On the Propagation of Deletions and Annotations through Views

Buneman, P., Khanna, S., & Tan, W-C. (2002). On the Propagation of Deletions and Annotations through Views. In PODS '02 Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. (pp. 150-158). ACM. doi: 10.1145/543613.543633

On Propagation of Deletions and Annotations Through Views

Peter Buneman*
University of Edinburgh
University of Pennsylvania
peter@cis.upenn.edu

Sanjeev Khanna†
University of Pennsylvania
sanjeev@cis.upenn.edu

Wang-Chiew Tan‡
University of Pennsylvania
wctan@saul.cis.upenn.edu

ABSTRACT

We study two classes of view update problems in relational databases. We are given a source database S, a monotone query Q, and the view $Q(S)$ generated by the query. The first problem that we consider is the classical view deletion problem where we wish to identify a minimal set T of tuples in S whose deletion will eliminate a given tuple t from the view. We study the complexity of optimizing two natural objectives in this setting, namely, find T to minimize the side-effects on the view, and the source, respectively. For both objective functions, we show a dichotomy in the complexity. Interestingly, the problem is either in P or is NP-hard, for queries in the same class in either objective function.

The second problem in our study is the annotation placement problem. Suppose we annotate an attribute of a tuple in S. The rules for carrying the annotation forward through a query are easily stated. On the other hand, suppose we annotate an attribute of a tuple in the view $Q(S)$, what annotation(s) in S will cause this annotation to appear in the view, minimizing the propagation to other attributes in $Q(S)$? View annotation is becoming an increasingly useful method of communicating meta-data among users of shared scientific data sets, and to our knowledge, there has been no formal study of this problem.

Our study of these problems gives us important insights into computational issues involved in data provenance or lineage — the process by which data moves through databases. We show that the two problems correspond to two fundamentally distinct notions of provenance, *why* and *where-provenance*.

*Supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444. Currently at University of Edinburgh.

†Supported in part by an Alfred P. Sloan Research Fellowship and by an NSF Career Award CCR-0093117.

‡Supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444.

1. INTRODUCTION

Given a desired update to a view of a database, what update to the source tables should be made to effect this update to the view? This is the view update problem, which has a long history in database research. Its importance derives from the fact that most access to databases is through views. Unfortunately, only in very restricted circumstances there is a unique update to a source database S that will cause a specified update to the view $Q(S)$. If we cannot find a unique update to S, an alternative is to find a *minimal* update to S that will cause the specified update to $Q(S)$. Here, there are two ways we can measure minimality: the first is by the number of changes in the source tables S; the second is by the number of side effects that changes in S cause in $Q(S)$ (in addition to the required update.) We study this minimization problem for two kinds of updates. The first is *deletions* to $Q(S)$ that are to be generated by deletions to S. The second kind update is that of an *annotation* placed on a location in the view $Q(S)$. This is novel view update problem, and it gives us interesting insights into issues of propagating annotations and of data provenance.

Deletion Minimization. In deletion minimization, the input is a source database S, a query Q, the view $V = Q(S)$, and a tuple $t \in V$. The *view side-effect* problem is to find a set $T \subseteq S$ so as to minimize $|\Delta V|$, where $Q(S \setminus T) = V \setminus (\Delta V \cup \{t\})$. In other words, we wish to find a set of source tuples whose removal will delete t while minimizing the number of other tuples deleted from the view. The *source side-effect* problem is to simply to find a smallest set T that causes the removal of t (regardless of the size of the side-effect in $Q(S)$.) Our goal is to study the complexity of these problems over the monotone fragment of relational queries. We will show that these problems are NP-hard in general. This is perhaps not surprising when the full generality of SPJU queries is allowed. However, we show that these problems become NP-hard even over a very restricted fragment where only join with either project or union

is allowed. For the case of view side-effect problem, we show that it is NP-hard to decide whether or not there exists a T whose deletion only deletes the tuple t from the output. Thus this problem is inapproximable. For the case of source side-effect an interesting phenomenon occurs whereby every class of queries is either poly-time solvable or is NP-hard via an approximation-preserving reduction from the set-cover problem[1]. The set-cover problem is known to be $O(\log n)$-approximable by a simple greedy algorithm[2]. It is known that unless NP $\subseteq$ DTIME($n^{\log \log n}$), no polynomial-time algorithm can achieve $o(\log n)$-approximation for this problem [12]. Altogether, our results show a dichotomy in the complexity of these deletion problems for any subclass of SPJU queries.

Annotation Placement. The propagation of annotations is a new topic and deserves some introduction. An increasing amount of scientific communication is carried out through shared annotations of existing data. In some cases the annotations are anticipated when the database is designed so that fields and/or tables are created to hold annotations. In this case annotating a view is an example of the standard view update problem.[3] A second form of annotation is much looser and we may allow annotations on annotations. Hence it is impractical to accomodate annotations by adding extra fields in the source relations. The changes are usually not anticipated and the annotators may not have update privileges to the database so that annotations have to be stored in a separate database. An example of this kind of annotation is found in biological annotation servers [9]. Although specific to genetic sequence annotation, there are issues in the design of the software of how widely an annotation can spread. From a database perspective an annotation is interesting in that a query cannot "see" the annotation, it can only transmit it. For such queries, we formulate the *annotation placement* problem as follows. Given a source database S, a query Q, the view $Q(S)$ generated by Q, and a location in $Q(S)$, find a location to annotate in the source such that annotating it propagates the annotation to a smallest number of locations including the specified view location. We develop a natural system of rules for propagating annotations from source to view, and show that under this system of rules, the annotation placement problem is NP-hard for queries that allow project and join together. However, for other relational queries (any normal form SJU queries or any SPU queries), the problem is polynomial time solvable. As before, our results establish a dichotomy theorem for the complexity of the annotation placement problem. We also establish a normal form theorem that allows us to rewrite queries but does not change the propagation of annotations.

Annotations, Deletions and Provenance. Data *provenance* and *lineage* are synonyms for the study of how, in an environment in which data is repeatedly copied and transformed, we trace the origins of a piece of data. Thus provenance is closely connected with views, which are simply queries that describe the tranformation of one database to another. In [7], two forms of provenance are discussed: *why-provenance*, which describes the reason, e.g., a proof tree, for the existence of a data item in the output. This is the form of "lineage" described in [15] and used in [14] to compute an exact deletion-to-deletion translation. A second form, *where-provenance*, describes the path by which a data item was copied into the database. This form of provenance is closely related to the transport of annotations through a view. Thus it is appropriate to study computational issues of both forms of provenance together. In fact, one interesting outcome of our work is a result (Corollary 3.1) that shows the intractability of tracing both forms of provenance through a view, and thus indicates that some limitations need to be placed on view definitions in order to make any progress with the study of provenance.

Related Work. While the view update problem has a long history, with the exception of [8], which discusses the complexity of finding the "complement" ([4]) of a view, we could find no discussions of the complexity of some of the basic problems in this area, such as that of finding even one witness [4] for a view deletion.

The problem of translating view deletions to source deletions has been studied in [14] which gives an algorithm that exploits lineage information[15] to find an exact deletion-to-deletion translation (i.e no side-effects) whenever possible. The lineage information is used to as a starting point, to enumerate all candidate witnesses for a deletion. However, as we just remarked, it is NP-hard to find all witnesses for a tuple in the output.

Previous work on view updates has explored general translations from views to source [11, 1, 2, 4, 8]. The kinds of updates allowed are generally insertions, modifications and deletions to the view. They generally consider different kinds of translations. For example, an insertion may be translated to a source insertion or even a source deletion. Hence, the view update translation process is generally ambiguous since there are usually many possible ways to translate a view update to source update(s).

In [11], the concept of *clean sources* was discussed. In the context of view deletion from views, clean sources correspond to source tuple(s) which when deleted will delete exactly the desired view tuple, i.e, no side-effects. And in the general setting of view updates, exact trans-

[1] In the set-cover problem, we are given $\mathcal{S} = \{S_1, ..., S_m\}$ where each S_i contains a subset of elements from $X = \{x_1, ..., x_n\}$. The goal is to find a $\mathcal{S}' \subseteq \mathcal{S}$ such that $\bigcup_{S \in \mathcal{S}'} S = X$ and $|\mathcal{S}'|$ is the smallest.

[2] An α-approximation algorithm for a problem is a polynomial time algorithm that computes a solution of cost at most α times the optimal cost.

[3] In practice such databases are heavily *curated*. The maintainers of the database carefully scrutinize and control changes to the data.

[4] A witness for a tuple t in a view is a minimal subset S' of source data S such that $t \in Q(S')$

lations (i.e, no side-effects) are only possible when functional dependencies are also considered. Finally, [1, 2] study the view update problem for select-project-join queries on relations that are in Boyce-Codd Normal Form. Various criteria are proposed to capture "acceptable" view update translations.

Annotation is an example of superimposed information as described in [3]. For example, bookmark files and the schema of a database are examples of superimposed information (data "placed over" existing information). There are also emerging efforts to build annotation systems such as Annotea [13] and BioDAS [9]. Annotea allows one to annotate on web pages through specialized web browsers and BioDAS allows one to annotate on genome sequences. However, to our knowledge, there has been no formal study of the annotation placement problem.

Organization. Section 2 studies the view deletion problem from both the source perspective as well as the view perspective. In Section 3 we study the annotation placement problem. Finally, we conclude with some remarks in Section 4.

2. DELETIONS IN THE VIEW

In this section, we consider the problems of minimizing side-effects on the view (or source) when we are allowed to delete a tuple in the view. The problem of minimizing side-effects on the view is to find a set of source tuples to delete so that the number of other tuples that are deleted in the view as a result, is minimized. The problem of minimizing side-effects on the source is to find a minimum set of source tuples to delete so as to delete the tuple in the view.

2.1 Minimizing Side-effects on the View

The table below is a summary on the complexity of determining whether there is a side-effect-free deletion for any subclass of SPJU queries. There is a dichotomy in the complexity for this problem.

A somewhat surprising result is that for the class of queries involving projection and join, it is already NP-hard to decide whether there is a side-effect-free deletion. We say a set of source deletions is *side-effect-free* if deleting those source tuples do not delete any other tuple in the view, other than the desired view tuple. The result is true even when the query is of constant size and only involves two relations.

Query class	Deciding whether there is a side-effect-free deletion
Queries involving PJ	NP-hard
Queries involving JU	NP-hard
SPU	P
SJ	P

2.1.1 Project and Join Queries

We will show that once we allow these two operators together, the view side-effect problem becomes NP-hard even when the source consists of two relations with two attributes each. For instance, consider an example of two relations taken from [14]: UserGroup(user,group) and GroupFile(group,file) where each user may belong to several groups and a file may be shared by several groups. For a query that projects out the user and file attributes after a join on the two relations, i.e, $\Pi_{\text{user,file}}(\text{UserGroup} \bowtie \text{GroupFile})$, it is NP-hard to decide if a (user,file) combination can be deleted from the output in a side-effect-free manner.

The intuition behind why the above query makes it difficult to determine if there is a side-effect-free deletion is that an output tuple in the result of this query may have many witnesses (due to projection) and there may be many possible ways of destroying each witness (due to join). A witness is destroyed if one of the tuples in the witness is removed from the source. Therefore the combination of project and join gives rise to many possible ways of removing an output tuple. The difficulty arises when one has to consider how a set of source tuple deletions affects the existence of other output tuples in order to minimize side-effects.

Our starting point is a NP-hard problem known as *monotone* 3SAT:

Input: A 3-CNF formula where each clause consists either of all positive literals or all negated literals.

Goal: Decide whether or not the formula is satisfiable.

The NP-hardness of this variant of 3SAT was shown by Gold [5] and also follows from Schaefer's Theorem [10].

THEOREM 2.1. *The problem of deciding whether there is a side-effect-free deletion for a PJ query in normal form is NP-hard.*

PROOF. The proof is by reduction from monotone 3SAT. Our reduction uses two relations with schema $R_1(A, B)$ and $R_2(B, C)$. For every variable x_i, there is a tuple (a, x_i) in R_1 and a tuple (x_i, c) in R_2. In addition, for each clause $C_i = (x_{i_1} + x_{i_2} + x_{i_3})$, R_1 contains tuples (a_i, x_{i_1}),(a_i, x_{i_2}) and (a_i, x_{i_3}) where a_i is a fresh constant that does not occur elsewhere in R_1. Similarly for each clause $C_j = (\overline{x_{j_1}} + \overline{x_{j_2}} + \overline{x_{j_3}})$, R_2 contains tuples (x_{j_1}, c_j),(x_{j_2}, c_j) and (x_{j_3}, c_j) where c_j is a fresh constant that does not occur elsewhere in R_2. For example, an encoding of the instance $(\overline{x_1} + \overline{x_2} + \overline{x_3})(x_2 + x_4 + x_5)(\overline{x_4} + \overline{x_1} + \overline{x_3})$ is shown in Figure 1.

The project join query is given by $\Pi_{A,C}(R_1 \bowtie R_2)$ which produces (i) the tuple (a, c), (ii) a tuple (a_i, c) for each clause $C_i = (x_{i_1} + x_{i_2} + x_{i_3})$, and (iii) a tuple (a, c_j) for each clause $C_j = (\overline{x_{j_1}} + \overline{x_{j_2}} + \overline{x_{j_3}})$. The goal is to delete (a, c). It is easy to see that in order to do so, for each variable x_i, we must delete either (a, x_i) or (x_i, c). We now show that there exists a side-effect-free solution to this view deletion problem if and only if the given 3SAT instance is satisfiable.

R_1

A	B
a	x_1
a	x_2
a	x_3
a	x_4
a	x_5
a_2	x_2
a_2	x_4
a_2	x_5

R_2

B	C
x_1	c
x_2	c
x_3	c
x_4	c
x_5	c
x_1	c_1
x_2	c_1
x_3	c_1
x_4	c_3
x_1	c_3
x_3	c_3

$\Pi_{A,C}(R_1 \bowtie R_2)$

A	C
a	c
a	c_1
a	c_3
a_2	c
a_2	c_1
a_2	c_3

Figure 1: Relations involved in reduction of Theorem 2.1.

Suppose we are given a satisfying assignment to the 3SAT instance. For each variable x_i, if it is set to true then we delete (a, x_i), and otherwise we delete (x_i, c). It is easy to verify that since this is a satisfying assignment, each clause $C_i = (x_{i_1} + x_{i_2} + x_{i_3})$, at least one of $(x_{i_1}, c), (x_{i_2}, c)$ or (x_{i_3}, c) will survive and thus (a_i, c) would continue to be in the output. A similar argument applies for clause with all negated literals.

To see the converse, consider any solution for deleting the tuple (a, c) from the view such that no other tuple in the output is affected. We may assume without any loss of generality, that the given solution only deletes one of (a, x_i) or (x_i, c). We will interpret deletion of (a, x_i) as assigning true to variable x_i and deletion of (x_i, c) as assigning false. Consider a clause $C_i = (x + y + z)$. Once again it is easy to verify that since (a_i, c) is in the output, at least one of x, y, or z was set to true. A similar argument applies to clauses with all negated literals. □

Functional Dependencies and Foriegn Key Constraints. Fortunately, most joins are performed on foriegn keys. It is easy to show that project join queries based on key constraints (eg. lossless joins with respect to a set of functional dependencies) allow us to decide whether there is a side-effect-free deletion in polynomial time. See, for example, [11, 1].

2.1.2 Join and Union Queries

We show that even in the absence of projection, the problem remains NP-hard if we allow union.

THEOREM 2.2. *The problem of deciding whether there is a side-effect-free deletion for a JU query in normal form is NP-hard.*

PROOF. The reduction is again from monotone 3SAT formulas. Given a monotone 3SAT formula consisting of clauses $C_1, C_2, ..., C_m$ defined over variables x_1,...,x_n, we introduce $2(m + n)$ relations as follows. There are two relations $R_i(A_1)$ and $R'_i(A_2)$ for each variable x_i where $R_i(A_1)$ contains a single tuple T and $R'_i(A_2)$ contains a single tuple F. For each clause C_i, there are two relations $S_i(A_2)$ and $S'_i(A_1)$ which each consists of a single tuple c_i.

Our query consists of union of $m+n$ queries, $Q_1, ..., Q_m$, ..., Q_{m+n}. For $1 \leq i \leq m$, the query Q_i corresponds to clause C_i and is itself a union of 3 queries. If $C_i = (x_{i_1} + x_{i_2} + x_{i_3})$, then $Q_i = (R_{i_1} \bowtie S_i) \cup (R_{i_2} \bowtie S_i) \cup (R_{i_3} \bowtie S_i)$. A similar query is written for clauses with all negated literals — we replace R_i with R'_i and S_i with S'_i. For $1 \leq j \leq n$, the query Q_{m+j} corresponds to the variable x_j and consists of $R_j \bowtie R'_j$. The output of these queries consists of $m + 1$ tuples: (i) a tuple (T, c_i) for each clause $C_i = (x_{i_1} + x_{i_2} + x_{i_3})$ (generated by Q_i), (ii) a tuple (c_j, F) for each clause $C_j = (\overline{x_{j_1}} + \overline{x_{j_2}} + \overline{x_{j_3}})$ (generated by Q_j), and (iii) a tuple (T, F) (generated by each of $Q_{m+1}, Q_{m+2}, ..., Q_{m+n}$). The goal is to delete the tuple (T, F) from the output. It is easy to see that in order to do so, we must delete either the tuple T from relation R_i or tuple F from relation R'_i. We now show that there exists a side-effect free solution to this view deletion problem if and only if the given formula is satisfiable.

Suppose we are given a satisfying assignment for the monotone 3SAT instance. For each variable x_i, if it is assigned value true, we delete the tuple F from the relation R'_i, and otherwise delete the tuple T from R_i. It is easy to verify that this deletes only tuple (T, F). For the converse, we assume without loss of any generality that the given solution deletes exactly one tuple from either R_i or R'_i. We construct an assignment by assigning to each variable x_i true if T remains in R_i and false otherwise. Since this is a side-effect free deletion, every other tuple (T, c_i) or (c_j, F) remains the in the output. This means the corresponding 3SAT clause is satisfiable.

An example reduction from the formula $(\overline{x_1}+\overline{x_2}+\overline{x_3})(x_2+x_4+x_5)(\overline{x_4}+\overline{x_1}+\overline{x_3})$ is shown in Figure 2. □

2.1.3 SPU and SJ Queries

THEOREM 2.3. *There is always a side-effect-free deletion for SPU queries and is poly-time solvable.*

PROOF. We will establish that for SP queries there is always a unique solution to the deletion problem. Thus it suffices to show that the problem is poly-time solvable for SP queries and the result follows for SPU queries. It takes linear time (in the size of the source database) to select all tuples that satisfy the select condition in the query. A second pass over these tuples identifies the ones that project on to the specified output tuple t. We must delete all of them and clearly, there deletion suffices to delete t from the output. □

For SJ queries, any tuple in the output has only one witness. If the query involves a join on k relations then the witness has k components. If some component of the witness does not participate in any other witness we have a side-effect-free deletion.

THEOREM 2.4. *The problem of deciding whether there is a side-effect-free deletion or determining a minimum side-effect deletion for a SJ query is poly-time solvable.*

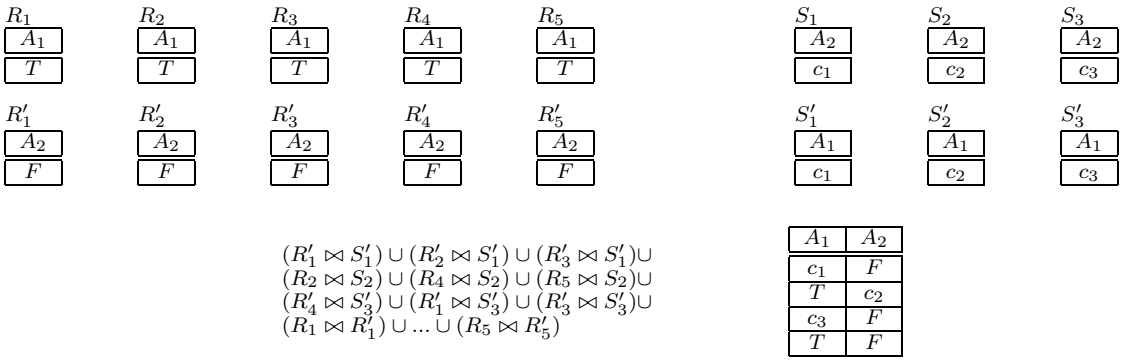

Figure 2: Example reduction in Theorem 2.2.

PROOF. Let t be the output tuple to be deleted. Let $R_1, ..., R_k$ be all the relations that participate in the SJ query. We consider for each $t.R_i$, $i \in [1, k]$ (the R_i component of t) whether there exists another tuple t' in the output such that $t'.R_i = t.R_i$. If yes, then there will be a side-effect if we delete the tuple $t.R_i$ from R_i. If no, this is a side-effect-free deletion. To find the minimum side-effect deletion, we determine i where $i \in [1, k]$ such that $t.R_i$ gives the smallest side-effect. □

2.2 Minimizing Side-effects on the Source

The table below is a summary on the complexity of determining the minimum source deletions for fragments of SPJU queries. Every class of queries is either polytime solvable or is as hard as the set cover problem. In what follows, we will use the *hitting set* problem [6] as the starting point for our hardness reductions. The input to the hitting set problem is same as the set cover problem, but the goal now is to find a smallest subset $X' \subseteq X$ of elements such that $S_i \cap X' \neq \emptyset$ for each set S_i in the collection. The hitting set problem is a dual to the set cover problem and has the same approximability threshold.

Query class	Finding the minimum source deletions
Queries involving PJ	NP-hard
Queries involving JU	NP-hard
SPU	P
SJ	P

2.2.1 Project and Join Queries

THEOREM 2.5. *The problem of minimizing source deletions in order to delete a tuple in a PJ query in normal form is NP-hard. Moreover, it is set cover-hard to approximate.*

PROOF. We give a reduction from the hitting set problem. We are given sets sets S_1,...,S_m such that each $S_i \subseteq \{x_1, ..., x_n\}$. We encode each set as a tuple in the relation $R_0(S, A_1, ..., A_n)$ as follows. For a set $S_i = \{x_{i_1}, x_{i_2}, ..., x_{i_k}\}$, we add a tuple
$(s_i, d, ..., d, x_{i_1}, d, ...d, x_{i_2}, d, ..., d, x_{i_k}, d, ..., d)$ where d is a "dummy" element. In other words, we are encoding the characteristic vector for each set in relation R_0 — if the value of A_i is x_i, it indicates that the set contains the element x_i, and otherwise it is d. In addition to R_0, we have n other relations, R_1,...,R_n. Each of these relations is of the form $R_i(A_i, B_i, C)$ and consists of $n + 1$ tuples: (x_i, α_0, c),(d, α_1, c),...,(d, α_n, c). See Figure 3.

We construct a PJ query as follows: $\Pi_C(R_0 \bowtie R_1 \bowtie ... \bowtie R_n)$. The output of this query is a single attribute tuple (c) and we wish to delete this tuple. Consider the intermediate expression before projection is applied, i.e, the expression $(R_0 \bowtie R_1 \bowtie ... \bowtie R_n)$. Observe that each set S_i will generate $(n)^{n-|S_i|}$ tuples in the intermediate expression. Hence we can "hit" a set S_i, (i.e, destroy all $n^{n-|S_i|}$ intermediate tuples) by either deleting a tuple (x_p, α_0, c) from R_p such that $x_p \in S_i$, or deleting the n tuples (d, α_1, c) through (d, α_n, c) from a relation R_q such that $x_q \notin S_i$.

We now show the equivalence between the minimum hitting sets and minimum source deletions in this encoding. Given a minimum hitting set H, for each $x_p \in H$, we delete the tuple (x_p, α_0, c) from R_p. It is easily verified that this deletes the tuple (c) from the output. On the other hand, consider any solution T that deletes c. We first argue that without any loss of generality, we may assume that T only contains tuples of the form (x_p, α_0, c). Suppose T contains a tuple from R_0 — say, corresponding to some set S_i. Then we can simply replace it by a tuple (x_p, α_0, c) for some $x_p \in S_i$. On the other hand, if T contains tuples of the form (d, α_j, c) from R_q, then it must contain all n such tuples (or else we can just discard them). In this case, we can simply replace them with (x_p, α_0, c) for each element x_p without increasing the cost of the solution. Thus T can be assumed to be in the form claimed above. Since deletion of T deletes (c) from the output, it is easy to see that for each set S_i, T must contain a tuple (x_p, α_0, c) for some element $x_p \in S_i$ and thus corresponds to a hitting set. □

Chain Joins: For PJ queries in normal form if the joins are restricted to be a *chain* join on distinct relations, the problem of minimizing source deletions is solvable opti-

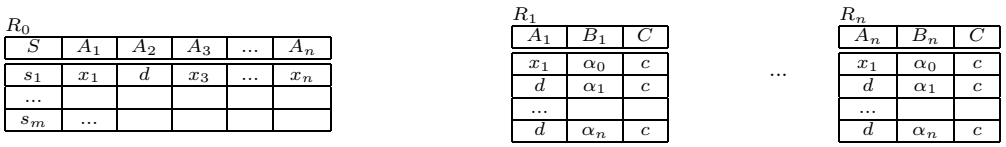

Figure 3: Relations involved in the reduction Theorem 2.5.

mally using flow networks. A join on k distinct relations R_1,...,R_k is called a chain join if the attribute sets of any two relations R_i and R_j are disjoint for $j > i+1$, i.e, only consecutive relations share attributes.

THEOREM 2.6. *For PJ queries in normal form whose joins on distinct relations form a chain, the problem of minimizing source deletions is solvable optimally.*

PROOF. Let $R_1, R_2, ..., R_k$ be the relation in the chain join, and let t_0 be the tuple that we would like to delete from the output. We first eliminate from each R_i any tuples that do not agree with t_0. Now we construct a layered network such that the ith layer of the network corresponds to the remaining tuples in R_i. There is a node for each tuple in R_i in the ith layer, and an edge from a node in the ith layer to one in the $(i+1)$th layer if the tuples agree on the common attributes of R_i and R_{i+1}. We now add two special nodes s and t such that s is connected to all nodes in the first layer and all nodes in the last layer are connected to t. All edges are assigned capacity ∞. We now replace each node v in a layer by two nodes v_i, v_o such that all incoming edges into v go to v_i and all outgoing edges from v leave from v_o. Finally, we add an edge from v_i to v_o of capacity 1. It is easy to see that any $s-t$ path in this network corresponds to a witness for the tuple t_0. Thus in order to delete the tuple t_0 we must delete all such witnesses. It is straightforward to show that an $s-t$ min cut in this graph corresponds to destroying all witnesses for t_0 and vice versa. □

2.2.2 Join and Union Queries

We next show that the hardness for PJ queries continues to hold if we replace projection by union. However, our proof, shown below, relies on renaming (δ) and it remains open if this hardness may be established without using renaming.

THEOREM 2.7. *The problem of minimizing source deletions in order to delete a tuple in a JU query in normal form with renaming is NP-hard. Moreover, it is set cover-hard to approximate.*

PROOF. We once again use a reduction from the hitting set problem. We assume without any loss of generality that each set S_i has the same number k of elements (otherwise, we can pad a set with additional distinct elements). For each element x_i in the universe, there is a relation $R_i(A)$ which consists of a single tuple (a). The query consists of a union of m queries where each query corresponds to a set. If a set S_i consists of elements $x_{i_1}, ..., x_{i_k}$, we write a query $Q_i = \delta_{A \mapsto A_1}(R_{i_1}) \bowtie ... \bowtie \delta_{A \mapsto A_k}(R_{i_k})$. Q is $Q_1 \cup ... \cup Q_m$. The output of Q applied on these relations is a single tuple $(a, a, ..., a)$ and we wish to delete this tuple.

Suppose we are given a hitting set H We first show that a hitting set is a source deletion and vice versa. Given a hitting set, if x_i belongs to the hitting set, we delete the tuple (a) from R_i. It is easy to see that this will delete $(a, a, ..., a)$ from the output since every witness corresponds to conjunction of all elements in a set. Conversely, if we are given a solution to the deletion problem, we can construct a hitting set H by including each element x_i such that tuple (a) from R_i is deleted. It is easy to see that H is a hitting set — otherwise, a witness that corresponds to a set not being hit still survives and this is a contradiction. □

2.2.3 SPU and SJ Queries

THEOREM 2.8. *There is a unique set of source tuples to delete for SPU queries and is poly-time solvable.*

PROOF. The proof is similar to that of Theorem 2.3 which establishes that there is always a unique solution in order to delete an output tuple for an SP query. The problem is poly-time solvable for SP queries and the result follows for SPU queries. It takes linear time (in the size of source relations) to select all tuples that satisfy the select condition of the query and which will have the same projected attributes as the output tuple to be deleted. We must delete all these tuples. □

THEOREM 2.9. *The problem of determining a minimum source deletion for SJ queries is poly-time solvable.*

PROOF. Let t be the output tuple to be deleted. We delete a source tuple $t.R$ for any relation R that participates in the join. $t.R$ denotes the components of the tuple that consists of attributes from relation R. We can determine $t.R$ in time linear to the size of R. □

3. ANNOTATION PLACEMENT

A topic closely related to view update is that of data *provenance* or *lineage* [15, 7]. In data provenance, one is concerned with how one trace the history of some piece of data as it moves through databases. A specific problem here is to ask how a data item is to be traced backwards through a query. In this context, finding a reasonable definition of provenance has been elusive. Most definitions are fragile in that they are sensitive to

query rewriting, and a clear semantic characterization of provenance has yet to emerge. We have found it useful to consider a related issue, that of annotations. As we observed in the introduction, annotations are increasingly being used as a method of scientific communication. We believe that it is useful to ask how annotations are carried through queries and use that as a basis for understanding provenance.

We are going to address some of the computational issues of tracing annotations through queries, but before doing this we should briefly justify the framework in which we are working. Suppose a tuple {`Name: Joe, Age: 41, tel: 1231`} exists in some table and imagine two possible annotations on the number `41` in the `Age` field: (a) "this number is prime" and (b) "this number is too low". (a) is clearly an annotation on the number itself and could, in principle, propagate to any occurrence of `41` in this or other databases. Thus the propagation of this annotation has nothing to do with the query that produced this tuple. However (b) is a statement about the contents of the `Age` field of the tuple and should *not* propagate to every other occurrence of value `41` in the database. We are concerned with annotations of the second type.

We assume that annotations propagate between *locations* in the database, and define a location as a triple (R, t, A), which refers to attribute A of a tuple t of relation R. We say that an annotation is propagated *forward* if it is propagated from the source to view. We consider the following set of *forward propagation rules* for each monotone relational operator. These determine how an annotation is carried from source to view under a relational query.

- **Selection.** If $t \in \sigma_C(R)$ then an annotation on (R, t', A) propagates to $(\sigma_C(R), t, A)$ if $t = t'$. C is the selection condition.
- **Projection.** If $t \in \Pi_{\overrightarrow{B}}(R)$ then an annotation on (R, t', A) propagates to $(\Pi_{\overrightarrow{B}}(R), t, A)$ if $A \in \overrightarrow{B}$ and $t'.\overrightarrow{B} = t$ where $t'.\overrightarrow{B}$ stands for $\overrightarrow{B}$ components in t'. $\overrightarrow{B}$ is the set of projected attributes.
- **Join.** If $t \in (R_1 \bowtie R_2)$ then an annotation on (R_1, t_1, A) (or (R_2, t_2, A)) propagates to $(R_1 \bowtie R_2, t, A)$ if $t.R_1 = t_1$ (or $t.R_2 = t_2$).
- **Union.** If $t \in (R_1 \cup R_2)$ then an annotation on (R_1, t_1, A) (or (R_2, t_2, A)) propagates to $(R_1 \cup R_2, t, A)$ if $t = t_1$ (or $t = t_2$).
- **Renaming.** If $t \in \delta_\theta(R)$ then an annotation on (R, t, A) propagates to $(\delta_\theta(R), t', \theta(A))$ if $t' = t$. θ is a mapping on attributes of R to attributes.

It should be noted that we have used "equality of similarly named fields" as the reason for forward propagation. However explicit equality is not used. For example (R, t', A) does *not* propagate to $(\sigma_{A=A'}(R), t, A')$ when $t = t'$. As a result of this certain rewrites do not preserve annotation propagation, e.g. $\Pi_{ACD}(\sigma_{A=B}(R \bowtie S))$ and $R \bowtie \delta_{B \to A}(S)$ on tables $R(A, C)$ and $S(B, D)$ are equivalent but do not preserve annotation propagation. There are other marking schemes for annotations that one might consider such as flagging variables in a datalog program or adding directives to SQL. These are not the topic of this paper. We consider an annotation placement problem that we believe will be an issue for any reasonable propagation scheme. In what follows, we shall make use of a normal form for SPJRU queries. Let $\mathcal{R}(Q, S)$ be the relation between locations in S and locations in $Q(S)$ induced by the rules given above.

THEOREM 3.1. *There is a normal form of PSJRU queries that preserves $\mathcal{R}$.*

The proof is straightforward and is omitted. It should also be noted that as a consequence of the given propagation rules, constants defined in the view do not carry annotations from any source location. For example, if $V = \{A : a\} \bowtie R$ on the table $R(C, D)$, (V, t, A) will not carry any annotation for any tuple t in the output. We will assume that our queries do not contain such constants.

3.1 Minimizing Annotation Side-effects

We can now formalize our *annotation placement problem*: Given a source database S, a query Q, the view $Q(S)$ generated by Q, and a location in $Q(S)$, find a location to annotate in the source such that annotating it propagates the annotation to a smallest number of locations including the specified view location. We briefly contrast the annotation placement problem with the deletion problem. In the latter, in order to delete a tuple we may have to delete multiple tuples from the source. In contrast, in the annotation placement problem, the optimal solution is always a single location in the view (hence we do not need to consider the problem of minimizing the number of source annotations needed). However, as we will shortly see, the problem continues to be intractable for PJ queries. But the class of JU queries now becomes polynomial time solvable.

Query class	Deciding whether there is a side-effect-free annotation
Queries involving PJ	NP-hard
SJU	P
SPU	P

3.1.1 Project and Join Queries

For queries involving both project and join operators, we show that it is NP-hard to determine if there is a source location that will give a side-effect-free annotation.

THEOREM 3.2. *The problem of deciding whether there is a side-effect-free annotation for a PJ query in normal form is NP-hard.*

PROOF. The proof is by reduction from 3SAT. Suppose the 3SAT formula has m clauses. For each clause

C_i consisting of variables x_{i_1}, x_{i_2} and x_{i_3}, we have a relation $R_i(C_i, x_{i_1}, x_{i_2}, x_{i_3})$. Each R_i includes seven "assignment" tuples, each of which corresponds to a possible satisfying assignment for the clause C_i. For example, if $C_i = (\overline{x_{i_1}} + x_{i_2} + \overline{x_{i_3}})$ then there is a relation $R_i(C_i, x_{i_1}, x_{i_2}, x_{i_3})$ consisting of seven tuples:
(c_i, F, F, F), (c_i, F, F, T), (c_i, F, T, F), (c_i, T, F, F), (c_i, F, T, T), (c_i, T, T, F), (c_i, T, T, T). In addition, each of the relation, R_1,..., R_{m-1}, has an additional "dummy" tuple (c_i, d, d, d) where d is a "dummy" element and R_m has two additional tuples (c_m, d, d, d) and (c'_m, d, d, d). We write the query $Q = \Pi_{C_1,...,C_m}(R_1 \bowtie ... \bowtie R_m)$. The output of this query consists of 2 tuples $(c_1, ..., c_m)$ and $(c_1, ..., c'_m)$. It is easy to see that the 3SAT formula is satisfiable iff there are m assignment tuples from $R_1, ..., R_m$, corresponding to the satisfying assignment, that can join successfully to produce the tuple $(c_1, ..., c_m)$ in the output.

Now suppose we are asked to annotate on the first component of the first output tuple, i.e, the location $(Q(S), (c_1, ..., c_m), C_1)$. There are two possible solutions — annotate either one of the assignment tuples in R_1 or annotate the dummy tuple. If the formula is satisfiable, annotating (R_1, t, C_1) where $t = (c1, ...)$ corresponds to the satisfying assignment is a feasible solution that does not annotate the second tuple in the output. On the other hand, suppose we are given a solution with no side-effects. We claim that it must correspond to annotating one of the assignment tuples or else the second tuple in the output would get annotated as well. Thus the 3SAT formula must be satisfiable. □

An interesting corollary of the above proof is that computing either kind of provenance information for an output tuple is NP-hard even for this restricted subclass of queries.

COROLLARY 3.1. *Given a source database S, a PJ query Q in normal form, its output $Q(S)$ and a tuple $t \in Q(S)$, it is NP-hard to determine if a given tuple $t' \in S$ is part of a witness for t. Moreover, it is also NP-hard to determine if an annotation on a source location appears in the output.*

3.1.2 SPU and SJU Queries

For SPU queries, it is easy to determine the source location that will give a side-effect-free annotation. For SJU queries, one can determine the source location that will give the minimum side-effects. The proofs are shown in the appendix.

THEOREM 3.3. *Given an SPU query, the annotation placement problem is solvable in polynomial time.*

PROOF. We will show that there is always a location in the source which produces an annotation on the desired view location without any side-effects. We first consider SP queries. Given an attribute A of an output tuple t which we wish to annotate, we scan the input relation until we find the tuple t' which satisfies the selection condition and whose projected attributes equal t. Annotate on attribute A of t' and it is an easy consequence of our propagation rules that only the desired view location receives the annotation. For SPU queries, we consider each SP query fragment and we apply the same procedure for each SP query till we find a tuple t' as above and annotate it. Once again, only the desired location receives the annotation. □

THEOREM 3.4. *Given an SJU query in normal form, the annotation placement problem is solvable in polynomial time.*

PROOF. We first show how one can find the source location to annotate with minimum side-effect for a single SJ query. Let Q denote the SJ query and suppose we wish to annotate on the location $(Q(S), t, A)$. Suppose A exists in relations $R_1, ..., R_k$ which participate in Q. We check for each $t.R_i$, $i \in [1, k]$, the number of other tuples t' in $Q(S)$ such that $t'.R_i = t.R_i$. We can determine $i \in [1, k]$ such that annotating on location $(R_i, t.R_i, A)$ gives the smallest side-effect. Now in order to do this for an SJU query, we do the same for each subquery except we now also check for the additional locations that would receive annotations through other queries in the union. We choose the location with minimum side-effects. In other words, suppose A exists in relations $R_{i1}, ..., R_{ik_i}$ which participates in an SJ query Q_i. We check for each $t.R_{ij}$, $j \in [1, k_i]$, the number of other tuples t' in $Q(S)$ such that $t'.R_{ij} = t.R_{ij}$ and $t' \in Q_i(S)$. In addition, we also check for every other SJ query where R_{ij} occurs, the number of other locations that gets annotated as well according to that SJ query. We choose the location with minimum side-effects. □

4. CONCLUSIONS

We establish a dichotomy in the complexity for both view side-effect and source side-effect minimization problems. Interestingly, the tractable and the intractable query classes are identical for both problems.

Our study of annotations introduces a new model for provenance, related to "where-provenance" [7]. One of the interesting findings of this work is that it is difficult for annotation propagation to be invariant under equivalent queries in general, i.e, equivalent queries may not carry annotations in the same way. However we can identify a set of rewritings, which gives rise to a normal form and which preserves the relationships between annotations on the input and output. It will be interesting to study other models of propagating annotations, based, for example, on datalog, SQL and non-relational query languages.

Our study of these problems also reveals that computing both why-provenance and where-provenance is intractable for monotone relational algebra. This suggests that additional constraints, such as key or foriegn key constraints, will be necessary for efficiently tracking provenance information.

5. REFERENCES

[1] A. M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections and Joins. In *Principles of Database Systems*, 1985.

[2] A. M. Keller. Choosing a view update translator by dialog at view definition time. In *Int'l Conf. on Very Large Data Bases*, 1986.

[3] D. Maier and L. Delcambre. Superimposed Information for the Internet. In *Proceedings of the Int'l Workshop on Web and Databases (WebDB)*, pages 1–9, 1999.

[4] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems*, 6(4):567–575, 1981.

[5] E. M. Gold. Complexity of automatic identification of given data, 1974. Unpublished manuscript.

[6] M.R. Garey and D.S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[7] P. Buneman and S. Khanna and W. Tan. Why and Where: A Characterization of Data Provenance. In *Int'l Conf. on Database Theory*, 2001.

[8] S.S. Cosmadakis and C. H. Papadimitrou. Updates of Relational Views. In *Principles of Database Systems*, 1983.

[9] Lincoln Stein. *Distributed Annotation Server*. `http://biodas.org`.

[10] T. J. Schaefer. The complexity of satisfiability problems. In *Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226, 1978.

[11] U. Dayal and P.A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Database Systems*, 8(3):381–416, 1982.

[12] U. Feige. A threshold of ln n for approximating set cover. *Journal of ACM*, 45(4):634–652, 1998.

[13] W3C. *Annotea Project*. http://www.w3.org/2001/Annotea.

[14] Y. Cui and J. Widom. Run-Time Translation of View Tuple Deletions Using Data Lineage. Technical report, Stanford University, 2001. http://dbpubs.stanford.edu:8090/pub/2001-24.

[15] Y. Cui and J. Widom and J.L. Wiener. Tracing the Lineage of View Data in a Data Warehousing Environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.