Provenance in a Modifiable Data Set

Jing Zhang and H.V. Jagadish

University of Michigan
{jingzh,jag}@umich.edu

Abstract. Provenance of data is now widely recognized as being of great importance, thanks in large part to pioneering work [4, 6] by Peter Buneman and his collaborators in a stream that continues to produce influential papers today [1–3, 7]. When we consume data from a database, we often care about where these data come from, how they were derived, and so forth. We may desire answers to such questions to establish trust in the data, to investigate suspicious values, to debug code in the system, or for a host of other reasons. Considerable recent work has addressed many issues related to provenance. However, the standard assumption is that data sources, from which result data have been derived, are static. In reality, we know that most data are modified over time, including data sources used for deriving results of interest. When we consider provenance in the context of such modifications, many new problems arise. This chapter addresses two key problems in this context:

1. Result data may no longer be valid after a source update. How can we efficiently determine whether a given result tuple is valid? When a result tuple is invalidated, can we explain what caused this invalidation?
2. We may have lost access to (some) source data. In such a situation, can we determine what is the missing source data on which some result tuple depends?

1 Validating an Answer

In a modern scientific project, there frequently is a huge body of raw data collected from experiments. Usually, this body of data is stored in a database, and processed by SQL queries to make it ready for further analysis. These derived data are vital for the final scientific conclusions the scientists draw from the experiments. When the raw data change, 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 [11] 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.

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

V. Tannen et al. (Eds.): Buneman Festschrift, LNCS 8000, pp. 557–567, 2013.
© Springer-Verlag Berlin Heidelberg 2013

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 1. Assume we have two simple tables *Orders* and *Customers* as shown in Figure 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 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 1.

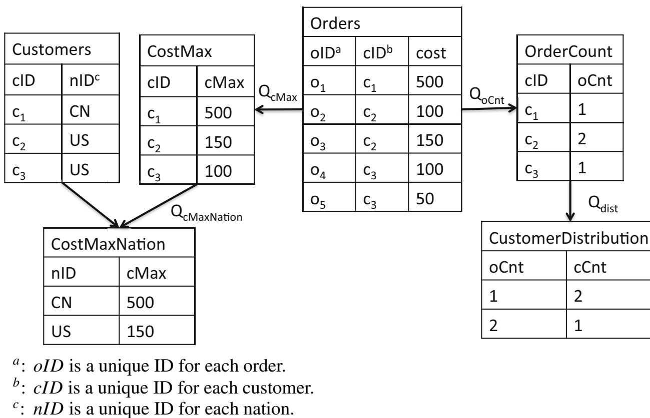

Fig. 1. Source Table And Derived Tables

Suppose we have updates to *Orders* table as *ΔOrders*, shown in Figure 3. The *CNT* attribute in *ΔOrders* is the number of derivations of each tuple. Tuples with positive *CNT* are to-be-inserted tuples and tuples with negative *CNT* are to-be-removed tuples. *ΔOrders* leads to the update *ΔCostMax* to the result table *CostMax*. For example,

[1] Tuples in delta tables can have positive counts or negative counts [11]. We call tuples with positive counts positive tuples, and tuples with negative counts negative tuples.

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
$r_{cMaxNation}$:
$CostMaxNation(nID, max(\langle cMax \rangle)\ AS\ \ cMaxNation) :-$
$CostMax(cID, cMax), Customers(cID, nID)$
$? - CostMaxNation(nID, cMaxNation)$

Fig. 2. Example Queries

$(o_6, c_3, 150) \in \Delta Orders$ is inserted into the 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 $(c_3, 100)$ 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 contributory provenance and the contradictory provenance have an interesting duality and correspondence. When an answer is invalidated, a negative version of it shows up in the delta answer set, e.g., $(c_1, 500, -1)$ in $\Delta CostMax$ indicates the invalidation of $(c_1, 500)$ in the original answer set $CostMax$. Therefore, the contributory provenance of the negative version of an answer in the delta answer set is in fact the contradictory provenance of the answer.

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

ΔOrders

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

ΔOrderCount

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

ΔCustomerDistribution

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

ΔCostMax

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

Fig. 3. (Delta) Tables Extended With *CNT*

The contributory provenance of the negative version consists of tuples from delta source tables and/or original source tables, and consists of both positive tuples and negative tuples. The queries that produce the negative version are delta query rules [11], which are derived from the original query. Then, the contributory provenance of the negative version of an answer can be retrieved by tracing queries based on the delta query rules that produced the negative version.

In general, we can validate selected answers in the following two 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)

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.

Since the pruning predicates are constructed for the delta rules and the delta rules evaluate over delta tables, the pruning predicates have to deal with both the positive tuples (i.e., to-be-inserted tuples) and the negative tuples (i.e., to-be-deleted tuples) in the delta tables. The positive tuples and negative tuples in the delta source tables affect the given answer in different ways. For example, if the given answer is the current maximum, then the positive tuples with a greater value have the potential to invalidate the current maximum while the negative tuples with the same value as the current maximum have the potential to invalidate the current maximum. Therefore, a pruning predicate is a disjunction of two predicates, one for the positive tuples and one for the negative tuples.

If the answer is derived through a single query rule, the pruning predicates are constructed directly based on the given answer. If the answer is derived through a stratified Datalog program consisting of multiple rules, and the single rule with the highest

stratum produces the final answer. Then the pruning predicates for the rule with highest stratum is constructed directly based on the given answer; and the pruning predicates for any other rule are inferred from the pruning predicates for rules with higher strata.

In Step 2, if we find a negative version of the selected answer in the delta result table, we know that the given answer is invalidated. Since this negative version is produced by the delta rules from the delta source tables and possibly original source tables, we can find the contributory provenance of this negative version in the delta source tables (and original source tables) using classical tracing queries derived from the delta rules. This contributory provenance of the negative version, also being the contradictory provenance of the given answer, explains the invalidation of the given answer.

A related problem has also been studied in [10]. The update techniques given there for the count of the derivations of each view tuple can easily be generalized to update the complete provenance of the view tuple instead. However, the update technique in [10] only applies to SPJU queries without aggregations. Furthermore, this technique also updates the entire derived dataset instead of just the subset of interest to the user.

1.1 Explanation of the Absence of Expected Answer

We considered above the question of identifying updates to source data that caused a result tuple to be invalidated. We have previously studied a closely related question of explaining why some expected result tuple is missing from the answer set [8]. In this previous work, we are not specifically looking at source updates.

When some answer tuples that are expected to be in the result set are missing, we seek to identify particular source tuples or particular manipulations in the derivation responsible for their absence. Such input data are defined to be *unpicked* and such manipulations are defined to be *picky* manipulations for these unpicked data. We proposed both top-down and bottom-up approaches to search over the derivation process to find the picky manipulations.

In other related work, [13] showed that proper changes can be made to some attribute values in the source data that have previously failed to produce the expected result tuples, such that these modified source data can now go through the query evaluation and produce the expected result tuples. [14] introduced the concept of functional causes, which explains the presence and absence of answers. Similar to [13], [12] also provides instance-based explanations for missing answers, but is more general since its technique can apply to a set of SPJUA queries instead of SPJ queries.

2 Lost Source Provenance

Modifications to a source data set may delete (or update/overwrite) some or all of the source data from which a result of interest was derived. Even in the absence of modifications, it is possible that a data source becomes unavailable, for instance because it is remote and goes off-line or because it is owned by an entity that decides to take it private.

In consequence, the provenance of an answer can be (partially) removed from the source data set. In order to retrieve this (partially) lost provenance when requested,

we have two possible strategies with different trade offs between the provenance we can provide and the storage/time overhead we are willing to pay.

One way to avoid this problem is to store a version of the source at the time the result was derived. We can thereby guarantee no provenance will be lost, but there is storage cost for keeping a duplicate of the source, and this cost could be substantial. Moreover, if we operate in an environment in which result tuples are lazily updated from the source, we may have to keep multiple versions of the source to meet the provenance needs of all result tuples.

In this section, we develop a second strategy. We show how we can add three (small) extra data structures to the database, and use these to recover the lost provenance.

First, we define the provenance of a given derived tuple as follows. It is a modified version of the definition introduced in [9].

Definition 1. *Given a database D of tables* $T_1, ..., T_n$*, a query Q and a derived tuple t, there exists a set of tables* $T'_1, ..., T'_n$ *such that*

- $T'_i \subseteq T_i$, *where* $i = 1, ..., n$
- $\{t\} = Q(T'_1, ..., T'_n)$
- $\forall T'_k : \forall t' \in T'_k : Q(T'_1, ..., T'_{k-1}, \{t'\}, T'_{k+1}, ..., T'_n) \neq \emptyset$

Notice that if a single table has more than one instance in the query, each instance is considered a separate table.

Second, we describe the three extra data structures we need to retrieve the possibly overwritten provenance. With these three data structures, we can have standard tracing queries modified to make use of them and retrieve the lost provenance. We refer to these queries as *extended tracing queries*.

1. We need a log, denoted as *provenance log*, recording the operations that have taken place over a time period till the current time point, beginning from some defined origin. Every entry records one operation and each entry has a unique log ID, which can be used to identify the operation in this entry.
2. We associate with each tuple in the current database an extra attribute, denoted as *since*, storing a log ID, which indicates the operation that introduced this tuple into the database.
3. We also associate with each table in the current database a so-called *shadow table* that keeps the tuples that were once in the database table but have been removed at some time point. In particular, the shadow table has the same schema as the database table except for two extra attributes storing log IDs, denoted as *begin* and *end*, with *begin* indicating the operation that introduced the tuple into the database and *end* indicating the operation that removed the tuple from the database.

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.

The shadow tables are for the historical tuples. 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.

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. Provenance capture and retrieval are the only procedures that can set its value or query it.

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

1. 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.
2. When a tuple is inserted into a table due to this database operation, the value of its *since* attribute is set with the ID of the newly created entry in the provenance log.
3. When a tuple is removed from a table due to this database operation, either by a delete or by an update, the removed tuple is inserted into the corresponding shadow table. 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. All the provenance information we need is recorded in these auxiliary structures.

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 standard tracing query, the extended tracing query need an extra piece of information, i.e., the ID of the provenance log entry that records the original query. 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*. These IDs are even better than real timestamps since they incur no ambiguity.

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

derivation time. In particular, if a source tuple's life span (identified by *begin* and *end*) covers the derivation time, this source tuple is eligible to be in the provenance. The extended tracing query only uses those eligible source tuples and retrieves the (lost) provenance from them.

In general, the construction of an extended tracing query needs three pieces of information:

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

Given a tuple t, suppose it is derived from the original query Q shown in Equation 1, and further suppose Q is logged in a provenance log entry with ID being id, then the extended tracing query to retrieve provenance in the table T_k is as shown in Equation 2.

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

$$\begin{aligned} \{\ & s_k : \langle B_1, ..., B_l\rangle \mid \\ & \exists t, s_1, ..., s_{k-1}, s_{k+1}, ..., s_m \\ & (T_1^H(s_1) \wedge ... \wedge T_m^H(s_m) \wedge f(s_1, ..., s_m, t) \\ & \wedge t.A_1 = a_1 \wedge ... \wedge t.A_n = a_n)\ \} \end{aligned} \tag{2}$$

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

$$\begin{aligned} \{\ & s_k : \langle B_1, ..., B_l\rangle \mid \\ & (T_k(s_k) \wedge s_k.since < id) \vee \\ & \exists s'_k(T_{k_sh}(s'_k) \wedge s'_k.begin < id \wedge s'_k.end >= id \\ & \wedge s'_k.B_1 = s_k.B_1 \wedge ... \wedge s'_k.B_l = s_k.B_l)\ \} \end{aligned} \tag{3}$$

Notice that, although the original query is a conjunctive query, with aggregation in this case, the extended tracing query is not a conjunctive query, because of the union connective used in Equation 3.

These three data structures incur some space overhead. We discuss next how to minimize this overhead.

First of all, the provenance log does not need to take extra space in practice, since all database management systems keep some kind of log and the provenance log can be implemented as a view over the system maintained logs. This is almost always possible since the really vital attributes in the provenance log are the log ID and the operation, which are very basic information a typical system log will keep.

As for the space overhead due to the attribute *since*, the number of cells of this attribute is equal to the number of tuples in the database. Since the database tuples usually have multiple attributes and some of them are of more space-costly data types than integer type, the total cost of this extra attribute in integer type is only a fraction of the total size of the database.

The shadow tables are the costliest of the three auxiliary data structures in terms of space. The size of shadow tables grows with the number of tuples that have been updated or removed. In a database with a moderate amount of change to data, the space cost due to shadow tables may be acceptable. Intuitively, this cost is unavoidable: if there is change to data and we need past values, we have to store them somewhere. The cost of shadow tables is much less than the cost of storing a version of the source database for each derived value.

In general, 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 [5, 15]. 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 increasing the time cost of reconstructing previous versions using historical data. For example, if the whole database is archived, the reconstruction of any table in the database at a previous time involves no complex queries but almost merely selecting. Comparatively, since we only archive the updated tuple when one or more attributes in it are changed, the reconstruction of the involved table needs to run a query as shown in Equation 3.

The book-keeping of these three data structures also incurs some time overhead during the execution of an operation.

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 both the log entry and 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 to execute it varies with the reconstructed historical versions.

The historical version of a table consists of tuples from the current table and from the shadow table. The execution 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. If most of the tuples in the output of concatenation are from the same table, (the sort at the heart of) the union may be faster than in the case where tuples come evenly from the two tables.

3 Conclusions

We live in a dynamic world and the digital artifacts we rely on must change to keep pace with the world we live in. Provenance is more difficult to specify and to work with when we cannot rely on immutable objects and data sources. However, it is possible to do and, in any case, we have no choice in the matter given that we live in a dynamic world. This chapter considered some of the challenges that arise due to change, and suggested solutions to these challenges.

References

1. Buneman, P., Chapman, A., Cheney, J.: Provenance management in curated databases. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 539–550 (2006)
2. Buneman, P., Cheney, J., Lindley, S., Müller, H.: Dbwiki: A structured wiki for curated data and collaborative data management. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 1335–1338 (2011)
3. Buneman, P., Cheney, J., Tan, W.-C., Vansummeren, S.: Curated databases. In: Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–12 (2008)
4. Buneman, P., Khanna, S., Tan, W.C.: Data provenance: Some basic issues. In: Foundations of Software Technology and Theoretical Computer Science, pp. 87–93 (2000)
5. Buneman, P., Khanna, S., Tajima, K., Tan, W.C.: Archiving scientific data. ACM Trans. Database Syst. 29, 2–42 (2004)
6. Buneman, P., Khanna, S., Tan, W.-C.: Why and where: A characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2000)
7. Buneman, P., Tan, W.-C.: Provenance in databases. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1171–1173 (2007)
8. Chapman, A., Jagadish, H.V.: Why not? In: Proceedings of the 35th SIGMOD International Conference on Management of Data, pp. 523–534 (2009)
9. Cui, Y., Widom, J.: Practical lineage tracing in data warehouses. In: Proceedings of the 15th International Conference on Data Engineering, pp. 367–378 (1999)
10. Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update exchange with mappings and provenance. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 675–686 (2007)

11. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 157–166 (1993)
12. Herschel, M., Hernández, M.A.: Explaining missing answers to spjua queries. Proc. VLDB Endow. 3, 185–196 (2010)
13. Huang, J., Chen, T., Doan, A., Naughton, J.F.: On the provenance of non-answers to queries over extracted data. Proc. VLDB Endow. 1(1), 736–747 (2008)
14. Meliou, A., Gatterbauer, W., Moore, K.F., Suciu, D.: Why so? or why no? functional causality for explaining query answers. In: CoRR (2009)
15. Müller, H., Buneman, P., Koltsidas, I.: Xarch: Archiving scientific and reference data. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1295–1298 (2008)