Figure 5.8, we set the x-axis as $|U(r.version)|$ and the y-axis as $|U|/|U(r)|$. They are also meant to measure the proportion of sizes as the x-axis and y-axis in Figure 5.7 do, but use a different measuring unit for size. In Figure 5.7, the measuring unit is byte, and in Figure 5.8 the measuring unit is the revision unit. Unlike Figure 5.7, Figure 5.8 does not show a clear pattern. This is because the size of the revision unit may vary greatly and the number of revision units as a measurement for size is not proper. However, from another point of view, it indicates that the length of the revision unit adjusts to the revisions.

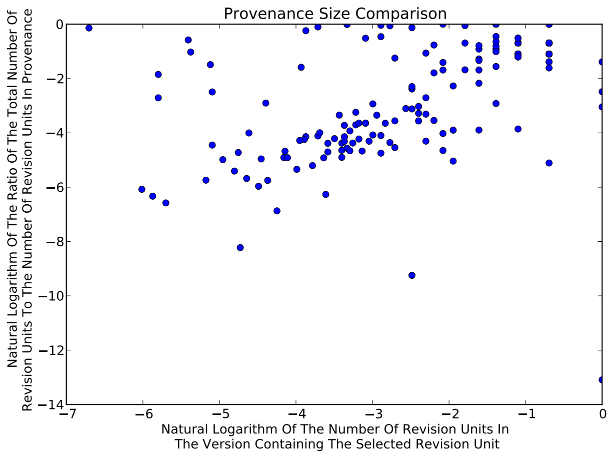


Figure 5.8: Provenance Size In Terms Of Revision Units

5.7 Conclusion

In this chapter, we introduced the provenance of a piece of text inside a collaborative document edited asynchronously by multiple users. Due to the lack of predefined structures in text documents, users can choose a piece of text ranging from a single word to the entire document. The provenance of the selected text piece should contain only the revisions relevant to the selected piece. We based the provenance for text pieces on revisions units. The choice of revision units as the granularity to represent revision provenance avoids being too fine or too coarse in the representation, since revision units adjust their ranges according to the revisions in the version tree of the document. A revision unit may depend on other revision units and the dependency is introduced by the revisions logged in the version tree. From the version tree, we can built a dependency graph of revision units. This dependency graph is then used to retrieve the provenance of any given revision unit. The provenance of revision units can further be used to construct the provenance of a selected text piece, which may contain (partially) one or more revision units, or may be contained in a single revision unit.

Algorithm 6: Construction Of Revision Units Out Of Two Consecutive Versions

```
Input: $e = (v_i, v_j)$
Output: $U(v_i), U(v_j), f : U(v_i) \to U(v_j)$
begin
    Call $wdiff$ on $v_i$ and $v_j$ and put the output in $v$
    Let $len$ be the length of $v$
    Let $units$ be an empty sequence of tuples $(v, p, s)$
    for $k = 0$ to $k < len$ do
        if $v[k] =' ('$ then
            Find the string $s$ inclusively between $v[k]$ and the first ')' after the position $k$
        end
        if $v[k, k+1] =' \{+'$ then
            Find the string $s$ inclusively between $v[k]$ and the first '+}' after the position $k$
        end
        if $v[k, k+1] =' [-'$ then
            Find the string $s$ inclusively between $v[k]$ and the first '-]' after the position $k$
        end
        Put $(v, k, s)$ into $units$
        Increase $k$ by the length of $s$
    end
    $pos_i \leftarrow 0$, $pos_j \leftarrow 0$
    for the head unit $r$ in $units$ do
        Remove $r$ from $units$
        Let $r'$ be the new head unit or be $null$ if $units$ is empty
        Let $str$ be $r.str$ stripped of the embracing delimiters
        Let $str'$ be $r'.str$ stripped of the embracing delimiters or $null$ if $r'$ is $null$
        if r.str is embraced with '(' and ')' then
            $r_i \leftarrow (v_i, pos_i, str)$, $r_j \leftarrow (v_j, pos_j, str)$
            $pos_i \leftarrow pos_i + str.length()$,$pos_j \leftarrow pos_j + str.length()$
        end
        if r.str is embraced with '[-' and '-]' and (r' is null or r'.str is braced with '(' and ')') then
            $r_i \leftarrow (v_i, pos_i, str)$, $r_j \leftarrow (v_j, pos_j, \emptyset)$
            $pos_i \leftarrow pos_i + str.length()$
        end
        if r.str is braced with '{+' and '+}' and (r' is null or r'.str is braced with '(' and ')') then
            $r_i \leftarrow (v_i, pos_i, \emptyset)$,$r_j \leftarrow (v_j, pos_j, str)$
            $pos_j \leftarrow pos_j + str.length()$
        end
        if r.str is braced with '{+' and '+}' and r'.str is braced with '[-' and '-]' then
            $r_i \leftarrow (v_i, pos_i, str')$, $r_j \leftarrow (v_j, pos_j, str)$
            $pos_i \leftarrow pos_i + str'.length()$,$pos_j \leftarrow pos_j + str.length()$
            Remove $r'$ from $units$
        end
        if r.str is braced with '[-' and '-]' and r'.str is braced with '{+' and '+}' then
            $r_i \leftarrow (v_i, pos_i, str)$, $r_j \leftarrow (v_j, pos_j, str')$
            $pos_i \leftarrow pos_i + str.length()$,$pos_j \leftarrow pos_j + str'.length()$
            Remove $r'$ from $units$
        end
        Put $r_i$ into $U(v_i)$, put $r_j$ into $U(v_j)$, put the pair $(r_i, r_j)$ into $f$
    end
end
```

Algorithm 7: Find Splittings To Merge Two Sequences Of Revision Units Of The Same Version

```
Input: $U(v)$, $U'(v)$
Output: $U_s(v), g : U(v) \rightarrow U_s(v), g' : U'(v) \rightarrow U_s(v)$
begin
    Suppose $U$ ($U'$) is an array of revision units that are sorted by their *position* values
    Initialize $U_s(v)$ to be an empty array
    $posInU \leftarrow 0$
    $posInU' \leftarrow 0$
    $currStart \leftarrow 0$
    while $posInU$ is less than the length of $U$ and $posInU'$ is less than the length of $U'$ do
        $nextStart1 \leftarrow U(v)[posInU].position + U(v)[posInU].str.length()$
        $nextStart2 \leftarrow U'(v)[posInU].position + U'(v)[posInU'].str.length()$
        if $nextStart1 < nextStart2$ then
            $s \leftarrow v.substring(currStart, nextStart1 - currStart)$
            $r_s \leftarrow (v, currStart, s)$
            Add $r_s$ to $U_s(v)$
            Add $(U(v)[posInU], r_s)$ to $g$
            Add $(U'(v)[posInU'], r_s)$ to $g'$
            $currStart \leftarrow nextStart1$
            $posInU + +$
        end
        if $nextStart1 > nextStart2$ then
            $s \leftarrow v.substring(currStart, nextStart2 - currStart)$
            $r_s \leftarrow (v, currStart, s)$
            Add $r_s$ to $U_s(v)$
            Add $(U(v)[posInU], r_s)$ to $g$
            Add $(U'(v)[posInU'], r_s)$ to $g'$
            $currStart \leftarrow nextStart2$
            $posInU' + +$
        end
        if $nextStart1 == nextStart2$ then
            $s \leftarrow v.substring(currStart, nextStart1 - currStart)$
            $r_s \leftarrow (v, currStart, s)$
            Add $r_s$ to $U_s(v)$
            Add $(U(v)[posInU], r_s)$ to $g$
            Add $(U'(v)[posInU'], r_s)$ to $g'$
            $currStart \leftarrow nextStart1$
            $posInU' + +$
            $posInU + +$
        end
    end
end
```

Algorithm 8: Splitting Revision Units

```
Input: $G = (V, E)$: a version tree
Input: $D = (U, M)$: a DAG of revision units computed from $G$, where $U$ is the set of all revision
        units, i.e., $\cup_{v \in V} U(v)$, and $M$ is the set of edges describing the dependency relationship between
        revision units
Input: $r \in U$: a revision unit
Input: $r_1, ...r_m$: consecutive units that are to replace $r$
Output: updated $D = (U, M)$
begin
    Remove $r$ from $U$
    Add $r_1, ..., r_m$ to $U$
    for every $(r, r')$ in $M$ do
        Remove $(r, r')$ from $M$
        if $r'$ is inherited from $r$ then
            for $i = 1$ to $m$ do
                $r'_i \leftarrow (r'.version, r'.position - r.position + r_i.position, r_i.str)$
                Add $(r_i, r'_i)$ to $M$
            end
            Call this algorithm recursively on $G, D, r', r'_1, ..., r'_m$
        else
            for $i = 1$ to $i = m$ do
                Add $(r_i, r')$ to $M$
            end
        end
    end
    for every $(r', r)$ in $M$ do
        Remove $(r', r)$ from $M$
        if $r$ is inherited from $r'$ then
            for $i = 1$ to $m$ do
                Let $r'_i$ be $(r'.version, r'.position - r.position + r_i.position, r_i.str)$
                Add $(r'_i, r_i)$ to $M$
            end
            Call this algorithm recursively on $G, D, r', r'_1, ..., r'_m$
        else
            for $i = 1$ to $i = m$ do
                Add $(r', r_i)$ to $M$
            end
        end
    end
end
```

Algorithm 9: Build A Dependence Graph Of Revision Units From A Version Tree

```
Input: $G = (V, E)$: a version tree
Output: $D = (U, M)$: a dependency DAG of revision units computed from $G$, where $U$ is the set of all
        revision units, i.e., $\cup_{v \in V} U(v)$, and $M$ is the set of edges describing the dependency
        relationship between revision units
1  begin
2      $U \leftarrow \emptyset$
3      $M \leftarrow \emptyset$
4      Suppose $v_1, ..., v_n$ is a topological ordering of the partial order defined by $G$
5      for $i = 1$ to $i = n$ do
6          for every $(v_i, v_j) \in E$ do
7              Call Algorithm 6 on $(v_i, v_j)$ to get $U(v_i)$ and $U(v_j)$ and $f : U(v_i) \leftarrow U(v_j)$
8              $U \leftarrow U \cup U(v_i)$
9              $U \leftarrow U \cup U(v_j)$
10             $M \leftarrow M \cup f$
11             if exists $U'(v_i)$ in $U$ already then
12                 Call Algorithm 7 on $U(v_i)$ and $U'(v_i)$ to get $U_s(v_i)$ and $g : U(v_i) \rightarrow U_s(v_i)$ and
                   $g' : U'(v_i) \rightarrow U_s(v_i)$
13                 for every $r$ in $U(v_i)$ or $U'(v_i)$ do
14                     Suppose $(r, r_1), ..., (r, r_m)$ are all the edges in $g$ or $g'$ that start with $r$
15                     Call Algorithm 8 on $G$, $D$, $r$, $r_1$,...,$r_m$
16                 end
17             end
18             if exists $U'(v_j)$ in $U$ already then
19                 Call Algorithm 7 on $U(v_j)$ and $U'(v_j)$ to get $U_s(v_j)$ and $g : U(v_j) \rightarrow U_s(v_j)$ and
                   $g' : U'(v_j) \rightarrow U_s(v_j)$
20                 for every $r$ in $U(v_j)$ or $U'(v_j)$ do
21                     Suppose $(r, r_1), ..., (r, r_m)$ are all the edges in $g$ or $g'$ that start with $r$
22                     Call Algorithm 8 on $G$, $D$, $r$, $r_1$,...,$r_m$
23                 end
24             end
25         end
26     end
27 end
```

Algorithm 10: Provenance Of A Revision Unit

```
   Input: $r = (v, p, s), G = (V, E), D = (U, M)$
   Output: $D(r) = (U(r), M(r))$
1  begin
       /* Compute $U(r)$ as the ``reachable'' set */
2      $U(r) \leftarrow \emptyset$
3      Add $r$ into $U(r)$
4      Initialize $found\_path$ to be an empty list
5      Add $r$ to $found\_path$
6      while $found\_path$ is not empty do
7          Let $last\_v$ be the last element of $found\_path$
8          if exists $(next\_v, last\_v) \in M$ and $next\_v \notin U(r)$ then
9              Add $next\_v$ to $U(r)$
10             Add $next\_v$ to $found\_path$
11         else
12             Remove $last\_v$ from $curr\_path$
13         end
14     end
       /* Compute $M(r)$ */
15     for every edge $(r, r')$ in $M$ do
16         if $r$ in $U(r)$ and $r'$ in $U(r)$ then
17             Add $(r, r')$ to $M(r)$
18         end
19     end
20 end
```

CHAPTER VI

Conclusion

Provenance of derived data helps the users in understanding and interpreting the data, evaluating the reliability of the data and debugging the data. Provenance of derived data lie in the source datasets from which the data are derived. Meanwhile, datasets are subject to modifications, such as removal of existing data and insertion of new data. In presence of modifications, we have to rethink our understanding and retrieval of provenance. In this thesis, we discuss the provenance related problems under the circumstance of modifiable datasets, structured and unstructured. For structured datasets, we focus on relational databases; for unstructured datasets, we focus on plain text documents.

The modifications may remove (part of) the requested provenance from the source dataset. Previous provenance retrieval techniques based on classical tracing queries rely on the provenance being present in the database. When we try to retrieve the lost provenance from the source dataset, we make adjustments to classical tracing queries as shown in Chapter II. The adjusted tracing queries take into considerations the modification log and archived historical values to construct the lost provenance.

Moreover, in case of partially removed provenance, if the user requests the remaining part of the provenance, it is desirable to retrieve that part independently of the rest. Previous provenance retrieval techniques based on classical tracing queries have to use all the source tables involved in the original derivation query. The independent retrieval of partial provenance calls for a technique that can rewrite the classical tracing queries to eliminate their references to certain source tables. In order to do that, we take advantage of the presence of more restrictive predicates than the relational predicates in the classical tracing queries as shown in Chapter III. This technique also enables the optimization of the classical tracing queries and the customized tracing queries.

The modifications may bring in new source tuples as well as remove existing ones. These new source tuples can potentially affect the derived tuples in an opposite way than the classical provenance. While previous work focuses on the classical provenance, i.e., the contributing source tuples to a derived tuple, we show that there exist contradicting source tuples of the derived tuple, which either eliminate the derived tuple or replace it. Given derived tuples, identifying their contradicting tuples can efficiently validate the derived tuples upon source dataset modifications as shown in Chapter IV.

Besides the relational datasets, another type of datasets that are frequently modified are plain text documents. They are unstructured, and have flexible data granularities. Most text editors are capable of keeping the change logs or revision histories of the text documents. However this type of view contains a lot of irrelevant information when the user requests the provenance of only part of a document. We propose a flexible data granularity, called revision units, and use revision units to present the provenance of an arbitrary fragment of text, which captures only the relevant revision information with regard to the given text fragment, as discussed in Chapter V.

Besides the projects presented in this thesis, there are other problems related to the provenance in modifiable datasets. We list two of them below.

(i). First, we have so far explored relational databases and text documents as popular types of modifiable datasets and there are certainly other interesting types of modifiable datasets, e.g., map data. Map data is represented as graphs with attributes attached to nodes and edges. Map data can have different types of modifications, e.g., changing the value of an attribute or changing the structure of the graph. There is also dependency among the changes to map data. To capture the provenance of map data, we need to make decisions on appropriate information units, modification operations and precise meaning of map data provenance.

(ii). Second, we can use the domain specific knowledge of the data to make its provenance more useful and more like knowledge. For example, if a plain text document is a pay stub, then a change of text may reflect a rise in salary; if the document is an inventory of a zoo, then a change of text may reflect the arrival of new animals or equipments. With domain specific knowledge about the underlying data, we can separate important provenance from trivial provenance and separate relevant provenance from irrelevant provenance. For example, a user can ask for provenance related to salaries, then the text changes on addresses don't need to be included in the requested provenance. In order to leverage the domain specific knowledge, the underlying data needs to be tagged with domain knowledge. In relational databases, the attribute names may be a good indicator of domain knowledge. In XML databases, the XML tags may be such an indicator. In plain text documents, tags need to be added explicitly. It can be done by domain experts and/or automatic tagging that makes use of natural language processing and/or text mining.

APPENDIX A

Tracing Queries and Aggregations

A.1 Tracing Queries and Provenance

In this appendix, we are going to show that the provenance retrieved by the tracing query as shown in Equation 2.4 is actually the provenance defined in Definition 2.2. If this is shown to be true, then the extended tracing query as shown in Equation 2.5 can retrieve the provenance defined in Definition 2.2 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' and T_2' are the provenance retrieved by the tracing query. In order to show that they are exactly the defined provenance by Definition 2.2, we only need to show

(i). $T_k' \subset T_k$ ($k = 1, 2$)

(ii). $\bar{t} \in Q(T_1', T_2')$

(iii). $\forall s_k' \in T_k' : Q(T_1', ..., \{s_k'\}, ..., T_2') \neq \emptyset$

(iv). 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 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, $Q(\{s_1'\}, T_2') \neq \emptyset$.

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 2.2, 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

(i). these two tuples have the same provenance according to Definition 2.2,

(ii). and they also have the same provenance retrieved by the tracing query in Equation 2.4,

(iii). and the provenance of t retrieved by the tracing query matches the provenance of t defined by Definition 2.2,

then we can say that the provenance of t_{agg} retrieved by the tracing query matches the provenance of t_{agg} defined by Definition 2.2.

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 2.2.

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 2.2. 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 2.4 is the provenance defined in Definition 2.2.

A.2 Aggregations in Formula

We only allow aggregate functions/attributes in the target list. In Section 2.2, 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 greater 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.

REFERENCES

[1] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[2] Peter Buneman, Adriane Chapman, and James 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.

[3] Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn 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.

[4] Peter Buneman, James Cheney, and Stijn 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.

[5] Peter Buneman, Sanjeev Khanna, and Wang chiew Tan. Why and where: A characterization of data provenance. In *In ICDT*, pages 316–330. Springer, 2001.

[6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.

[7] Adriane 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.

[8] Adriane P. Chapman, H. V. Jagadish, and Prakash 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.

[9] James Cheney, Umut A. Acar, and Amal Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

[10] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In *DBPL'07: Proceedings of the 11th international conference on Database programming languages*, pages 138–152, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Yingwei Cui. *Lineage Tracing In Data Warehouses*. PhD thesis, Stanford University, December 2001.

[12] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *In ICDE*, pages 367–378, 1999.

[13] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1345–1350, New York, NY, USA, 2008. ACM.

[14] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated xml: queries and provenance. In Maurizio Lenzerini and Domenico Lembo, editors, *PODS*, pages 271–280. ACM, 2008.

[15] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. pages 174 –185, mar. 2009.

[16] Boris Glavic and Gustavo 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.

[17] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *In Very Large Data Bases (VLDB*, pages 675–686, 2007.

[18] Todd J. Green, Grigoris Karvounarakis, and Val 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.

[19] Christopher Browne Greg Smith, Robert Treat. Tuning your postgresql server. $http : //wiki.postgresql.org/wiki/Tuning\_Your\_PostgreSQL\_Server/$.

[20] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166, New York, NY, USA, 1993. ACM.

[21] Melanie Herschel and Mauricio A. Hernández. Explaining missing answers to spjua queries. *Proc. VLDB Endow.*, 3:185–196, September 2010.

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

[23] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1):736–747, 2008.

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

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

[26] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1108–1119, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Benjamin Livshits. Dynamine: Finding common error patterns by mining software revision histories. In *In ESEC/FSE*, pages 296–305. ACM Press, 2005.

[28] David Lomet, Roger Barga, and Rui Wang. Transaction time support inside a database engine. In *In Proceedings of the 22nd ICDE Conference*, 2006.

[29] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. Why so? or why no? functional causality for explaining query answers. *CoRR*, 2009.

[30] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *CoRR*, abs/1009.2021, 2010.

[31] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, 2008.

[32] Rani Nelken and Elif Yamangil. Mining wikipedias article revision history for training computational linguistics algorithms, 2008.

[33] Sudarshan Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *In Proceedings of the International Conference on Very Large Databases*, pages 501–511, 1991.

[34] Mikalai Sabel. Structuring wiki revision history. In *Proceedings of the 2007 international symposium on Wikis*, WikiSym '07, pages 125–130, New York, NY, USA, 2007. ACM.

[35] Wang-Chiew Tan. Research problems in data provenance. *IEEE Data Engineering Bulletin*, 27:45–52, 2004.

[36] Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. Studying cooperation and conflict between authors with history flow visualizations. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 575–582, New York, NY, USA, 2004. ACM.

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

[38] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard Thomas Snodgrass, V. S. Subrahmanian, and Roberto Zicari. *Advanced database systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[39] Honglei Zeng, Maher A. Alhossaini, Richard Fikes, and Deborah L. Mcguinness. Mining revision history to assess trustworthiness of article fragments. 2006.

[40] Jing Zhang and H.V. Jagadish. Lost source provenance. In *EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology*, 2010.