On the Complexity of View Update Analysis and Its Application to Annotation Propagation

On the Complexity of Annotation Propagation and View Update Analyses

Gao Cong Wenfei Fan Floris Geerts Jianzhong Li Jizhou Luo

Abstract—This paper investigates three problems identified in [1] for annotation propagation, namely, the *view side-effect*, *source side-effect* and *annotation placement problems*. Given annotations entered for a tuple or an attribute in a view, these problems ask what tuples or attributes in the source have to be annotated to produce the view annotations. As observed in [1], these problems are fundamental not only for data provenance but also for the management of view updates. For an annotation attached to a *single existing* tuple in a view, it has been shown that these problems are often intractable even for views defined in terms of simple SPJU queries [1]. We revisit these problems by considering several dichotomies: (a) views defined in various subclasses of SPJU, versus SPJU views under a practical *key preserving* condition; (b) annotations attached to existing tuples in a view versus annotations on tuples to be inserted into the view; and (c) a single-tuple annotation versus a group of annotations. We provide a complete picture of intractability and tractability for the three problems in all these settings. We show that key preserving views often simplify the propagation analysis. Indeed, some problems become tractable for certain key preserving views, as opposed to the intractability of their counterparts that are not key preserving. However, group annotations often make the analysis harder. In addition, the problems have quite diverse complexity when annotations are attached to existing tuples in a view and when they are entered for tuples to be inserted into the view.

Index Terms—Annotation, View updates, View maintenance, SPJU queries.

1 Introduction

Database annotations have been recognized by scientists as an essential feature for new generation database management systems [2], [3], [4]. Annotations are additional information attached to tuples or attributes, either entered manually or generated by programs, to explain or correct the data [2]. This information is essential to the quality and semantics of the data, and should be carried over along with the regular data when the data is migrated, transformed or integrated. With this comes the need for studying annotation propagation. The analysis of annotation propagation is important in tracing the origin of the data [2], [1], [5], [6], [7], [8], [9] (*a.k.a.* lineage [10], [11]), data cleaning [12], access control [13], semantic Web [14], and in digital libraries [15], among other things. Several systems and tools have been developed to support annotation propagation analysis, *e.g.,* DBNotes [16], MONDRIAN [17] and InfoVis [18].

Annotation propagation analysis. In many applications data transformations are expressed as views defined as SPJU queries in terms of the selection (S), projection (P), join (J), union (U) and renaming operators of the relational algebra. Annotations attached to some tuples in a database are carried forward to the views: the selection and projection operators preserve the annotations placed at selected tuples and the projected attributes, respectively; join merges annotations of the tuples joined together, while union simply copies the annotation of each tuple. In addition, annotating a view of some data, the annotations are carried backward to the source data as well as forward to other views [2]. As observed by [1], annotation propagation analysis is closely related to classical view update problem (see, *e.g.,* [19] for a detailed discussion about the view update problem).

Three problems fundamental to the propagation analysis have been identified in [1], stated as follows. Consider a source database D, an SPJU query Q, the view $Q(D)$ and a tuple $\triangle V$ in the view, given as input.

- The *view side-effect problem* is to find a *smallest* set $\triangle D$ of tuples in D such that $Q(D) \setminus \triangle V = Q(D \setminus \triangle D)$, *i.e.,* with zero side effect, if such $\triangle D$ exists.
- The *source side-effect problem* is to find a smallest set $\triangle D$ of tuples in D such that $\triangle V$ is in $Q(D) \setminus Q(D \setminus \triangle D)$.
- The *annotation placement problem* is to find, given a field in the tuple $\triangle V$, a single tuple $\triangle D$ in D such that an annotation in a field of $\triangle D$ propagates to the minimum number of fields in the view including $\triangle V$.

Intuitively, when an annotation is entered for $\triangle V$ in the view, the view side-effect problem is to identify a smallest set $\triangle D$ of tuples in the source such that annotations in those places produce the view annotation, without spreading to other view tuples. Alternatively, it means that the deletion of $\triangle D$ from the source leads to the removal of $\triangle V$ from the view without side effect, *i.e.,* $\triangle D$ indicates how the tuple $\triangle V$ gets into the view.

We should remark that our statement of the view side-effect problem is referred as the "side-effect free" view

- *G. Cong is with the Department of Computer Science, Aalborg University, Denmark.*
- *W. Fan and F. Geerts are with the School of Informatics, University of Edinburgh, Edinburgh, U.K.*
- *J. Li and J. Luo are with the Department of Computer Science and Technology, Harbin Institute of Technology, Heilongjiang, China.*

AuName	*Journal*
Joe	TKDE
John	TKDE
Tom	TKDE
John	TODS

(a) Author(*AuName*, *Journal*).

Journal	*Topic*	*#Papers*
TKDE	XML	30
TKDE	CUBE	30
TODS	XML	30

(b) Journal(*Journal*, *Topic*, *#Papers*).

	AuName	*Topic*
	Joe	CUBE
	Joe	XML
	Tom	CUBE
	Tom	XML
	John	CUBE
$\triangle V$	John	XML

(c) View definition Q_1: $\pi_{AuName,Topic}(\text{Author} \bowtie \text{Journal})$.

AuName	*Journal*	*Topic*
Joe	TKDE	CUBE
Joe	TKDE	XML
Tom	TKDE	CUBE
Tom	TKDE	XML
John	TKDE	CUBE
John	TKDE	XML
John	TODS	XML

(d) Key preserving view: $\pi_{AuName,Journal,Topic}(\text{Author} \bowtie \text{Journal})$.

Fig. 1. Example of propagation problems.

side-effect problem in [1]. Here we enforce $|Q(D) \backslash Q(D \backslash \triangle D)| = 0$. In contrast, [1] aims to minimize $|Q(D) \backslash Q(D \backslash \triangle D)|$, although all the complexity results of [1] for the view side-effect problem are established for the "side-effect free" version, *i.e.,* the problem studied here.

As opposed to the view side-effect problem, the source side-effect problem is to find a smallest set of tuples in the source such that the desired annotation in the view can be obtained by annotating those places in the source, although it may have side effects on the view. Note that the source side-effect problem does not require $|\nabla V| = 0$.

When some annotation is attached to a location in $\triangle V$, the annotation placement problem is to find the corresponding location in the source D to concretely annotate such that the view annotation propagates backward to the source with minimum side effects.

Example 1.1: Consider a database D with two relations: Author(*AuName*, *Journal*), Journal(*Journal*, *Topic*, *#Papers*) (with keys underlined), and an SPJ query (view definition) $Q_1 = \pi_{AuName,Topic}(\text{Author} \bowtie \text{Journal})$. Instances of both relations and the view $Q_1(D)$ are shown in Figures 1(a)–(c) (ignore Fig.1(d) for now).

Suppose that John is not a researcher on XML and thus the tuple (John, XML) in the view $Q_1(D)$ is an error. Let $\triangle V = \{(\text{John, XML})\}$. We want to find tuples $\triangle D$ in the base relations of D to annotate the error such that the annotations propagate to the fields in the view tuple $\triangle V$ via Q_1; or alternatively we want to delete $\triangle D$ such that their removal leads to the deletion of the erroneous $\triangle V$. The three problems stated above impose different conditions on how to achieve this.

(1) View side-effect problem: There are multiple ways to remove tuples in D in order to delete $\triangle V$ from the view. The tuples in D related to $\triangle V$, *i.e.,* those with matching values in $\triangle V$, are (John, TKDE), (John, TODS), (TKDE, XML, 30) and (TODS, XML, 30). We want to find a smallest set $\triangle D$ of tuples such that deleting $\triangle D$ from D leads to the removal of $\triangle V$ from $Q_1(D)$ but incurs no side effects, *i.e.,* it deletes $\triangle V$ but no other tuples from $Q_1(D)$. To delete $\triangle V$ one can remove {(John, TKDE), (John, TODS)} from the Author table (denoted by $\triangle_1 D$), or delete (John, TKDE) from Author and (TODS, XML, 30) from Journal ($\triangle_2 D$). However, none of these is side-effect free: the first option, for example, also results in the deletion of (John, CUBE) from the view. Hence there exists *no* solution to the view side-effect problem.

(2) Source side-effect problem: It differs from (1) in that we do not care about the view side effect when we search for a smallest set $\triangle D$ of tuples in D to delete. Thus in this case, both $\triangle_1 D$ and $\triangle_2 D$ are solutions. In addition, removing {(TODS, XML, 30), (TKDE, XML, 30)} from Journal is also a solution although it incurs more severe view side effects than $\triangle_1 D$ and $\triangle_2 D$.

(3) Annotation placement problem: Suppose that the information "John is not an XML researcher" is attached to the *AuName* field of $\triangle V$. We want to find a single tuple $\triangle D$ in the database D to annotate such that the annotation propagates to the *AuName* field of $\triangle V$ and a least number of other fields in the view $Q_1(D)$. Here the solution is $\triangle D =$ (John, TODS): by annotating the *AuName* field of $\triangle D$ we get the desired annotation in the view with zero side effect. □

View updates and data provenance. The need for investigating these problems is evident in view update management and data provenance. As observed by [2], the view and source side-effect problems are the classical view deletion problems. Moreover, these two problems are important to why-provenance, while the annotation placement problem is related to where-provenance [5].

• *The connection with why-provenance*. Given an annotation attached to some tuple t in the output of a query, the view and source side-effect problems are to find which tuples in the input should be annotated such that the annotations in the input are propagated forward to the view. To answer these questions, we need to identify a (smallest) set of the input tuples that suffices to make t appear in the view. This is what why-provenance concerns, which aims to find a "proof" or "witness" for t to appear in the output, *i.e.,* a minimum set of source tuples that suffices to produce t in the output.

• *The connection with where-provenance*. When an annotation a is attached to some field (location) l of a tuple in the view, the annotation placement problem is to find a single field (location) in the input to place the annotation a, such that a propagates to a minimum number of output locations including l. That is, we want to propagate annotations backwards from the output of a query to the source database [3]. In other words, we want to identify where the value in the output location l is copied from. This is the focus of where-provenance, which is to find where a value in the output comes from.

As will be discussed shortly, there has been a host of work on annotation processing [1], [2], [4], [5], [6], [7],

[8], [9], [10], [11], [13], [16], [17], [20], [21], [22], [23], [24], [25], [26], [27]. However, the complexity bounds for the problems described above are only studied in [1], [13]. In those papers it is shown that the analysis is in general beyond reach in practice. Indeed, the view and source side-effect problems are NP-hard for views expressed in fragments of SPJU. Similarly, the annotation placement problem is NP-hard for PJ and SPJ views [13]. While these problems are also important to the management of view updates, their complexity bounds have not been studied in that line of work.

Contributions. In this paper, we extend [1], [13] in several aspects. First, we identify a practical condition under which the analysis of annotation propagation becomes feasible. The condition, referred to as the *key preservation* condition, requires that an SPJU view Q retains a key of every base relation involved in the definition of Q. In other words, a view Q is key preserving if the primary keys of all the base relations involved in Q are included as distinct attributes in the projection fields of Q. This is less restrictive than other proposals for restricting view definitions [28], [29]. Furthermore, many views for data transformation or integration found in practice can be naturally modified to be key preserving by extending the projection-attribute list to include the primary keys.

Second, we investigate the impact of *group updates* on the analysis of annotation propagation. That is, we generalize the problem statements given earlier by allowing the given view update $\triangle V$ to include multiple tuples. The need for studying this is evident: in practice annotations are often entered for multiple view tuples at the same time, rather than for a single tuple.

Third, in addition to annotations attached to existing tuples in a view, we study the view and source side-effect problems when the given $\triangle V$ is a set of tuples to be *inserted*. These are the classical view insertion problems. The motivation for studying this is that one often wants to know, when new tuples along with annotations are inserted into the view, how the annotations should be propagated back to the source (*a.k.a.* feedback loop [12]). We study these problems both in the presence and in the absence of the key preservation condition.

We give a full treatment of the three problems for annotation propagation *w.r.t.* the following dichotomies:

- general views vs. key preserving views,
- singleton $\triangle V$ vs. a set $\triangle V$ of view tuples, and
- $\triangle V$ to be deleted vs. $\triangle V$ to be inserted.

We examine the impact of different combinations of these factors on the complexity of these problems.

We provide a comprehensive picture of *the combined complexity* on these problems for views defined in various fragments of SPJU queries, identifying all those cases that are intractable. The results tell us the following.

(1) Key preserving views often simplify the analysis of annotation propagation. For instance, the annotation placement problem is NP-hard for general PJ (with projection and join) views [13], but it is in polynomial time (PTIME) for key preserving SPJU views. When $\triangle V$ consists of a single existing tuple in the view, the source side-effect problem is NP-hard for general PJ views [1], but it is in PTIME for key preserving SPJ views. This tells us that key preserving views make it feasible to efficiently conduct certain propagation analysis.

(2) Group updates complicate the propagation analysis. For instance, the view side-effect and source side-effect problems become NP-hard for key preserving SPJ views when group deletions are considered, whereas they are in PTIME for single-tuple deletions.

(3) The presence of selections in the views does not complicate the analysis. More specifically, the complexity of all problems is independent of the presence of selection predicates in the view definition.

(4) These problems have quite diverse complexity for view insertions and view deletions. On one hand, the view side-effect problem is in PTIME for key preserving SPU and SPJ views and single-tuple deletions, whereas it is intractable for single-tuple insertions and views defined with join only. On the other hand, the source side-effect problem is in PTIME for key preserving SPJU and single-tuple insertions, but it becomes NP-hard for JU views and single-tuple deletions.

Taken together, these tell us what cases of the annotation propagation analysis are intractable or in PTIME, for all subclasses of SPJU views, from general views to key-preserving views, and from single-tuple update to group view updates. To our knowledge, no previous work has established complexity results for these problems for key preserving views, group view updates, or for view insertions. These results are useful in both the analysis of data provenance and the study of view update management.

Related work. This paper extends an earlier version [21] as follows: (1) we investigate the annotation analysis for key preserving views that also support the union operator, and (2) we revise the statement of the view side-effect problem used in [21] to align with its counterpart of [1], and redeveloped the results accordingly. Several new intractability and PTIME results are established. In particular, we show that the view side-effect problem for insertions and views defined with join alone is already NP-hard, and that the annotation placement problem is tractable for key-preserving SPJU views.

Recent research on querying annotated databases can be classified into two categories: *annotation querying* [6], [16], [17], [20], [22], [23] and *annotation propagation* [1], [7], [9], [21], [24], [25], [26]. In the former, queries access annotations as well as the regular data directly. In the latter, queries are directed primarily at the regular data, while annotations are merely carried to the query results.

Annotation querying has focused on (a) propagation schemes for processing annotations explicitly or implicitly [9], [16], (b) the expressive power of various propagation schemes [22], [24], and (c) their extensions to deal

with complex queries [6], [7], [23], [30]. The problems studied there are entirely different from the problems considered in this work.

For annotation propagation, the only previous complexity results were established in [1], [13]. Key preservation, group updates and propagation of view insertions were not considered in [1], [13].

There has also been work on modeling and managing provenance information [8], [10], [27], [31], [32], [33], [34], [35], [36], in which only [10] gave a complexity result. In [10], a key-preserving condition was also considered. It was shown there that the condition simplifies the computation of lineage. However, [10] studied generic mapping functions, which are quite different from SPJU views. Hence their complexity results do not apply to the problems considered in this paper and vice versa. The key preservation condition was also studied in [37] for XML view updates. The problems investigated in [37] are quite different from those considered in this work.

An algorithm was given in [11] for translating view deletions to base relations without side effects, based on data lineage. It performs an exhaustive search over all candidate solutions, which takes exponential time. In contrast, with our key-preservation condition, the computation of data lineage is simplified and the view side-effect deletion problem is PTIME resolvable.

There has been a host of work on view updates (*e.g.*, [38], [39], [28], [29], [35]). Algorithms were provided in [28] for translating restricted view updates to base-table updates without side effects in the presence of certain functional dependencies. An algorithm was developed in [29] to translate (with side effects) a class of SPJ view updates to base relations, with the following restrictions: base tables may only be joined on keys and must satisfy foreign keys; a join view corresponds to a single tree where each node refers to a relation; join attributes must be preserved; and comparisons between two attributes are not allowed in selection conditions. Our key preservation condition (to be defined in Section 2) is less restrictive than those in [28], [29]. More recently in [35], a bi-directional query language was proposed, which imposes conditions on the operators in the language such that arbitrary changes to views can be carried out. The conditions are more restrictive than the key preservation condition studied in this paper.

On relational view updates, surprisingly few complexity bounds are known. The only tractability and intractability results we are aware of were established in [38], [39], [40], for finding a minimal view complement for relational views, a problem very different from ours.

Commercial database systems [41], [42], [43] allow updates on very restricted views, while allowing users to specify updates manually with the INSTEAD OF triggers. For example, for views to be deletable IBM DB2 [41] restricts the from clause to reference only one base table.

Organization. We first present key preserving views in Section 2. We then establish the complexity bounds of the view side-effect problem, the source side-effect problem and the annotation placement problem in Sections 3, 4 and 5, respectively. We identify open issues in Section 6.

2 KEY PRESERVATION

In this section we define the notion of key preservation.

SPJU queries. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a relational schema. An SPJ query (*a.k.a.* conjunctive query) on databases of $\mathcal{R}$ is a query defined in terms of the selection (σ), projection (π), join ($\bowtie$) and renaming (δ) operators in the relational algebra. It can be expressed in the normal form as follows [19]:

$$\pi_Y(R_c \times E_s), \text{ where } E_s = \sigma_F(E_c),\ E_c = S_1 \bowtie \cdots \bowtie S_n,$$

where (a) Y is a list of attributes in relations of $\mathcal{R}$; (b) $R_c = \{(A_1 : a_1, \ldots, A_m : a_m)\}$, a constant relation, such that for each $i \in [1, m]$, A_i is in Y, A_i's are distinct, and a_i is a constant in the domain of A_i; (c) for each $j \in [1, n]$, S_j is $\rho_j(R_i)$ for some R_i in $\mathcal{R}$, and ρ_j is a renaming operator, such that A_i does not appear in any S_j; and (d) F is a conjunction of equality atoms such as $A = B$ and $A =$ 'a' for a constant a in the domain of A.

An SPJU query (*a.k.a.* union of conjunctive queries) defined on $\mathcal{R}$ is a query of the form $Q_1 \cup \cdots \cup Q_n$, where Q_i's are union-compatible SPJ queries on $\mathcal{R}$ [19].

We study various subclasses of SPJU, denoted by listing the operators supported. The renaming operator is included in all subclasses by default without listing it explicitly. For instance, PJ is the class of queries defined with the projection, join and renaming operators.

For example, the view given in Fig. 1(c) is a PJ view.

Key preserving views. Consider an SPJ query $Q = \pi_Y(R_c \times E_s)$ defined above. We say that Q is *key preserving* if all the primary key attributes (with possible renaming) of each occurrence of the base relations in E_s are included in the projection fields Y of Q.

An SPJU query $Q_1 \cup \cdots \cup Q_n$ is said to be *key preserving* if for each $i \in [1, n]$, Q_i is key preserving.

Example 2.1: The query Q_1 given in Example 1.1 (and corresponding view shown in Fig. 1(c)) can be extended such that it is key preserving. Indeed, let $Q_2 = \pi_{AuName,Journal,Topic}(\text{Author} \bowtie \text{Journal})$. Then Q_2 is key preserving. The view $Q_2(D)$ is shown in Fig. 1(d). □

The analysis of Example 1.1 becomes simpler for the key preserving view of Example 2.1. Consider the deletion of $\triangle V = \{$(John, TKDE, XML)$\}$ from $Q_2(D)$. (1) View side-effect problem. Since Q_2 is key preserving, it is obvious that the deletion can be performed by deleting either (John, TKDE) from Author or (TKDE, XML, 30) from Journal. Leveraging key preservation we can easily check the view side effect by finding the occurrences of key values of deleted relation tuples in the view. This tells us that there is no solution to the problem that has zero view side effect. (2) Source side-effect problem. Similar to (1), we can easily determine

that the solution is either {(John, TKDE)} or {(TKDE, XML, 30)}. (3) Annotation placement problem. Similarly we can see that the solution is {(John, TKDE)}. □

3 The View Side-Effect Problem

In this section we investigate the view side-effect problem. We first study the problem for single-tuple and *group deletions* in Section 3.1. We then investigate the problem for *insertions* in Section 3.2, for key preserving SPJU views and general SPJU views.

3.1 Deletion Propagation

Given a view deletion $\triangle V$, a source database D, an SPJU query Q and the view $Q(D)$, the view side-effect problem for deletion propagation is to find a smallest set of source tuples $\triangle D$ to delete such that tuples in $\triangle V$ are deleted, without side effects.

The view side-effect problem has been studied in [1] for general SPJU views and single-tuple deletions (when $\triangle V$ is a singleton set). We investigate this problem for key preserving SPJU views and single-tuple deletions, and for SPJU views (key preserving or not) and group deletions (when $\triangle V$ consists of multiple tuples).

We present in Table 1 the complexity of the view side-effect problem for various subclasses of SPJU, for single-tuple deletions and for group deletions. In Table 1 and all other tables in the paper, we omit the selection predicate S since it does not affect our complexity results.

Single-tuple deletions. It is known that without the key preservation condition the view side-effect problem for single deletions on a PJ view is NP-hard [1]. In contrast, the problem becomes tractable for key preserving SPJ views. This shows that key preservation may indeed simplify the analysis of annotation propagation.

Proposition 1. *The view side-effect problem is in PTIME for key preserving SPJ views and single tuple deletions.*

Proof: Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a relational schema, Q a key preserving SPJ query, D an instance of $\mathcal{R}$, and $\triangle V$ consist of a single tuple t to be deleted from the view $Q(D)$. Given D, Q, $Q(D)$ and $\triangle V$, we show that it is in PTIME to find $\triangle D$ with zero side effects, if it exists.

Since Q is key preserving, we can associate with t (necessarily unique) tuples s_i in the base relations R_i appearing in Q, such that s_i and t have the same key for this relation. In order to delete t from $Q(D)$, it suffices to delete a single such s_i from its base relation R_i. Any such deletion obviously results in an update $\triangle D$ of minimum size since $\triangle D$ consists of a single tuple only, as illustrated in Example 2.1.

For $\triangle D$ to be a solution for the view side-effect problem, we need to find tuple s_i without any side effects. Let S_i be the set of tuples in $Q(D) \setminus \{t\}$ carrying the key of s_i (in R_i). Note that computing S_i requires only a linear scan over the view $Q(D)$. Clearly, the size of S_i determines the number of side effects obtained when choosing $\triangle D$ to be $\{s_i\}$. Let s be the tuple s_i such that its corresponding S_i is of minimum size. Clearly, s can be found in PTIME. Moreover, if $|S_i| = 0$ then $\triangle D = \{s\}$ is a solution to the view side-effect problem. □

However, key preservation does not make our lives easier for JU views. From the proof of [1] for JU views it follows that the problem remains NP-hard for key preserving JU views.

Corollary 2. *The view side-effect problem is NP-hard for key preserving JU views and single tuple deletions.*

Proof: The proof of Theorem 2.2 in [1] is applicable here. The proof shows that the view side-effect problem is NP-hard for single tuple deletions and JU views by reduction from the 3SAT problem. The reduction, however, uses JU views that are key preserving. □

Group deletions. Our first result for group deletions is a PTIME algorithm for the view side-effect problem for SPU views, which extends the algorithm given in the proof of Proposition 1. It should be remarked that the complexity of group view deletions is considered in neither [1] nor [13].

Corollary 3. *The view side-effect problem is in PTIME for* SPU *views and for group deletions.*

Proof: Let $\mathcal{R}$ be a schema and D database as described in Proposition 1, $Q = \bigcup_{j=1}^{k} Q_j$ a union of SP queries, and $\triangle V = \{t_1, \ldots, t_m\}$ be a group deletion. We give a PTIME algorithm for computing $\triangle D$ that is side-effect free, if it exists. The algorithm is an extension of the algorithm for single deletions developed in [1].

The algorithm first scans D and returns for each $R_i \in \mathcal{R}$ the set of tuples satisfying at least one of the selection conditions in one of the SP queries Q_j. We denote the resulting database by D'_0. Next, the algorithm considers the tuples in $\triangle V$ and removes them from $\triangle V$ if a side-effect free update has been found for the tuples considered so far. That is, at each step a set $\triangle D'$ is computed, if it exists, such that $Q(D \setminus \triangle D') = Q(D) \setminus \triangle V'$, where $\triangle V'$ denotes the tuples in $\triangle V$ removed so far. It is important to remark that for SPU views the set $\triangle D'$, if it exists, is uniquely determined.

Initially, $D' = D'_0$, $\triangle V' = \emptyset$ and $\triangle D = \emptyset$. As long as $\triangle V \setminus \triangle V' \neq \emptyset$, let t be the first tuple (based on some arbitrary ordering) in $\triangle V \setminus \triangle V'$. The algorithm then computes $\triangle D'$ as the set of all tuples from D' that project on t. Observe that this set is indeed uniquely determined. We distinguish between the following two cases: (a) there exists a tuple $s \in \triangle D'$ such that $Q(\{s\}) \not\subseteq \triangle V$. In this case, the algorithm halts and no side-effect free solution exists. (b) $Q(\triangle D') \subseteq \triangle V$. In this case, the deletion of $\triangle D'$ in D' causes side effects that all belong to $\triangle V$. Clearly, these view tuples do not have to be further processed by the algorithm. Hence, we set $\triangle D = \triangle D \cup \triangle D'$ and call the algorithm with $D' = D' \setminus \triangle D'$ and $\triangle V' = \triangle V' \cup (\triangle V \cap Q(\triangle D'))$. If the algorithm successfully terminates, *i.e.*, when $\triangle V' = \triangle V$,

the side-effect free update $\triangle D$ is returned. The solution $\triangle D$ is minimum because of the uniqueness of each update computed during the execution of the algorithm, and in addition, the need to remove all tuples in these updates that is necessary in any solution. The algorithm clearly runs in polynomial time. □

Group updates may complicate the annotation propagation analysis. In contrast to Proposition 1, the view side-effect problem becomes NP-hard for group deletions and key preserving views defined with join only.

Theorem 4. *The view side-effect problem is NP-hard for key preserving* J *views and group deletions.*

Proof: We show that the problem is NP-hard by reduction from the minimum set cover problem. An instance of the minimum set cover problem consists of a collection C of subsets of a finite set S. It is to find a subset $C' \subseteq C$ such that every element in S belongs to at least one member of C' and moreover, $|C'|$ is minimum. This problem is NP-complete (cf. [44]).

Given S and C, we define an instance of the view side-effect problem such that there exists a minimum side-effect free update for the view side-effect problem if and only if there exists a minimum cover of the set S.

Let $S = \{x_i \mid i \in [1,n]\}$ and $C = \{c_j \mid j \in [1,k]\}$. We construct two base tables R and R_S, a join view Q and a group view deletion $\triangle V$, as follows.

Base relations. We define two base relations R and R_S.

- $R(A)$, where A is the key and is to hold a number in $[1,k]$. Initially, $R(A)$ contains $k = |C|$ tuples $\{(1),(2),\ldots,(k)\}$ that represent the index of k subsets.
- $R_S(j, A_1, \ldots, A_k)$, where the key consists of all the columns. We encode each element in S with tuples in R_S as follows. For each x_i in S, let T_i be the collection of all the subsets in C that contain x_i. We assume w.l.o.g. that $T_i \neq \emptyset$ (otherwise there is no solution for the minimum set cover problem). Enumerate the elements of T_i as $(c_{i_1}, \ldots, c_{i_{n_i}})$. We generate a list of size k from T_i, $L_i = \langle i_1, \ldots, i_{n_i}, \ldots, i_{n_i}\rangle$, by replacing c_{i_j} with its index i_j and appending $(k - |T_i|)$ i_{n_i}'s at the end of the list (to make the size of the list to be k).

If $|S| > k$, then we generate $|S|$ tuples by adding index $i \in [1,|S|]$ at the beginning of each list L_i. Otherwise, we generate $k+1$ tuples by adding index i in $[1,|S|]$ at the beginning of each list L_i and generate $k+1-|S|$ tuples by adding numbers $[|S|+1, k+1]$ to L_n. Thus R_S initially contains $l = \max\{|S|, k+1\}$ tuples.

View. We define a query $Q = R_S \bowtie \delta_{f_1}(R) \bowtie \cdots \bowtie \delta_{f_k}(R)$, where δ_{f_i} renames A in R to A_i. Initially, $Q(D)$ consists of l view tuples, which are the same as those in the relation R_S. Obviously, the view defined as above is key preserving since it contains no projections.

View deletion. The group deletion $\triangle V$ is to remove all tuples in the view $Q(D)$.

The view side-effect problem is to find a smallest set of the tuples from R and R_S so that $\triangle V$ is deleted without side-effect. We include an example toward the end of the proof to illustrate the intuition behind the reduction.

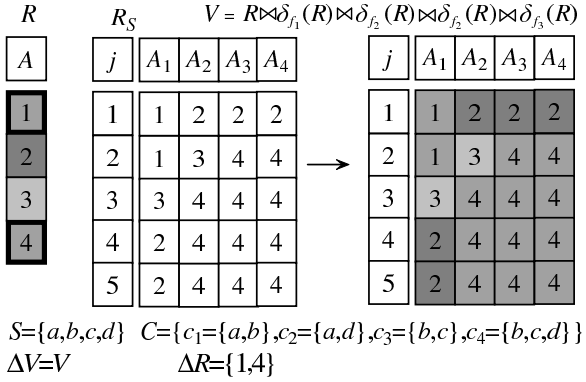

Fig. 2. Illustration of the proof of Theorem 4.

We now verify that the construction above is indeed a reduction from the minimum set cover problem. First suppose that C' is a minimum cover of S. We define $\triangle D$ such that it consists of the tuples $\{(i_1), \ldots, (i_{|C'|})\}$ from R, where i_j is the index of subset $c_j \in C'$. In order to delete a tuple t in $\triangle V$, we delete either $t[R_S]$ (*i.e.*, its component in R_S) or one of its components in R. Since C' is a cover of S, at least one of components of t in R is in $\triangle D$. Hence, it is clear that $Q(D \setminus \triangle D) = Q(D) \setminus \triangle V = \emptyset$. Furthermore, $\triangle D$ is minimum since (1) although deleting all the C'-tuples from table R_S suffices to delete $\triangle V$, it is not a minimum solution since $|C'| \leq k$ (and by construction, $l > k$), and (2) C' is a minimum cover of S.

Conversely, suppose that $\triangle D$ is a solution to the view side-effect problem. Then as discussed above $\triangle D$ will be only composed of tuples in R. Let C' be the subset of C such that an element c_j of C is in C' if and only if $\triangle D$ involves deletion of the tuple (j) from relation R. To see that C' is a cover of S, note that $Q(D \setminus \triangle D) = Q(D) \setminus \triangle V = \emptyset$, and thus for each $x_i \in S$, some set c_{i_j} is in C'. Moreover, C' is minimum since $\triangle D$ is minimum.

Before we conclude the proof, we present an example for the reader who may be interested in the details of the reduction. The database D is the set of all tuples as defined above. The construction is illustrated in Fig. 2 for $S = \{a,b,c,d\}$ and $C = \{c_1 = \{a,b\}, c_2 = \{a,d\}, c_3 = \{b,c\}, c_4 = \{b,c,d\}\}$. Suppose that we want to delete all tuples in the view V shown in Fig. 2. For each view tuple t, we indicate with colors which tuples (or c_i's) in R should be deleted in order to remove t from V. When all tuples are to be removed from V, *i.e.*, $\triangle V = V$, clearly deleting (1) and (4) from R achieves this goal (each tuple in V contains either (1) or (4)). Hence, $\triangle R = \{1,4\}$ and $C' = \{c_1, c_4\}$ is a minimum cover of S. □

3.2 Insertion propagation

Given a source database D, a query Q, the view $V = Q(D)$ and a set $\triangle V$ of tuples, the view side-effect problem for insertion propagation is to find a minimum set $\triangle D$ of tuples such that $Q(D \cup \triangle D) = Q(D) \cup \triangle V$, *i.e.*,

the insertion of $\triangle D$ into D produces $\triangle V$ and does not incur any side effect.

For single-tuple and group insertions, The complexity bounds of the view side-effect problem are summarized in Table 2 for various fragments of SPJU views, key preserving or not. Compared with Table 1, one can see that view insertions complicate the view side-effect analysis. In particular, the problem becomes coNP-hard for key preserving views defined with join only, even for single-tuple insertions.

To see the complication introduced by view insertions, consider the differences between insertions and deletions for key preserving views. To insert a tuple t into V, one can identify the key k_i of the tuple t_i that needs to be inserted into each occurrence of each R_i relation involved in V. As will be seen shortly, based on k_i one can either identify an existing tuple t_i in the R_i relation with k_i, or otherwise, construct a tuple t_i carrying k_i as its key and insert it into the R_i relation. Observe that while view tuple deletions can always be carried out when side effects are allowed, it is not always doable to insert a tuple into views in the presence of the key preservation condition, even if side effects are allowed.

Example 3.1: Consider the key preserving query $Q_3 = (\text{Author} \bowtie \text{Journal})$ in the setting of Example 1.1, and the insertion of tuple (Kate, TODS, XML, 35) into the view $Q_3(D)$. At first glance, it seems that this insertion can be carried out by inserting (Kate, TODS) into table Author and (TODS, XML, 35) into Journal. However, this insertion is not possible: the insertion of (TODS, XML, 35) has to be rejected since taken together with (TODS, XML, 30) it violates the key in the table Journal. □

Intractability results. In contrast to Proposition 1, the view side-effect problem is already intractable when Q is a key preserving view defined with join only, even if $\triangle V$ consists of a single tuple to be inserted.

Proposition 5. *The view side-effect problem is coNP-hard for* J *views and single-tuple insertions.*

Proof: We prove the coNP-hardness by reduction from the complement of the Boolean conjunctive query problem. An instance of that problem is an *SJ* query $q = \sigma_C(\delta_{f_1}(R_1) \bowtie \cdots \bowtie \delta_{f_k}(R_k))$ over an instance I of relational schema $\mathcal{R} = \{R_1, \ldots, R_m\}$. It is to decide whether $q(I) \neq \emptyset$. This problem is NP-complete (cf. [44]).

Given q and I, we define a source database D, a *J* query Q, and a single tuple $\triangle V$ to be inserted into the view $V = Q(D)$, such that $q(I) = \emptyset$ iff there exists a side-effect free solution $\triangle D$.

Base relations. Database D consists of $1+m+w$ relations, where w is the number of the selection conditions in C of the form "$\sigma_{A=c}$", for some attribute A and constant c. More specifically, the database includes (a) $R_0(A)$, initially empty, where A is a distinct attribute; (b) R'_1, and R_i for $i \in [2, m]$ to code the input instance I, where R'_1 is R_1 extended with the attribute A; and

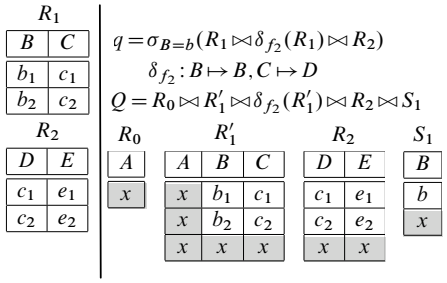

Fig. 3. Illustration of the proof of Proposition 5.

(c) a new relation S_i with one attribute A_i for each selection condition "$\sigma_{A_i=c}$" in C, to encode constants in the selection condition C. Initially, $S_i = \{(c)\}$.

View. We first define a *J* query $Q' = R_0 \bowtie q' \bowtie S_1 \bowtie \cdots \bowtie S_w$, where q is obtained from q by replacing each occurrence of R_1 by R'_1. Then, suppose that C contains p selection conditions of the form "$\sigma_{A_i=B_i}$" for some attributes A_i and B_i, for $i \in [1, p]$. We incorporate these equality conditions into the view, one by one. Initially $Q_0 = Q'$. Suppose that we have already encoded the first $j-1$ conditions as Q_{j-1}. Let $\sigma_{A_j=B_j}$ be the next selection condition. We then define $Q_j = Q_{j-1} \bowtie \delta_{A_j/B_j}(q')$, where δ_{A_j/B_j} renames A_j as B_j, while keeping the other attributes unchanged. Finally, we define $Q = Q_p$. Note that the size of Q is quadratic in the size of q. Initially, $Q(D) = \emptyset$.

View insertions. We define $\triangle V$ as a single tuple $(x, x, \ldots, x)$ to be inserted into $Q(D)$, where x is a distinct value.

See Fig. 3 for an illustration of the reduction.

One can readily verify that $q(I) = \emptyset$ iff there exists a (minimum) $\triangle D$ such that $Q(D \cup \triangle D) = V \cup \triangle V$. □

As opposed to Corollary 3, the problem also becomes harder for key preserving *PU* views and insertions. The intractability remains intact on general *PU* views.

Theorem 6. *The view side-effect problem is NP-hard for key preserving* PU *views and single-tuple insertions.*

Proof: We prove the NP-hardness by reduction from the 1-in-3 3SAT problem. An instance of the latter is $\phi = C_1 \wedge \cdots \wedge C_n$, where all variables in ϕ are $x_1, \ldots, x_k$, and each clause C_j is of the form $\ell_{j_1} \vee \ell_{j_2} \vee \ell_{j_3}$ and ℓ_{i_j} is either x_s or $\bar{x}_s$, $s \in [1, k]$. The problem is to determine whether there is a truth assignment that makes ϕ true and for which exactly one literal in each clause is assigned true. This problem is NP-complete (cf. [44]).

Given ϕ, we define a source database D, a key preserving PU query Q, and a single tuple $\triangle V$ to be inserted into the view $V = Q(D)$, such that ϕ has a 1-in-3 truth assignment iff there exists a minimum $\triangle D$ that is side-effect free, *i.e.*, $Q(D \cup \triangle D) = V \cup \triangle V$.

Base relations. The database D consists of two relations $R(K, X_1, \ldots, X_k, Y_1, \ldots, Y_k, C)$ and $R_v(K, A_1, A_2, A_3, B_1, B_2, B_3)$. Here, K is the key attribute of R and

is to enforce that zero-side effect solutions consist of a single tuple insertion in R only. The attributes X_i and Y_i are to encode truth values $\{T, F\}$, and C is an auxiliary attribute needed to check that tuples in R are truth assignments of $X = \{x_1, \ldots, x_k\}$. Initially, R is empty. As will be seen shortly, the view will extract all permutations of the truth values of the literals occurring in each clause. Clearly, when dealing with 1-in-3 truth assignments, this set of permutations is necessarily limited to $\{(T, F, F), (F, T, F), (F, F, T)\}$. However, since our view is initially empty and we are considering single-tuple insertions only, we use the relation R_v to populate the initial view with two fixed tuples, corresponding to the permutations $\{(T, F, F), (F, T, F)\}$. For this, R_v consists of a single tuple $(0, T, F, F, F, T, F)$. As before, the key attribute K of R_v is to avoid the insertion of additional tuples in R_v.

View. We define a key preserving PU query $Q = V_0 \cup V_1 \cup V_2$ as follows.

- $V_0 = \pi_{K,X,Y,Z}(\delta_{f_1}(R_v)) \cup \pi_{K,X,Y,Z}(\delta_{f_2}(R_v))$, where δ_{f_1} renames A_1, A_2, A_3 as X, Y, Z, and δ_{f_2} renames B_1, B_2, B_3 as X, Y, Z. This query yields two tuples corresponding to two of the three valid 1-in-3 truth assignments of clauses. Initially, $V_0(D) = \{(0, T, F, F), (0, F, T, F)\}$.
- $V_1 = \bigcup_{i=1}^{k} \bigcup_{\delta \in \lambda(X_i, Y_i, C)} \pi_{K,X,Y,Z}(\delta(R))$, where $\lambda(X_i, Y_i, C)$ is the set of all (bijective) renaming of X_i, Y_i, C into X, Y, Z. Intuitively, each such renaming corresponds to a permutation of X_i, Y_i, C. Since C will be enforced to equal to F (false), the view V_1 is used to verify whether tuples in R define a truth assignment. Indeed, μ defines a truth assignment if the permutations of $(\mu(x), \mu(\bar{x}), F)$ correspond to $\{(T, F, F), (F, T, F), (F, F, T)\}$, *i.e.,* the tuples in the view. Initially, $V_1(D) = \emptyset$.
- $V_2 = \bigcup_{i=1}^{n} V_{2,i}$, where each $V_{2,i}$ is to encode the clause C_i of ϕ. Suppose that $C_i = \ell_{j_1} \vee \ell_{j_2} \vee \ell_{j_3}$. If $\ell_{i_j} = x_s$ for some $s \in [1, k]$ then let $A_{i_j} = X_s$; if $\ell_{i_j} = \bar{x}_s$ then $A_{i_j} = Y_s$. We then define $V_{2,i} = \bigcup_{\delta \in \lambda(A_{j_1}, A_{j_2}, A_{j_3})} \pi_{K,X,Y,Z}(\delta(R))$, where as before, $\lambda(A_{j_1}, A_{j_2}, A_{j_3})$ is the set of all (bijective) renaming of $A_{j_1}, A_{j_2}, A_{j_3}$ into X, Y, Z. Initially, $V_2(D) = \emptyset$.

This view simply checks whether all permutations of literals in all clauses conform the 1-in-3 condition. Note that Q is a key preserving PU query.

View insertions. We define $\triangle V$ to consist of a single tuple $(0, F, F, T)$ to be inserted into $V = Q(D)$.

The reduction is illustrated in Fig. 4 for $\phi = (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_4 \vee x_5) \wedge (x_1 \vee x_3 \vee x_4)$. The tuple inserted into the view V and the tuple to be inserted into D are indicated by the bold rectangles.

We next verify that there is a 1-in-3 truth assignment for ϕ iff there exists a minimum side-effect free $\triangle D$.

First, assume that μ is a 1-in-3 truth assignment for ϕ. Let $\triangle D$ be the single tuple $(0, \mu(x_1), \mu(x_2), \ldots, \mu(x_n), \mu(\bar{x}_1), \mu(\bar{x}_2), \ldots, \mu(\bar{x}_n), F)$ to be inserted in R. Note that

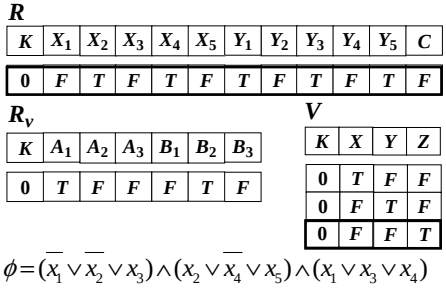

Fig. 4. Illustration of the proof of Theorem 6.

$V_0(D \cup \triangle D)$ remains unchanged. Since μ is a truth assignment, $\mu(x_i)$ and $\mu(\bar{x}_i)$ are complements for each $i \in [1, k]$. In other words, for each possible renaming δ of (X_i, Y_i, C) into X, Y, Z, $\pi_{K,X,Y,Z}(\delta(R))$ consists of tuples t of the form $(0, t[X, Y, Z])$ with $t[X, Y, Z]$ having a single T and two F-values. In other words, $V_1(D \cup \triangle D) = V \cup \triangle V$. In addition, there is exactly one T among the three literals of each clause C_j, and $Q_{2,j}$ takes all the permutations of the values of these three literals. Thus, again $V_2(D \cup \triangle D) = V \cup \triangle V$. That is, $Q(D \cup \triangle D) = V \cup \triangle V$ and hence $\triangle D$ is side-effect free. Furthermore, since no tuples could have been be added to R_v (due to the key constraints) and $\triangle D$ consists of a single tuple, $\triangle D$ is necessarily minimum.

Conversely, let $\triangle D$ be a minimum side-effect free solution such that $Q(D \cup \triangle D) = V \cup \triangle V$. Since $(0, F, F, T)$ is in $Q(D \cup \triangle D)$ and $\triangle D$ is side-effect free, we know that $\triangle D$ consists of a single tuple of the form $(0, a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n, F)$. Furthermore, from the definition of V_1 and the fact that $Q(D \cup \triangle D) = V \cup \triangle V$ we have that the mapping $\mu(x_i) = a_i$ and $\mu(\bar{x}_i) = b_i$ is a truth assignment for ϕ. Finally, since $Q(D \cup \triangle D)$ is equal to $\{(0, T, F, F), (0, F, T, F), (0, F, F, T)\}$, by the definition of V_2, we have that μ is indeed a 1-in-3 truth assignment for ϕ. Indeed, otherwise side effects would occur in $Q(D \cup \triangle D)$. □

Remark. From Proposition 5 and Theorem 6 it follows that any fragment of SPJU that contains both PU and J is both coNP-hard and NP-hard (lower bound), and is intractable. Hence such a fragment is in a complexity class that subsumes NP and coNP (see, *e.g.,* [45] for details about complexity hierarchies).

Tractability results. The good news is that the problem is tractable for SP and SU views and for group insertions, no matter whether these views are key preserving or not.

Theorem 7. *The view side-effect problem is in PTIME for (a)* SP *views and (b)* SU *views, both for group insertions.*

Proof: The proof is constructive. For each of the cases we provide a PTIME algorithm which either halts (indicating that no solution exists) or outputs a solution for the view side-effect problem. Note that some view insertions may not be doable, as shown in Example 3.1.

Key preserving SP views. Let D be a source database, Q a key preserving SP query, and $\triangle V$ be a group update

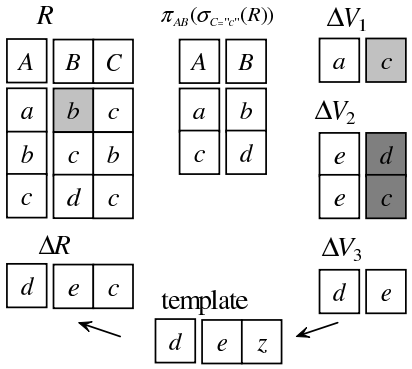

Fig. 5. Illustration of the algorithm of Theorem 7(a).

consisting of insertions only. We may assume that Q is of the form $\pi_{B_1,\ldots,B_n}(\sigma_C(\delta_f(R)))$ where R is one of the relations in the schema $\mathcal{R}$. If we denote by $A_1,\ldots,A_k$ the primary key attributes of R, then by key preservation, $\{A_1,\ldots,A_k\} \subseteq \{B_1,\ldots,B_n\}$.

For each tuple $t \in \triangle V$, we define its *tuple template* $\widehat{t} = (\vec{a}, \vec{b}, \vec{z})$, where $\vec{a} = t[A_1,\ldots,A_k]$, $\vec{b}$ consists of the constants in the remaining attributes in t, and z consists of distinct variables for each remaining attribute in R.

The PTIME algorithm for the view side-effect problem performs the following steps. We illustrate some of them in Fig. 5 for the base relation R (with A as its key), SP view $Q = \pi_{AB}(\sigma_{C="c"}(R))$ and for different group updates $\triangle V_1$, $\triangle V_2$ and $\triangle V_3$.

The algorithm first checks whether $\triangle V$ contains different tuples with the same key attributes. If so, then clearly no solution to the problem exists, and the algorithm halts. This happens, *e.g.,* for $\triangle V_2$ in Fig. 5 (the gray color indicates the conflict; it is impossible to insert two distinct tuples with the same key e).

Otherwise the algorithm continues to test for each tuple $t \in \triangle V$ whether there already exists a tuple s in R with the same key, *i.e.,* whether $s[A_1,\ldots,A_k] = \vec{a}$. If so then $\widehat{t}$ should be equal to s. If one of the $\vec{b}$ attributes of t differs from those in s, then no solution exists and the algorithms halts. This happens, *e.g.,* for $\triangle V_1$ in Fig. 5 (the gray color indicates the conflict).

Moreover, in order to get t inserted into the view, a necessary condition is that $\sigma_C(s)$ holds. If not, then no solution for the group update can be found and, again, the algorithm halts. Otherwise, we can safely remove all t from $\triangle V$ whose key already appears in R.

Finally, for each remaining tuple t in $\triangle V$ we need to instantiate the variables in its template $\widehat{t}$. More specifically, we need to instantiate these variables such that the resulting tuple (a tuple to be added to R) satisfies the selection condition C in Q. Because Q does not contain joins, we can treat each tuple in $\triangle V$ independently.

We recall that C is a conjunction of equalities of the form $x = y$, where x, y are either attributes or constants. By plugging in C the constants available in $\widehat{t}$, *i.e.,* those in $\vec{a}$ and $\vec{b}$, we obtain a new conjunction C' (with possibly less variables). By constructing a dependency graph G between the constants and variables in C' and computing its transitive closure G', one can then easily check whether a desired instantiation of the variables exists. Indeed, if there exists an edge $(a, b) \in G'$ with a, b two different constants, then no instantiation exists. We say that C' is conflicting. Consequently, in this case no solution to the view side-effect problem exists and the algorithm halts. Otherwise, one assigns to all the variables in the same connected component in G' the same constant value (*i.e.,* the value of the unique constant in this component, or an arbitrary one if the connected component consists of variables only). Variables not appearing in C can be instantiated arbitrarily. The resulting tuple is then added to $\triangle D$.

The algorithm successfully computes a solution to the view side-effect problem if for each tuple in $\triangle V$ (modulo the ones whose key already appeared in R) a tuple is added to $\triangle D$. In all other cases, no solution exists. For example, in Fig. 5 a solution for $\triangle V_3$ exists. First, the tuple in $\triangle V_3$ is expanded to a template (introducing the variable z), then this variable is instantiated using the condition $C =$"c" of the selection predicate of Q.

We remark that when a solution exists, $\triangle D$ computed by the above algorithm is of minimum size. Indeed, for each new key in $\triangle V$, a single tuple with this key is added to $\triangle D$. Since Q is key preserving, this is the minimum number of tuples required for any solution. Moreover, it is easy to see that this solution is side-effect free.

The algorithm clearly runs in polynomial time.

Arbitrary SP views. Let us now drop the key preservation condition on the view Q. We use the same approach as in the key preserving case, except that we do not have to check for conflicting keys. However, even in the absence of key preservation, the update to the view cannot always be performed successfully. As we will see below, a necessary condition is that the tuples to be inserted in the view can be extended to tuples in the base relation satisfying the selection condition C.

Indeed, for each tuple t in $\triangle V$ to be inserted into the SP view $Q(D)$, we create a tuple template $\widehat{t} = (t, \vec{z})$ where $\vec{z}$ consists of variables for the attributes in $R \setminus \{B_1,\ldots,B_m\}$.

We then proceed by checking for each template $\widehat{t}$ whether there exists already tuples s in R such that (i) $\widehat{t}$ and s agree on the output schema of Q; and (ii) $\sigma_C(s)$ holds. If there exists such a tuple s, then $\widehat{t}$ is set to s, and we can safely remove t from $\triangle V$ (it will be in the view). Otherwise, if $\sigma_C(s)$ does not hold or no such s exists, then we need to instantiate the variables $\vec{z}$ in $\widehat{t}$ in such a way that for the resulting tuple t_0, $\sigma_C(t_0)$ holds. The tuples t_0 will make up the update $\triangle D$ to the database.

Testing whether such an instantiation exists can be done similarly as in the key preserving case above. If this can be done successfully for each template, then $\triangle D$ will be a solution to the view side-effect problem. In fact, this solution does not introduce any side effects. Also, $\triangle D$ is minimum, because only the necessary tuples are inserted in D (we use existing tuples where possible).

The algorithm clearly runs in polynomial time.

Key preserving SU-views. Consider a key preserving SU query $Q = \bigcup_{i=1}^{k} Q_i$, an instance D of the schema $\mathcal{R}$, and a group insertion $\triangle V$. We assume that each S query Q_i is of form $Q_i = \sigma_{C_i}(\delta_{f_i}(R_{j_i}))$. Observe that if there exists a source insertion $\triangle D$ that produces $\triangle V$, then there exists a side-effect free solution. Indeed, since Q does not involve joins one can always trim irrelevant tuples from the query result by adjusting values in $\triangle D$.

The PTIME algorithm first models the view side-effect problem as a flow network with integer capacity on each edge, and then computes the maximum flow of the network in polynomial time. If the value of the maximum flow equals to $|\triangle V|$, then we can get a solution to the view side-effect problem. Otherwise, there exists no solution for group insertions $\triangle V$.

We first model $\mathcal{R}$, Q and $\triangle V$ as a flow network. The initial vertex set of the flow network is $\mathcal{N} = \{S, T\} \cup \{T_i \mid t_i \in \triangle V\} \cup \{r_i \mid R_i \in \mathcal{R}\}$, where S is the source node, T is the sink node, T_i represents a tuple node for each $t_i \in \triangle V$, and r_i represents a relation node for each relation R_i involved in V. The initial edge set consists of $\mathcal{E} = \{(S, T_i) \mid i \in [1, k]\} \cup \{(r_i, T) \mid i \in [1, n]\}$, where k is the size of $\triangle V$, n is the size of $\mathcal{R}$, the capacity of edges (S, T_i) is 1 and that of edge (r_i, T) is ∞.

We next encode $\triangle V$. For each tuple t in $\triangle V$, we check for each Q_i whether 1) t satisfies the selection condition C_i in Q_i; and 2) whether there exists no tuple in $Q_i(D)$ having conflicting key with t. If both conditions are satisfied for Q_i, the relation R_{j_i} in Q_i is called a *candidate host* for tuple t. If there exists no such a candidate host for t, the algorithm halts and no solution to the view side-effect problem exists. If all tuples in group update have at least one candidate host, then the algorithm continues.

Note, however, that t will not necessarily be inserted in a candidate host. Indeed, whether or not t will be inserted into one of its candidate hosts, say R_{j_i}, also depends on whether the possible other insertions (from $\triangle V$) into R_{j_i} cause a key violation together with the insertion incurred by t. The information regarding key information amongst tuples in their candidate hosts is modeled in the flow network as follows. For each candidate host R_{j_i} and each $t_\ell \in \triangle V$, we update the flow network $\mathcal{N}$. Let $\vec{a}_{j_i}$ be the tuple consisting of key attributes of t_ℓ in R_{j_i}. We distinguish between the following two cases.

• *Case 1:* There is no edge of the form $(v(\vec{a}_{j_i}), r_{j_i})$, where v is a vertex with $\vec{a}_{j_i}$ as its label. We add a new vertex v_{new} with $\vec{a}_{j_i}$ as label into $\mathcal{N}$, and add new edges (T_ℓ, v_{new}) and $(v_{\text{new}}, r_{j_i})$. The capacity of each new edge is 1, representing that at most one tuple with this key can be inserted in the host R_{j_i}.

• *Case 2:* There is already an edge of the form $(v(\vec{a}_{j_i}), r_{j_i})$, where v is a vertex with $\vec{a}_{j_i}$ as its label. This indicates that some other tuples (conflicting with t_ℓ on R_{j_i}) with the same key attributes $\vec{a}_{j_i}$ exist. Note that together with t_ℓ, only one such tuple can be inserted into R_{j_i}. Thus, we add an edge (T_ℓ, v) with capacity 1.

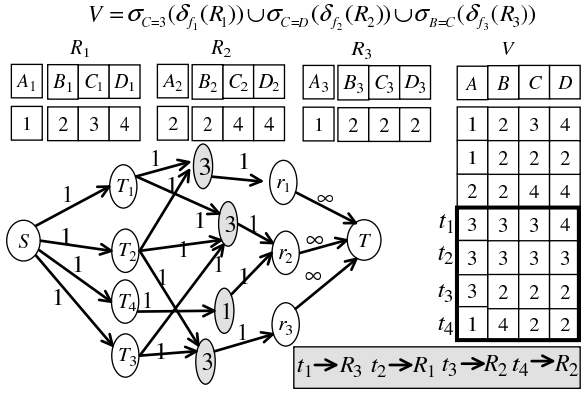

Fig. 6. Illustration of the algorithm of Theorem 7(b).

These steps complete the construction of flow network. The construction clearly runs in polynomial time. Below we illustrate the construction. Figure 6 shows the flow network for base relations $R_i(A_i, B_i, C_i, D_i)$, for $i = 1, 2, 3$, and SU view $V = \sigma_{C=3}(\delta_{f_1}(R_1)) \cup \sigma_{C=D}(\delta_{f_2}(R_2)) \cup \sigma_{B=C}(\delta_{f_3}(R_3))$, where δ_{f_i} renames A_i, B_i, C_i, D_i as A, B, C, D for $i = 1, 2, 3$. The group update $\triangle V$ is indicated by the bold rectangle. The gray shadowed vertexes are added following the rules in case 1 and case 2 and each of them has only one outgoing edge. As formally shown below, there is a solution for the insertion of $\triangle V$ into V iff the value of the maximum flow of the constructed flow network is $|\triangle V|$. For this example, there is a solution $\triangle D$ that inserts t_1, t_2, t_3, t_4 into R_3, R_1, R_2, R_2, respectively.

More formally, we show that there is a solution to the view side-effect problem iff the value of the maximum flow of the constructed flow network is $|\triangle V|$. First, we assume that the maximum flow $\phi : \mathcal{E} \to R^+$ equals $|\triangle V|$. By definition, this implies that $\sum_{(S,T_i)} \phi(S, T_i) = |\triangle V|$, or in other words, that ϕ assigns 1 to every edge starting from S. By the flow conversation law, *i.e.,* for each $v \in \mathcal{N} \setminus \{S, T\}$, $\sum_{(v,w)\in\mathcal{E}} \phi(w, v) = \sum_{(v,w)\in\mathcal{E}} \phi(v, w)$, and the fact that all other edges (except those adjacent to T) have weight 1 assigned to them, this in turn implies that ϕ defines a unique path from S to T, one for each tuple in $\triangle V$. Since the in- and outgoing edges from the key-labeled vertexes have weight 1, these paths do not share key-labeled vertexes. Thus, form those paths one can infer the candidate hosts into which to insert each of the tuples. Indeed, for $t_i \in \triangle V$, one can simply follow the path (as defined by ϕ) starting form T_i until the corresponding host relation is reached.

Conversely, if a side-effect free solution exists, then there is a $\triangle D$ such that $Q(D \cup \triangle D) = V \cup \triangle V$. Since the view is key preserving, for any two distinct tuples $t_1, t_2 \in \triangle V$, there are different $t_1', t_2' \in \triangle D$ inserted in relations R_1, R_2, which guarantee that t_1 and t_2 appear in the view. We distinguish between the following two cases: (a) R_1 and R_2 are different relations; and (b) R_1 and R_2 are the same relation R. For case (a), the construction of the flow network indicates that starting from S,

we can reach T_1 and T_2 through different edges, and then reach different labeled nodes through different edges; finally we reach different nodes r_1 and r_2. For case (b), the validity of the insertion guarantees that t_1 and t_2 have different keys on relation R. Thus, we can start from S and reach T_1 and T_2 through different edges, and then reaches different labeled nodes, and finally reach node r. Since t_1, t_2 were chosen arbitrarily, this implies the existence of S to T paths passing through each tuple node in $\mathcal{N}$. Furthermore, all these paths share no labeled node and tuple node. This means that there is a feasible flow with value $|\triangle V|$.

Since for any flow ϕ, $\sum_{(S,T_i)} \phi(S,T_i) \leqslant |\triangle V|$, the maximum flow value is $|\triangle V|$. From this it follows that the algorithm is in PTIME.

Arbitrary SU views. Since SU views do not have projections, they are key preserving. Hence the PTIME algorithm above also works on arbitrary SU views. □

4 The Source Side-Effect Problem

In this section we investigate the source side-effect problem. We study the problem for various key preserving SPJU views in Section 4.1 for both single-tuple and group deletions. Insertions are considered in Section 4.2.

4.1 Deletion propagation

Given a view deletion $\triangle V$, the source side-effect problem for deletion propagation is to find a smallest set $\triangle D$ of source tuples to be deleted so that the tuples in $\triangle V$ are removed from the view.

Table 3 gives the complexity of determining the minimum $\triangle D$ for various subclasses of SPJU queries, for single-tuple and group deletions. Compared to Table 1, the results tell us that it is already hard to determine whether there exists $\triangle D$ to produce $\triangle V$, even without checking whether $\triangle D$ is side-effect free.

It has been shown that the source side-effect problem is already NP-hard for single deletions and PJ views [1]. We show that the problem for single deletions becomes polynomial-time solvable when the key preservation condition is imposed. This again verifies our observation that key preservation simplifies the analysis.

Proposition 8. *The source side-effect problem is in PTIME for key preserving* SPJ *views and single-tuple deletions.*

Proof: The PTIME algorithm presented in the proof of Proposition 1 is already able to compute a minimum source update $\triangle D$. We can therefore use the same algorithm for the source side-effect problem, except that we do not have to perform the steps for selecting the update that minimizes the number of view side effects. □

However, the problem for group deletions remains hard. Similar to Theorem 4, we show that the source side-effect problem is intractable for views defined with join only and for group deletions. This problem has not been considered by previous work.

Corollary 9. *The source side-effect problem is NP-*hard *for key preserving J views and group deletions.*

Proof: The proof of Theorem 4 suffices to show this. Indeed, the reduction given in that proof assures the minimality of the size of the source updates, and it does not impose any constraints on the size of side effects. □

For JU views, the problem is getting no easier, similar to its view side-effect counterpart (Corollary 2).

Corollary 10. *The source side-effect problem is NP-hard for key preserving JU views and single-tuple deletions.*

Proof: The proof of Corollary 2 is applicable here. Its reduction [1] concerns only the existence of a smallest source update that produces view updates. □

In contrast, for SPU views the analysis is simpler, comparable to its view side-effect counterpart (Corollary 3).

Corollary 11. *The source side-effect problem is in PTIME for* SPU *views and group deletions.*

Proof: The PTIME algorithm given in the proof of Corollary 3 suffices to find smallest source updates. □

4.2 Insertion Propagation

Given a source database D, a query Q, the view $V = Q(D)$ and a set $\triangle V$ of tuples, the source side-effect problem for insertion propagation is to find a smallest set $\triangle D$ of tuples such that $Q(D \cup \triangle D)$ contains $\triangle V$, *i.e.*, we want to find a smallest set of tuples to insert into the source database such that the insertion will get $\triangle V$ into the view, regardless of side effects on the view.

For single-tuple and group insertions, the complexity results for the source side-effect problem are summarized in the Table 4. Compared to its view side-effect counterpart (Table 2), the source side-effect problem is relatively easier for insertions since we no longer need to check whether source insertions are side-effect free.

Intractability results. We first show that like its view side-effect counterpart (Proposition 5), the source side-effect problem is intractable for general PJ (and thus SPJ) views and single-tuple insertions. This tells us that the source side-effect analysis for insertions is more intriguing than its deletion counterpart (Proposition 8).

Theorem 12. *The source side-effect problem is NP-*hard *for PJ views and for single-tuple insertions.*

Proof: We prove the NP-hardness by reduction from the minimum set cover problem (see the proof of Theorem 4 for the statement of this problem). It is known that this problem is NP-complete [44].

Given S and C, we define an instance of the source side-effect problem. Let $S = \{x_i \mid i \in [1,n]\}$ and $C = \{c_j \mid j \in [1,k]\}$. We construct a source database D, a PJ-query Q, the view $V = Q(D)$, and a single tuple $\triangle V$ to be inserted into V. We show that we can find a minimum cover C' of S iff there exists a smallest set $\triangle D$ of tuples such that $Q(D \cup \triangle D)$ contains $\triangle V$.

Base relations. We define $k+2$ relations, including R_S^i for $i \in [1, k+1]$ and a relation R_C.

• $R_S^i(I_S, I_C)$, for $i \in [1, k+1]$, where I_S and I_C range over $[1, n]$ and $[1, k]$, respectively. Initially, (i, j) is in D iff $x_i \in c_j$, *i.e.,* (i, j) indicates whether or not the element x_i of S is in the subset c_j in the collection C. As will be seen shortly, we keep $k+1$ copies of the $R_S^i(I_S, I_C)$ relation to prevent insertions into any of these relations.

• $R_C(I_C)$ is to hold the elements of C to be picked for covering S. In other words, $R_C(I_C)$ is to represent a cover C' (after it is picked) such that (j) is in D iff $c_j \in C'$, for $j \in [1, k]$. Initially R_C in D is empty, *i.e.,* no element of C is picked yet.

View. We define a PJ view $Q = \delta_{f_1}(Q') \bowtie \cdots \bowtie \delta_{f_n}(Q')$, where Q' is $\pi_{I_S}(R_S^1 \bowtie R_S^2 \bowtie \cdots \bowtie R_S^{k+1} \bowtie R_C)$, and δ_{f_i} renames I_S to a distinct name I_S^i in order to conduct cross product (rather than natural join). A tuple in the view is a n-vector $(a_1, \ldots, a_n)$, where $a_i \in [1, n]$. Initially, $V = Q(D)$ is empty. Note that Q is not key preserving.

View insertion. The tuple $\triangle V$ is $(1, \ldots, n)$. It is to force a cover C' to be picked, *i.e.,* every element x_i in S is to be covered by some subset c_j in C'.

An example to illustrate the reduction can be found toward the end of the proof.

We show that this is indeed a reduction. First, assume that C' is a minimum cover of S. Then we construct source tuples $\triangle D$ such that (j) is inserted into $R_C(I_C)$ iff $c_j \in C'$. Obviously, $\triangle V \in Q(D \cup \triangle D)$ since C' is a cover, and moreover, $\triangle D$ is minimum since C' is minimum. Note that, however, $\triangle D$ is not side-effect free: $Q(D \cup \triangle D)$ contains all permutations of $(1, \ldots, n)$. But side effects are not the concern of the source side-effect problem.

Conversely, suppose that there is a smallest $\triangle D$ such that $\triangle V \in Q(D \cup \triangle D)$. Note that $\triangle D$ consists of insertions to $R_C(I_C)$ only. Indeed, if one wants to insert tuples into $R_S^i(I_S, I_C)$, for some $i \in [1, k+1]$, in order to add a tuple to the view, the same insertions always have to be performed to all $k+1$ source relations $R_S^i(I_S, I_C)$. The minimum solution consists of maximal k updates.

Given the minimum update $\triangle D$ to $R_C(I_C)$, we define a set C' such that c_j is in C' iff (j) is in $R_C(I_C)$ in $\triangle D$. Since $(1, \ldots, n)$ is in $Q(D \cup \triangle D)$, from the definition of Q it follows that $\triangle D$ consists of (j)'s such that for any $i \in [1, k]$, there exists $(i, j) \in R_S(I_S, I_C)$. Thus C' is a cover of S. In addition, C' is a minimum cover since $\triangle D$ is minimum. That is, C' is a minimum cover of S.

Finally, we present an example to illustrate the reduction. Consider $S = \{a, b, c, d\}$ and $C = \{c_1 = \{a, b\}, c_2 = \{a, d\}, c_3 = \{b, c\}, c_4 = \{b, c, d\}\}$. Figure 7 illustrates the reduction: the tuple inserted into the view and the tuples to be inserted into R_C are indicated by the bold rectangles. Tuples in R_C determine which sets in C are considered to be in a (minimum) cover of S. The colors represent the two elements in C, $c_1 = \{a, b\}$ and

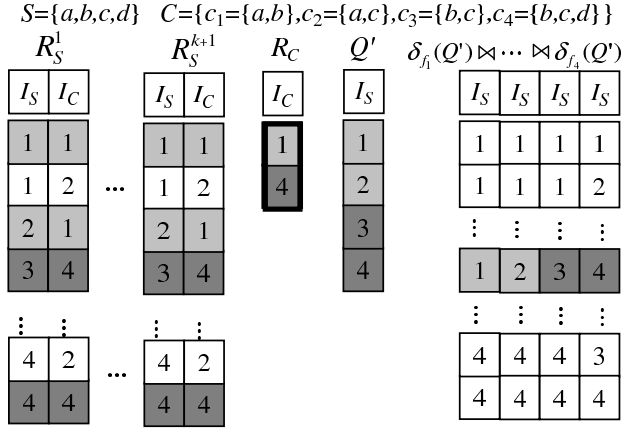

Fig. 7. Illustration of the proof of Theorem 12.

$c_4 = \{b, c, d\}$, selected by the insertion of (1) and (4) in R_C. It can be seen that the intermediate relation Q' contains all elements in S, which implies that $\{c_1, c_4\}$ form a cover of S. The insertion of these two elements in R_C is forced by the insertion of $(1, 2, 3, 4)$ in the view. As explained below, $k+1$ copies of R_S are needed to prevent an insertion in those base relations (as updates to one relation will cause an update in all $k+1$). □

The problem is no easier for *JU* views and group insertions, even when the views are key preserving.

Theorem 13. *The source side-effect problem is NP-hard for key preserving* JU *views and group insertions.*

Proof: We show this by reduction from the hitting-set problem. An instance of that problem consists of a collection C of subsets of a finite set S; it is to find a minimum subset X of S such that $X \cap c_i \neq \emptyset$ for all $c_i \in C$. The problem is NP-complete (cf. [44]).

Given S and C, we define an instance of the source side-effect problem. Let $S = \{x_i \mid i \in [1, n]\}$ and $C = \{c_j \mid j \in [1, k]\}$. We construct a source database D, a key-preserving JU query Q, the view $V = Q(D)$, and a group insertion $\triangle V$. We show that we can find a minimum hitting set X of S iff there exists a minimum set $\triangle D$ such that $Q(D \cup \triangle D) \supseteq V \cup \triangle V$.

Base relations. For each $x_i \in S$ ($i \in [1, n]$), we define two relations $R_{(i,1)}(A_i, B_i)$ and $R_{(i,2)}(B_i, D_i)$ as follows.

• $R_{(i,1)}(A_i, B_i)$, where A_i is the key, and A_i, B_i range over $[1, k]$ and $\{T, F\}$, respectively. Intuitively, a tuple in $R_{(i,1)}$ indicates whether or not x_i belongs to a subset of C. Initially, (j, F) is in $R_{(i,1)}$ iff $x_i \notin c_j$.

• $R_{(i,2)}(B_i, D_i)$, where D_i is the key, and B_i, D_i range over $\{T, F\}$ and $[0, k]$, respectively. Intuitively, a tuple in $R_{(i,2)}$ indicates whether or not x_i is included in the hitting-set. Initially, $R_{(i,2)}$ is empty.

View. We define a JU query $Q = \delta_{f_1}(Q_1) \cup \cdots \cup \delta_{f_n}(Q_n)$ where $Q_i = R_{(i,1)}(A_i, B_i) \bowtie R_{(i,2)}(B_i, D_i)$ and δ_{f_i} renames A_i, B_i, D_i as A, B, D respectively. Initially, $V = Q(D) = \emptyset$. Intuitively, a tuple in the view is a triple

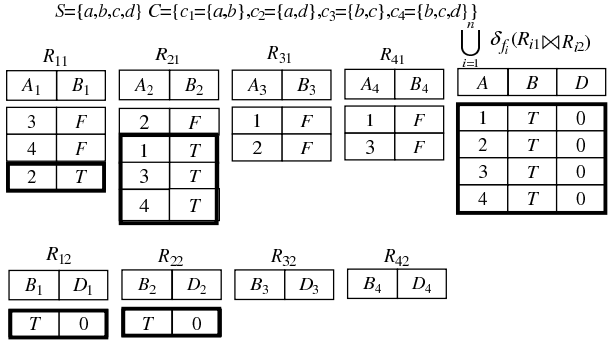

Fig. 8. Illustration of the proof of Theorem 13.

(j, b, x), indicating whether at least one element of c_j is covered by the hitting set.

View Insertions. The set $\triangle V$ is $\{(j, T, 0) \mid j \in [1, k]\}$.

The construction is illustrated in Fig. 8 for $S = \{a, b, c, d\}$ and $C = \{c_1 = \{a, b\}, c_2 = \{a, d\}, c_3 = \{b, c\}, c_4 = \{b, c, d\}\}$. The tuples inserted into the view and the tuples to be inserted into D are indicated by the bold rectangles. As will be seen shortly, $\triangle D$ in this example corresponds to a hitting-set $X = \{a, b\}$ of C.

We next show the correctness of the reduction. We first make the following crucial observation: for any source insertion $\triangle D$, if $V \cup \triangle V \subseteq Q(D \cup \triangle D)$, then $|\triangle D| \geq k + h$, where h is the size of minimum hitting-set of C. To show this, we consider an index set $I = \{i \mid i \in [1, n],$ tuple $(T, 0)$ is inserted into $R_{(i,2)}$, $R_{(i,1)}$ accepts at least one tuple from $\triangle D\}$. Tuples inserted into $R_{(j,1)}$ or $R_{(j,2)}$ $(j \notin I)$ are redundant, because they never generate any tuple in the view. Moreover, since $Q(D \cup \triangle D)$ contains $\triangle V$, for each $j \in [1, k]$, (j, T) must be inserted into some $R_{(i,1)}$ with $i \in I$. This means that each c_j must contain an element x_i with $i \in I$, which results in a hitting set $\{x_i \mid i \in I\}$ of C. Putting these together, $|\triangle D| \geq k + |I| \geq k + h$.

We next continue with verifying the correctness of the reduction. First, assume that X is a minimum hitting-set of C. We then construct a set of source tuples $\triangle D$ such that $Q(D \cup \triangle D) \supseteq V \cup \triangle V$ and $|\triangle D|$ is minimum. Let $c_j \in C$ and take one x_i from $c_j \cap X$ that is not empty. Then we insert (j, T) into relation $R_{(i,1)}$ and insert $(T, 0)$ into $R_{(i,2)}$. Note that by the keys, only one insertion can succeed. Hence the total number of successful insertions of $(T, 0)$ into D will never exceed $|X| = h$. As a consequence, $(j, T, 0)$ belongs to $Q(D \cup \triangle D)$. Thus, $Q(D \cup \triangle D) \supseteq V \cup \triangle V$ and $|\triangle D| \leq k + h$. Together with the observation above, we see that $|\triangle D|$ is minimum.

Conversely, assume that $\triangle D$ is a minimum set such that $Q(D \cup \triangle D) \supseteq V \cup \triangle V$. We construct a minimum hitting-set X of C. Let $I = \{i \mid$some tuple in $\triangle D$ is inserted into relation $R_{(i,1)}\}$. Since $|\triangle D|$ is minimum, there are no redundant insertions and thus $(T, 0)$ must be inserted into $R_{(i,2)}$ for all $i \in I$. Similarly, since $Q(D \cup \triangle D) \supseteq \triangle V$, we have that for each $j \in [1, k]$, (j, T) is inserted into D exactly once (into some relation $R_{(i,1)}$ with $i \in I$). Thus, $X = \{x_i \mid i \in I\}$ is a hitting-set of C and $|\triangle D| = k + |I| = k + |X|$. If there exists another hitting-set X' such that $|X'| < |X|$, we can use the construction method previously described to find a solution $\triangle D'$ such that $|\triangle D'| \leq k + |X'| < k + |X| = |\triangle D|$, which contradicts the assumption that $\triangle D$ is minimum. □

The problem for *PU* views is as hard as for *JU* views.

Theorem 14. *The source side-effect problem is NP-hard for key preserving* PU *views and group insertions.*

Proof: We prove this by reduction from the 1-in-3 3SAT problem. We refer to the proof of Theorem 6 for the statement of the 1-in-3 3SAT problem.

Given an instance ϕ of the 1-in-3 3SAT problem, we define a source database D, a key preserving PU query Q, and a set $\triangle V$ of tuples to be inserted into the view $V = Q(D)$ such that ϕ is 1-in-3 satisfiable iff there exists a (minimum) source insertion $\triangle D$ such that $V \cup \triangle V \subseteq Q(D \cup \triangle D)$. The reduction is similar to the one given in the proof of Theorem 6.

Base relations. The database D consists of one relation $R(K, X_1, \ldots, X_k, Y_1, \ldots, Y_k, C_1, \ldots, C_{n+k}, W)$, where K is the key, the attributes X_i and Y_i range over $\{T, F\}$ for $i \in [1, k]$, C_j ranges over $[1, n+k]$ for all $j \in [1, n+k]$, and W is in $\{F\}$. Intuitively, $X_i, Y_i (1 \leq i \leq k)$ encode the truth value of x_i and its negation, respectively. We use C_{n+i} together with W to determine whether a truth assignment is valid, *i.e.,* whether only one of X_i and Y_i is T. We use C_j $(1 \leq j \leq n)$ to check whether clause c_j is satisfied by the truth assignment. Initially, D is empty.

View. We define a key preserving PU query $Q = V_1 \cup V_2$.

- $V_1 = \cup_{i=1}^{k} V_{1,i}$, where
$$V_{1,i} = \cup_{\delta \in \lambda(X_i, Y_i, W)} \pi_{K,X,Y,Z,C}(\delta_{n+i}(\delta(R))),$$
where $\lambda(X_i, Y_i, W)$ is the set of all (bijective) renaming of X_i, Y_i, W as X, Y, Z, and δ_{n+i} renames C_{n+i} as C. Intuitively, renaming in $\lambda(X_i, Y_i, W)$ enumerates permutations of X_i, Y_i and W. The query V_1 is used to verify whether tuples in R define a valid truth assignment. Initially, $V_1(D) = \emptyset$.

- $V_2 = \bigcup_{i=1}^{n} V_{2,i}$, where each $V_{2,i}$ is defined according to the clause C_i of ϕ. Suppose that $C_i = \ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3}$. If $\ell_{i_j} = x_s$ for some $s \in [1, k]$ then let $A_{i_j} = X_s$; if $\ell_{i_j} = \bar{x}_s$ then let $A_{i_j} = Y_s$. We then define
$$V_{2,i} = \bigcup_{\delta \in \lambda(A_{i_1}, A_{i_2}, A_{i_3})} \pi_{K,X,Y,Z,C}(\delta_i(\delta(R))),$$
where as before, $\lambda(A_{i_1}, A_{i_2}, A_{i_3})$ is the set of all (bijective) renaming of $A_{i_1}, A_{i_2}, A_{i_3}$ as X, Y, Z, and δ_i renames C_i as C. Intuitively, $V_{2,i}$ introduces a tuple into V iff clause c_i is 1-in-3 satisfied by the truth assignment encoded by a tuple of D. Initially, $V_2(D) = \emptyset$.

View updates. Let $\triangle V = \{(0, \tau, j) \mid \tau = (\tau[X], \tau[Y], \tau[Z]),$ τ is a permutation of $(T, F, F), j \in [1, n + k]\}$.

We next verify the correctness of the reduction. First, assume that μ is a 1-in-3 truth assignment for ϕ. Let $\triangle D$ be the tuple

$(0, \mu(x_1), \ldots, \mu(x_k), \bar{\mu}(x_1), \ldots, \bar{\mu}(x_k), 1, \ldots, n+k, F)$.

By the view definition and the fact that μ is a truth assignment, we have that $V_1(\triangle D)$ is the set $\{(0, \tau, j) \mid \tau = (\tau[X], \tau[Y], \tau[Z])$ is a permutation of $(T, F, F), j \in [n+1, n+k]\}$. Similarly, we have that $V_2(\triangle D)$ is the set $\{(0, \tau, j) \mid \tau = (\tau[X], \tau[Y], \tau[Z])$ is a permutation of $(T, F, F), j \in [1, n]\}$. Hence, $Q(D \cup \triangle D) = V \cup \triangle V$.

Conversely, assume that there exists a (minimum) source insertion $\triangle D$ such that $V \cup \triangle V = \triangle V \subseteq Q(\triangle D) = Q(D \cup \triangle D)$. Since the tuple $(0, T, F, F, 1) \in \triangle V$ is inserted into the view, R has key attribute K, and Q is key preserving, one can see that $\triangle D$ must contain a unique tuple t with $t[K] = 0$. Since all other tuples in $\triangle V$ share this key value, t is the only tuple in $\triangle D$ contributing to $\triangle V$. We can therefore assume that $\triangle D$ consists of the single tuple t. By the definition of V, each $V_{i,j}$ can only insert exactly one set of tuples into the view, where the set is of the form $\{(0, \tau, g(i,j)) \mid \tau$ is a permutation of $(T, F, F)\}$, and $g(i,j)$ is the value of $t[C_{n+j}]$ when $i = 1$ and it is $t[C_j]$ when $i = 2$. Since we know that there are precisely $n+k$ such sets in $\triangle V$, and there are precisely $n+k$ sub-queries $V_{i,j}$, each mapping $g(i,j)$ must be a bijection. That is, $g : V_{i,j} \to [1, n+k]$ is a bijection, where $g(1,j) = t[C_{n+j}]$ and $g(2,j) = t[C_j]$.

We use the mapping g to define a truth assignment of $x_1, \ldots, x_k$. For arbitrary x_i, $(0, T, F, F, g(1,i))$ is inserted into V through a subquery $V_{1,i}$. By the definition of $V_{1,i}$, we can conclude that $\{t[X_i], t[Y_i]\} = \{T, F\}$. Hence, a truth assignment for ϕ is defined by $\mu(x_1) = t[X_1], \ldots, \mu(x_k) = t[X_k]$. Next, we verify that this assignment is a 1-in-3 truth assignment of ϕ. Consider an arbitrary clause C_i of ϕ. Since $(0, T, F, F, g(2,i))$ is inserted into V through subquery $V_{2,i}$, by the definition of $V_{2,i}$, we know that there is exactly one T among the truth values of the three literals in the clause. Hence, μ is indeed a 1-in-3 truth assignment. □

Tractability results. We next identify some polynomial-time solvable subclasses. As in the view side-effect analysis (Theorem 7), *SP* views and *SU* views behave well. In contrast to their view side-effect counterparts (Proposition 5), the source side-effect analyses of *SJ* views and key preserving *SPJ* views also become simpler.

The result below also tells us that the source side-effect problem is in PTIME for SJ views and group insertions. In contrast, for group deletions the problem is NP-hard for any views defined with join operations, no matter whether they are key preserving or not (Corollary 9).

Theorem 15. *The source side-effect problem is in PTIME for (a)* SP *views, (b)* SU *views, (c)* SJ *views, and (d) key preserving* SPJ *views, for group insertions.*

Proof: The PTIME algorithms for cases (a) and (b) are similar to those of the view side-effect problem given in the proof of Theorem 7. In fact, the algorithms provided there return solutions for the source side-effect problem. We therefore concentrate on cases (c) and (d). These cases requires a bit more effort (recall that the view side-effect problem for these cases is intractable, by Proposition 5).

SJ views. Consider an SJ query $Q = \sigma_C(\delta_{f_1}(R_1) \bowtie \cdots \bowtie \delta_{f_k}(R_k))$. A PTIME algorithm is given as follows.

Because Q does not contain any projection, we can derive for each tuple t in $\triangle V$ and for each relation R_i ($i \in [1,k]$) in Q a candidate insert tuple $\widehat{t}_i = (\vec{a}_i, \vec{b}_i)$ over the attributes of R_i, where $\vec{a}_i$ corresponds to the key attributes of R_i. We then check for each $t \in \triangle V$ whether $(\delta_{f_1}(\widehat{t_1}) \bowtie \cdots \bowtie \delta_{f_k}(\widehat{t_k}))$ satisfies the selection condition C. If not, then no solution exists and the algorithm halts. Otherwise, we check for each $\widehat{t}_i$ whether there exists already a tuple s_i in R_i having the same key $\vec{a}_i$. If this is the case, $\widehat{t}_i$ should be equal to s_i. If there exists a tuple $\widehat{t}_i$ for which this does not hold, then no solution exists and the algorithm stops. Otherwise, the algorithm continues.

Denote by $\triangle R_i$ the set of tuples $\widehat{t}_i$ for which no tuple in R_i exists with the same key. We finally check whether $\triangle R_i$ contains distinct tuples having the same key. If such tuples exists, no solution can be found and the algorithm halts. Otherwise, we define $\triangle D$ to be $\{\triangle R_1, \ldots, \triangle R_k\}$.

We remark that $\triangle D$ is the minimum solution. Indeed, in each instance R_i, the number of tuples inserted into D is the same as the number of new keys for R_i present in $\triangle V$. This is the minimum requirement for any solution.

The algorithm clearly runs in polynomial time.

Key preserving SPJ views. Consider a key preserving SPJ query $Q = \pi_{B_1,\ldots,B_m}\sigma_C(\delta_{f_1}(R_1) \bowtie \cdots \bowtie \delta_{f_k}(R_k))$.

As before, for each tuple t in $\triangle V$ and each relation R_i in Q, we associate a template $\hat{t}_i = (\vec{a}_i, \vec{b}_i, \vec{z}_i)$.

The PTIME algorithm first checks for incompatible templates. More specifically, the algorithm checks whether (i) there are no two different templates with the same key; and (ii) no template $\hat{t}_i$ with the same key as an existing tuple s_i in R_i, but which differs from s_i in another attribute. If no incompatible templates are found, then the algorithm continues.

If no conflicts are found, we define $\triangle R_i$ to be the set of templates $\hat{t}_i$ which have no matching tuple s_i in R_i. We instantiate the variables in these templates as follows. For each tuple t in $\triangle V$, we compute a conjunctive formula ϕ_t representing the selection and join conditions to hold on $\hat{t}_1 \wedge \hat{t}_2 \wedge \cdots \wedge \hat{t}_k$, such that it will generate t in the view. The formula ϕ_t consists of conjuncts of equations of the form $x = y$, where x and y are either variables or constants in $\hat{t}_i$, for $i \in [1,k]$. We group together all conjuncts ϕ_t into a single conjunction $\Phi = \wedge_{t \in \triangle V} \phi_t$, and check whether there exists an instantiation of the variables that satisfies Φ, using a method similar to the one given for case (a) in the proof of Theorem 7.

The algorithm is illustrated in Fig. 9 for the base relations R_1, R_2, R_3, with keys A, C and D, respectively, the key preserving SPJ view $Q = \pi_{ACD}(\sigma_{A=E}(R_1 \bowtie R_2 \bowtie R_3))$, and view update $\triangle V$. We also depict the templates for each tuple in $\triangle V$. For example, we show Φ and a possible instantiation of the variables. The updated view is shown on the bottom right. In this case no side

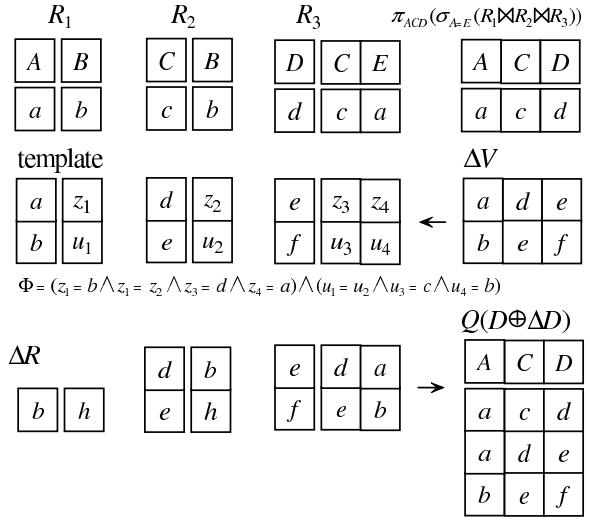

Fig. 9. Illustration of the algorithm of Theorem 15(d).

effects were created (while in general, side effect may be inevitable).

Since we are not concerned about the size of the side effects, we do not have to take into account constraints regarding existing constants in the database (this is in contrast to the coNP-hardness proof of Proposition 5). Hence, if an instantiation exists, we can convert the templates into tuples that populate the update set $\triangle R_i$. Finally, we define $\triangle D = \{\triangle R_1, \ldots, \triangle R_k\}$.

Obviously, $\triangle D$ is a solution. It is also minimum. Indeed, at most a single tuple for each new key in tuples in $\triangle V$ is added, a necessary requirement for any solution. □

As opposed to Proposition 5 and Theorem 6, we show that the source side-effect problem is tractable for *SPU* and *SJU* views for single-tuple insertions. Putting these together with Theorems 13 and 14, we can see that group insertions complicate the source side-effect analysis.

Corollary 16. *The source side-effect problem is in PTIME for (a)* SPU *views and (b)* SJU *views, for single-tuple insertions.*

Proof: (a) *SPU* views. Consider an SPU query $Q = \bigcup_{i=1}^{k} \pi_{B_1,\ldots,B_m}(\sigma_{C_i}(\delta_{f_i}(R_i)))$, a database D, the view $Q(D)$, and view update $\triangle V$ consisting of a single tuple t. We present a PTIME algorithm that, given Q, D, $Q(D)$ and $\triangle V$, computes a minimum $\triangle D$ to produce $\triangle V$.

The algorithm first checks whether the only tuple t in $\triangle V$ is already present in the view $V = Q(D)$. If yes, then $\triangle D = \emptyset$. Otherwise, the algorithm proceeds as follows.

For $i \in [1, k]$, we invoke the PTIME algorithm for SP views given in the proof of Theorem 15 to check whether or not t can be inserted into $\pi_{B_1,\ldots,B_m}(\sigma_{C_i}(\delta_{f_i}(R_i)))$, by inserting a source tuple $\triangle D_i$ into base relation R_i. If the answer is yes for some $i \in [1, k]$, then it inserts a single tuple $\triangle D_i$ into the relation R_i. Otherwise, t cannot be inserted into V. Obviously, the algorithm is in PTIME, and $\triangle D$ contains at most one tuple (thus minimum).

(b) *SJU* views. Similarly, one can develop a PTIME algorithm to handle SJU views and single-tuple insertions. The algorithm leverages the PTIME algorithm for SJ views (Theorem 15), along the same lines as above. □

Finally, we show that single-tuple insertions and key preservation taken together make our lives much easier. Contrast this with Proposition 5, which shows that the view side-effect is intractable for (key preserving) views defined with join alone and single-tuple insertions.

Corollary 17. *The source side-effect problem is in PTIME for key preserving* SPJU *views, for single-tuple insertions.*

Proof: There exists a PTIME algorithm that, given a key preserving SPJU query Q, a database D, the view $Q(D)$, and view update $\triangle V$ with a single tuple as input, computes a minimum $\triangle D$ to produce $\triangle V$, if it exists. The algorithm is similar to the one for SPU views given in the proof of Corollary 16. Indeed, the only difference between the two is that here we invoke the PTIME algorithm for key preserving SPJ views (Theorem 15) for single insertions, rather than the one for SP views. □

5 The Annotation Placement Problem

In this section we investigate the annotation placement problem. Given a single location (field) l in a view tuple $\triangle V$, a source database D, an SPJU query Q and the view $V = Q(D)$, the problem is to identify a single field l' in a single tuple $\triangle D$ such that (a) if the value of the field l of $\triangle V$ is changed to a, then so is the field l' of $\triangle D$, and (b) the change a to l' is propagated to a smallest number of fields in the view. As stated in Section 1, this problem studies how an annotation attached to a location in the view is propagated backward to the source data.

In contrast to the view side-effect problem and the source side-effect problem (Sections 3 and 4), we do not need to consider group updates or insertions for the annotation placement problem. As a consequence this section presents a single result: the annotation placement problem is in PTIME for all subclasses of key preserving SPJU views. In contrast, it was shown in [1] that the problem is NP-*hard* for general PJ views, and it is in PTIME for SJU and SPU views.

Theorem 18. *The annotation placement problem is in PTIME for any subclass of key preserving* SPJU *views.*

Proof: It suffices to prove it for key preserving SPJU views. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a relational schema, D an instance of $\mathcal{R}$, and $Q = \cup_{i=1}^{k} Q_i$ an SPJU query, where $Q_i = \pi_{A_1,\ldots,A_m}(\sigma_{C_i}(\delta_{f_{i_1}}(R_{i_1}) \bowtie \cdots \bowtie \delta_{f_{i_{n_i}}}(R_{i_{n_i}}))$. We use (V, t, A_l) to denote a location in the view, *i.e.*, an annotation is attached to the A_l attribute of tuple $t \in V$ $(= Q(D))$. We develop a PTIME algorithm that, given D, Q, $Q(D)$ and (V, t, A_l) as input, finds a single tuple $\triangle D$ in D and a single location $(D, \triangle D, B_l)$ in $\triangle D$ such that an annotation in this field of $\triangle D$ propagates to a smallest number of tuples in V including (V, t, A_l).

We first show how we process each SPJ subquery Q_i, $i \in [1, k]$. We decompose t into $t_{i_1}, \ldots, t_{i_{n_i}}$ based on Q_i, where t_{i_j} is the attribute value vector (including key attributes) associated with the relation R_{i_j}. Utilizing the key preservation condition, the algorithm checks whether the following conditions are satisfied: (a) there is a tuple $t'_{i_j} \in R_{i_j}$ (unique if exists) with the same key-attribute values as t_{i_j} on R_{i_j} for each $j \in [1, n_i]$, (b) $t' = t'_{i_1} \bowtie \cdots \bowtie t'_{i_{n_i}}$ satisfies the selection condition C_i, and (c) $t = \pi_{A_1,\ldots,A_m}(t')$. If not, we know that t cannot be generated by subquery Q_i and thus $\triangle D_i = \emptyset$. Otherwise, for each t_{i_j}, if it contains attribute B_l and an annotation on B_l can be propagated to the specified field A_l in t, we check the other tuples in the entire view $Q(D)$ to compute the side effects generated by annotating B_l of t_{i_j}. After processing all t_{i_j}, $j \in [1, n_i]$, we can find the location with the minimum side effects for annotating a tuple in $Q_i(D)$, as well as its side effects S_i on the entire view $Q(D)$, where S_i is the set of view locations affected by annotating B_l. This can be done in PTIME.

By processing each Q_i as above, we find $\triangle D_i$ and S_i for all $i \in [1, k]$. We pick a nonempty $\triangle D_j$ such that S_j is minimum among all S_i's with nonempty $\triangle D_i$. Then $\triangle D_j$ is the location with minimum side effects. □

6 Conclusion

We have identified a practical condition, namely, the key preservation condition, which simplifies the propagation analysis of annotations. For key preserving views, we have shown that the annotation placement problem is tractable for all subclasses of SPJU queries, and that the view and source side-effect problems are in PTIME for SPJ views and single-tuple deletions, as opposed to NP-hard for general SPJ views [1], [13]. We have also investigated the impact of group updates on the complexity of the propagation analysis, and shown that group updates complicate the analysis: for group deletions the view and source side-effect problems become NP-hard for all subclasses of key preserving SPJU views that involve join operation. In addition, we have established the first complexity results for the analysis of view insertions for SPJU views, key preserving or not. These provide a complete picture of the complexity (intractability and tractability) of the annotation propagation analysis, which is useful in both data provenance and view-update processing.

We are currently studying approximation (heuristic) algorithms for conducting the propagation analysis when the associated problems are intractable. We also plan to identify other practical conditions on view definitions such that the analysis can be performed efficiently. Finally, we only considered lower bounds for the intractable cases. The identification of upper bounds is left for future work.

References

[1] P. Buneman, S. Khanna, and W. Tan, "On propagation of deletion and annotation through views," in *PODS*, 2002.

[2] P. Buneman, J. Cheney, W. Tan, and S. Vansummeren, "Curated databases," in *PODS*, 2008.

[3] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2009.

[4] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu, "Believe it or not: Adding belief annotations to databases," in *VLDB*, 2009.

[5] P. Buneman, S. Khanna, and W. Tan, "Why and where: A characterization of data provenance," in *ICDT*, 2001.

[6] B. Glavic and G. Alonso, "Provenance for nested subqueries," in *EDBT*, 2009.

[7] M. Eltabakh, W. G. Aref, A. Elmagarmid, M. Ouzzani, and Y. N. Silva, "Supporting annotations on relations," in *EDBT*, 2009.

[8] J. Huang, T. Chen, A. Doan, and J. F. Naughton, "On the provenance of non-answers to queries over extracted data," in *VLDB*, 2008.

[9] D. Bhagwat, L. Chiticariu, G. Vijayvargiya, and W. Tan, "An annotation management system for relational databases," *VLDB J.*, vol. 14, no. 4, pp. 373–396, 2005.

[10] Y. Cui, J. Widom, and J. Wiener, "Tracing the lineage of view data in a warehousing environment," *TODS*, vol. 25, no. 2, pp. 179–227, 2000.

[11] Y. Cui and J. Widom, "Run-time translation of view tuple deletions using data lineage," Technical Report, Stanford, 2001.

[12] E. Rahm and H. H. Do, "Data cleaning: Problems and current approches," *IEEE Data Engineering Bulletin*, vol. 23, no. 4, 2000.

[13] W. C. Tan, "Containment of relational queries with annotation propagation," in *DBPL*, 2003.

[14] S. Handschuh and S. S. (Editors), *Annotation for the Semantic Web (Frontiers in Artificial Intelligence and Applications)*. IOS Press, 2003.

[15] M. Agosti, N. Ferro, I. Frommholz, and U. Thie, "Annotations in digital libraries and collaboratories facets, models and usage," in *ICADL*, 2007.

[16] L. Chiticariu, W. Tan, and G. Vijayvargiya, "DBNotes: A post-it system for relational databases based on provenance," in *SIGMOD*, 2005.

[17] F. Geerts, A. Kementsietsidis, and D. Milano, "MONDRIAN: Annotating and querying databases through colors and blocks," in *ICDE*, 2006.

[18] INRIA Futurs, LRI, IN-SITU, "The infovis toolkit," *http://ivtk.sourceforge.net/*.

[19] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[20] F. Geerts, A. Kementsietsidis, and D. Milano, "*i*MONDRIAN: A visual tool to annotate and query scientific databases," in *EDBT*, 2006.

[21] G. Cong, W. Fan, and F. Geerts, "Annotation propagation revisited for key preserving view," in *CIKM*, 2006.

[22] F. Geerts and J. V. den Bussche, "Relational completeness of query languages for annotated databases," *JCSS*, to appear.

[23] B. Glavic and G. Alonso, "Perm: Processing provenance and data on the same data model through query rewriting," in *ICDE*, 2009.

[24] P. Buneman, J. Cheney, and S. Vansummeren, "On the expressiveness of implicit provenance in query and update languages," *TODS*, vol. 33, no. 4, pp. 1–47, 2008.

[25] T. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2005.

[26] Y. R. Wang and S. E. Madnick, "A polygen model for heterogeneous database systems: The source tagging perspective," in *VLDB*, 1990.

[27] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *SIGMOD*, 2006.

[28] U. Dayal and P. Bernstein, "On the correct translation of update operations on relational views," *TODS*, vol. 7, no. 3, 1982.

[29] A. Keller, "Algorithms for translating view updates to database updates for views involving selections, projections, and joins," in *PODS*, 1985.

[30] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar, "Tracing lineage beyond relational operators," in *VLDB*, 2007.

[31] T. Lee, S. Bressan, and S. E. Madnick, "Source attribution for querying against semi-structured documents," in *WAIM*, 1998.

[32] D. P. Groth and K. Streefkerk, "Provenance and annotation for visual exploration systems," *IEEE Trans. Visualization and Computer Graphics*, vol. 12, no. 6, pp. 1500–1510, 2006.

[33] M. Agosti and N. Ferro, "A formal model of annotations of digital content," *TOIS*, vol. 26, no. 1, 2007.

TABLE 1
Complexity of the view side-effect problem for deletion propagation

Query class	single deletion		group deletion	
	general view	key preserving view	general view	key preserving view
JU	NP-hard ([1])	NP-hard (Cor. 2)	NP-hard ([1])	NP-hard (Cor. 2)
PU	PTIME ([1])	PTIME (Cor. 3)		
PJ	NP-hard ([1])	PTIME (Prop. 1)	NP-hard ([1])	NP-hard (Thm. 4)
J	PTIME ([1])		NP-hard (Thm. 4)	

TABLE 2
Complexity of the view side-effect problem for insertion propagation

Query class	single insertion		group insertion	
	general view	key preserving view	general view	key preserving view
PU	NP-hard (Thm. 6)			
J	coNP-hard (Prop. 5)			
P, U	PTIME (Thm. 7)			

TABLE 3
Complexity of source the side-effect problem for deletion propagation

Query class	single deletion		group deletion	
	general view	key preserving view	general view	key preserving view
JU	NP-hard ([1])	NP-hard (Cor. 10)	NP-hard ([1])	NP-hard (Cor. 10)
PU	PTIME (Cor. 11)			
PJ	NP-hard ([1])	PTIME (Prop. 8)	NP-hard ([1])	NP-hard (Cor. 9)
J	PTIME ([1])		NP-hard (Cor. 9)	

TABLE 4
Complexity of the source side-effect problem for insertion propagation

Query class	single insertion		group insertion	
	general view	key preserving view	general view	key preserving view
PJU	NP-hard (Thm. 12)	PTIME (Thm. 17)	NP-hard (Thm. 12)	NP-hard (Thm. 13)
JU	PTIME (Cor. 16)		NP-hard (Thm. 13)	
PU			NP-hard (Thm. 14)	
PJ	NP-hard (Thm. 12)	PTIME (Thm. 15)	NP-hard (Thm. 12)	PTIME (Thm. 15)
U, P, J	PTIME (Thm. 15)			

[34] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage." in *CIDR*, 2005.

[35] A. Bohannon, B. Pierce, and J. A. Vaughan, "Relational lenses: A language for updateable views," in *PODS*, 2006.

[36] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom, "ULDBs: Databases with uncertainty and lineage," in *VLDB*, 2006.

[37] B. Choi, G. Cong, W. Fan, and S. Viglas, "Updating recursive XML views of relations," *Journal of Computer Science and Technology*, vol. 23, no. 4, pp. 516–537, 2008.

[38] S. S. Cosmadakis and C. H. Papadimitriou, "Updates of relational views," in *PODS*, 1983.

[39] J. Lechtenborger and G. Vossen, "On the computation of relational view complements," *TODS*, vol. 28, no. 2, pp. 175–208, 2003.

[40] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *TODS*, vol. 6, no. 4, pp. 557–575, 1981.

[41] *IBM DB2 Universal Database SQL Reference*, IBM, *http://www.ibm.com/software/data/db2/* .

[42] *SQL Reference*, Oracle, *http://www.oracle.com/technology/documentation/database10g.html*.

[43] *MSDN Library*, SQL server, *http://msdn.microsoft.com/library*.

[44] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman and Co., 1979.

[45] C. H. Papadimitriou, *Computational Complexity*. AW, 1994.

Gao Cong is an Assistant Professor at Nanyang Technological University, Singapore. He was an Assistant professor at Aalborg University, Denmark from 2008 to 2010. Before that, he worked as a researcher at Microsoft Research Asia, and as a postdoctoral research fellow at the University of Edinburgh. He received his PhD from the National University of Singapore. His current research interests include search and mining social media, and spatial keyword query processing.

Wenfei Fan is Professor of Web Data Management in the School of Informatics, University of Edinburgh, UK. He is also a National Professor of the Thousand-Talent Program and a Yangtze River Scholar at Harbin Institute of Technology, China. He received his PhD from the University of Pennsylvania, and his MS and BS from Peking University. He is a recipient of the Alberto O. Mendelzon Test-of-Time Award of ACM PODS 2010, the Roger Needham Award in 2008, the Outstanding Overseas Young Scholar Award in 2003, the Career Award in 2001, the ICDE Best Paper Award in 2007, and the Best Paper of the Year Award from Computer Networks in 2002. His current research interests include data quality, data integration, integrity constraints, distributed query processing, Web services and XML.

Floris Geerts received his PhD from Hasselt University, Belgium and is currently a Senior Research Fellow at the University of Edinburgh. His research interests include data quality, annotation management and database query languages.

Jianzhong Li is a professor and the chairman of the Department of Computer Science and Engineering at the Harbin Institute of Technology, China. He worked in the University of California at Berkeley as a visiting scholar in 1985. From 1986 to 1987 and from 1992 to 1993, he was a staff scientist in the Information Research Group at Lawrence Berkeley National Laboratory, Berkeley, USA. His current research interests include database management systems, data warehousing and data mining, sensor network, and data intensive super computing. He has authored three books, including Parallel Database Systems, Principle of Database Systems and Digital Library, and published more than 200 technical papers in refereed journals and conference proceedings. He is an associate editor of Knowledge and Data Engineering and refereed papers for varied journals and proceedings.

Jizhou Luo is an associate professor of the School of Computer Science and Technology, Harbin Institute of Technology, China, from which he received his PhD in 2006. His current research interests include query processing in sensor network, the design and analysis of algorithms in intensive data computing, and annotation management in RDBMS.