Updating Recursive XML Views of Relations

Byron Choi   Gao Cong   Wenfei Fan   Stratis D. Viglas
School of Informatics, University of Edinburgh
{v1bchoi@inf,wenfei@inf,gao.cong@,sviglas@inf}.ed.ac.uk

Abstract

This paper investigates the view update problem for XML views published from relational data. We consider XML views defined in terms of mappings directed by possibly recursive DTDs, compressed into DAGs and stored in relations. We provide new techniques to efficiently support XML view updates specified in terms of XPath expressions with recursion and complex filters. The interaction between XPath recursion and DAG compression of XML views makes the analysis of XML view updates rather intriguing. In addition, many issues are still open even for relational view updates, and need to be explored. In response to these, on the XML side, we revise the notion of side effects and update semantics based on the semantics of XML views, and present efficient algorithms to translate XML updates to relational view updates. On the relational side, we propose a mild condition on SPJ views, and show that under this condition the analysis of deletions on relational views becomes PTIME while the insertion analysis is NP-complete. We develop an efficient algorithm to process relational view deletions, and a heuristic algorithm to handle view insertions. Finally, we present an experimental study to verify the effectiveness of our techniques.

1. Introduction

As a classical technical problem, view updates have been studied for relational databases for decades (see, *e.g.*, [9, 11, 17, 23]), and techniques developed in that area have been introduced into commercial DBMSs [16, 25, 28]. Recently, a number of systems have been developed for publishing relational data to XML [1, 4, 12, 16, 25, 28]. The published XML documents can be seen as *XML views* of the relational data. For all the reasons that updating data through its relational views is needed, it is also important to update relational databases through their XML views.

In this paper we study the XML *view update problem*, which can be stated as follows. Given an XML view defined as a mapping $\sigma : \mathcal{R} \rightarrow D$ from relations of a schema $\mathcal{R}$ to XML documents (trees) of a DTD D, a relational instance I of R, the XML view $T = \sigma(I)$, and *updates* Δ_X *on the* XML *view* T, we want to compute *relational updates* Δ_R such that $\Delta_X(T) = \sigma(\Delta_R(I))$. That is, the relational updates Δ_R, when propagated to XML via the mapping σ, yield the desired XML updates Δ_X on the view T.

While several commercial systems [16, 25, 28] allow users to define XML views of relations, their support for XML view updates is either very restricted or not yet available. Previous work

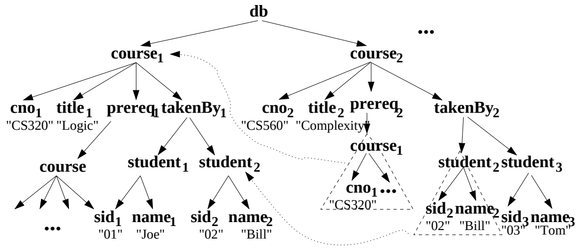


Figure 1: Example XML view

on XML view updates [2] has addressed the problem by translating XML view updates to relational view updates and delegate the problem to the relational DBMS; however, most commercial DBMSs only have limited view-update capability [16, 25, 28]. The state of the art in XML view updates research [30, 31, 32] solves the problem by explicitly focusing on *non*-recursively defined XML views and XML updates defined *without* recursive XPath queries. Though a complete solution, the restrictions posed in [32] are unfortunate since the recent proposals on XML update languages [22, 29] employ recursive XPath queries while DTDs (and thus XML view definitions) found in practice are often recursive [6]. In accordance to these requirements we advance the state of the art by supporting recursively defined XML views and recursive XPath update specifications. These requirements extend the side effects considered in [32], which we identify and address. In doing so, we provide an end-to-end (*i.e.*, from XML views to the underlying DBMS) solution to the problem and advance the theory of relational view updates.

We consider more general XML views and updates: possibly recursive XML view definitions and XML updates specified in terms of XPath expressions with recursion (descendant-or-self '//') and complex filters, as illustrated by the example below.

Example 1.1: Consider a *registrar* database I_0, which maintains *student* data, *enroll*ment records, *course* data and a relation *prereq*. It is specified by the relational schema R_0 (with keys underlined):

course(cno, title, dept),	student(ssn, name),
enroll(ssn, cno),	prereq(cno1, cno2),

where a tuple *(c1, c2)* in *prereq* indicates that *c2* is a prerequisite of *c1*. That is, *prereq* gives the prerequisite hierarchy of courses.

As depicted in Fig. 1 (the dotted lines will be illustrated shortly), from the relational database an XML view T_0 is published for the CS department by extracting CS course-registration data from I_0. The view is required to conform to the DTD D_0 below (the definition of elements whose type is PCDATA is omitted):

<!ELEMENT db (course*)>
<!ELEMENT course (cno, title, prereq, takenBy)>
<!ELEMENT prereq (course*)>
<!ELEMENT takenBy (student*)>
<!ELEMENT student (ssn, name)>

Note that the view is defined recursively since the DTD D_0 is recursive (*course* is defined indirectly in terms of itself via *prereq*). Now consider an XML update Δ_X = insert T' into P_0 posed on the XML view T_0, where P_0 is a (recursive) XPath query *course[cno=CS650]//course[cno=CS320]/prereq*, and T' is the subtree representing the course CS240. It is to find all the CS320 nodes below CS650 in T_0 and for each CS320 node v, insert T' as a prerequisite of v. To carry out Δ_X, we need to find updates Δ_R on the underlying database I_0 such that $\Delta_X(T_0) = \sigma_0(\Delta_R(I_0))$. □

Already a hard problem for relational views, the view update problem for XML views introduces several new challenges, which previous work [2, 30, 32, 31] on XML view updates cannot handle.

First, the notion of update *side effects* and update semantics should be revised in the context of XML views of relations. Referring to the example above, Δ_X asks for inserting CS240 as a *prereq* of *only* those CS320 nodes below CS650, whereas in reality, CS320 has a unique *prereq* hierarchy (published from the same relational records) and thus the insertion will result in side effects. In order to be consistent with the semantics of the XML view, we resolve the side effect problem by revising the insert semantics such that the insertion will be performed at *every* CS320 node. The effect of side effects on deletions is even more subtle and calls for a new semantics (see in Section 3.) Previous work [2, 30, 32, 31] did not consider the new side-effect issues of XML view updates on possibly recursive views.

Second, the XML view $\sigma(I_0)$ may be *compressed* by storing each subtree shared by multiple nodes in the tree *only once*, as indicated in Fig. 1 (replacing the subtrees in the dotted triangles by dotted edges). The need for this is evident: the compressed view becomes a directed acyclic graph (DAG), which is often significantly smaller than the original tree and may even lead to exponential savings in space. Furthermore, one may want to store the view (DAG) in *relations* itself. This raises another question: how should one define relational views that characterize the compressed XML view (DAG)? If one is to reduce the XML view update problem to its relational counterpart, this question has to be answered. However, this is nontrivial: the XML view is recursively defined, and a naïve relational encoding may require *infinitely many* relational views. Previous work [2, 30, 32, 31] did not consider the relational-view characterization of compressed and possibly recursively defined XML views.

Third, to locate where the updates take place, one has to evaluate the (recursive) XPath query P_0 embedded in Δ_X, on *DAGs* instead of XML trees. Added to the complication of the predefined DTD D_0 (resp. the XML view definition σ_0) being recursive, the interaction between recursion in XPath and recursion in the XML view definition makes it hard to translate XML view updates to relational (view) updates. As observed in [21], translation from (recursive) XPath queries (resp. updates) over recursive XML views (stored in relations) to SQL queries (resp. updates) is nontrivial. To our knowledge, no efficient algorithm has been published for evaluating XPath queries with *complex filters* on DAGs *stored in relations*.

While these are new issues beyond what we have encountered in relational view updates, automated processing of relational view updates is already intricate, even under various restrictions on the views [9, 11, 17]. In fact even the updatability problem, *i.e.*, the problem of determining whether a relational view is updatable w.r.t. given updates, is mostly unsolved and few complexity results are known about it [9, 3]. This tells us that it is unrealistic to reduce the XML view update problem to its relational counterpart and then rely on the DBMSs to do the rest.

Contributions. We propose new techniques for updating *compressed* and possibly *recursively* defined XML views via *schema-directed* XML *publishing*, in particular ATGs [1] [1]. We allow XML updates specified in terms of XPath expressions with *recursion* and *complex filters*. Given XML updates Δ_X on an XML view $T = \sigma(I)$, which is compressed into a DAG and stored in relations, we do the following. (a) We define relational views V that characterize the compressed XML view, such that the number of views in V is bounded by the size of σ even if σ is recursively defined. (b) We revise the notion of side effects of view updates based on the semantics of XML views, and provide an algorithm for translating Δ_X to *group updates* Δ_V on V while capturing the side effects of XML view updates. (c) We develop our own algorithms for processing relational view updates, and translate Δ_V to updates Δ_R on the underlying database I by means of our algorithms, such that $\Delta_X(T) = \sigma(\Delta_R(I))$ under the new semantics of XML view updates. If Δ_V or Δ_R does not exist, we detect and report it as early as possible. More specifically, we make the following contributions to the study of view updates in both XML and relational settings.

- *On the* XML *side.* (a) We refine the notion of side effects and the update semantics for XML views of relations, based on the semantics of XML views. (b) We develop an algorithm to translate (*recursive*) updates Δ_X on a (*possibly recursively defined*) XML view to updates Δ_V on the relational representation V of the XML view. (c) To do the translation, we present an efficient algorithm for evaluating XPath queries with *complex filters* on DAGs, based on a new indexing structure to handle recursion and a new technique for handling filters. (d) We also develop efficient algorithms to incrementally maintain the indexing structure.
- *On the relational side.* (a) We identify a *key-preservation* condition on SPJ views, which is less restrictive than the conditions imposed by previous work [9, 11, 17]. This condition does not reduce the expressive power of ATGs. (b) We establish complexity results for the updatability problem. We show that under key-preservation on SPJ views, while the problem for tuple insertions is NP-complete, it becomes *tractable* for *group* deletions (which is NP-complete without key preservation). (c) We propose a PTIME algorithm for processing group deletions on SPJ views. (d) To process group insertions we give an efficient heuristic algorithm.
- *Experimental study.* Our experimental results verify the effectiveness and efficiency of our techniques.

These techniques are the first for processing XML updates with *recursion and complex filters* on *compressed and possibly recursively defined* XML views, without relying on the high-end and mostly unavailable view-update functionality of the underlying relational DBMS. They provide the capability of supporting XML view updates within the immediate reach of most XML publishing systems. On the relational side, our complexity results and algorithms are a useful addition to the study of relational view updates.

Organization. Section 2 reviews ATGs and XML compression. Section 3 introduces relational views, characterizes DAG compression of XML views, defines XML updates, and refines the notion of side effects for XML view updates. Section 4 develops our indexing structure and algorithms for translating XML updates to relational view updates, and Section 5 presents our complexity results and algorithms for handling relational view updates. An experimental study is given in Section 6, followed by related work in Section 7 and future work in Section 8. Proofs are given in [13].

2. Schema-Directed XML View Definition

To study XML view updates we first fix an XML view definition language. We choose ATG [1] for its capability to recursively define XML views of relations. In this section we first review ATGs and then present a DAG compression of XML views.

[1] Our techniques are applicable to XML views published from relations via other systems (e.g., SilkRoute, XPERANTO) as long as they represent the XML views in terms of SPJ queries.

2.1 Attribute Translation Grammar

An Attribute Translation Grammar (ATG) is defined by annotating a DTD with SPJ queries. To present ATGs, we first review DTDs.

DTDs. Without loss of generality, we formalize a DTD D to be a triplet (E, P, r), where E is a finite set of *element types*; r is in E and is called the *root type*; P defines the element types: for each A in E, $P(A)$ is a regular expression of the form:

$$\alpha \ ::= \ PCDATA \mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \ldots + B_n \mid B^*$$

where ϵ is the empty word, B is a type in E (referred to as a *child type* of A), and '$+$', ',' and '$*$' denote disjunction, concatenation and the Kleene star, respectively (we use '$+$' instead of '|' to avoid confusion). We refer to $A \rightarrow P(A)$ as the *production* of A. A DTD is *recursive* if it has an element type that is defined (directly or indirectly) in terms of itself. As shown in [1] all DTDs can be converted to this form in linear time.

ATGs. We now briefly review the syntax and semantics of ATGs (see [1] for details). An ATG $\sigma : \mathcal{R} \rightarrow D$ specifies a mapping from instances of the source relational schema $\mathcal{R}$ to documents of the target DTD D as follows. (a) For each element type A of D, σ defines a semantic attribute $\$A$ whose value is a single relational tuple of a fixed arity and type; intuitively, $\$A$ controls the generation of A elements in the XML view, and is used to pass data downwards as the document is produced. (b) For each production $p = A \rightarrow \alpha$ in D and each type B in α, σ specifies a SPJ query, $rule(p)$, which extracts data from a relational database; using the data and $\$A$, it generates the B children of an A element and their $\$B$ values.

Given a relational database I with schema $\mathcal{R}$, the ATG σ is evaluated top-down starting at the root r of D. A partial tree T is initialized with a single node of type r, and this node is marked as a *bud* to be expanded. The tree T is then grown by repeatedly selecting a bud b of some element type A and evaluating the queries associated with A. More specifically, we find the production $p = A \rightarrow \alpha$ in D, and generate the children of b by evaluating $rule(p)$ and using the value of the attribute $\$A$ of b. Here $rule(p)$'s are defined and evaluated based on the form of α:

(1) If α is $B_1, \ldots, B_n$, then a node tagged B_i is created for each $i \in [1, n]$ as a child of b. The tuple value of $\$B_i$ associated with the new B_i child is determined by projection from $\$A$, *i.e.*,, $\$B_i = (\$A.a_i^1, \ldots, \$A.a_i^k)$ is in $rule(p)$ for $i \in [1, n]$, where a_i^j is a field of the tuple $\$A$.

(2) If α is $B_1 + \ldots + B_n$, then $rule(p)$ is defined by

case f($\$A$) of 1: $\$B_1 := \$A$, $\ldots$ n: $\$B_n := \$A$,

where f is a function that maps $\$A$ to natural numbers in $[1, n]$. That is, based on the conditional test, exactly one child, B_i, is created. The value of the parent attribute $\$A$ is passed down to that child. No B_j child is created if $i \neq j$.

(3) If α is B^*, then $rule(p)$ is defined by $\$B \leftarrow Q(\$A)$, where Q is a SPJ query over I, and it treats $\$A$ as a constant. For each *distinct* tuple t returned by $Q(\$A)$, a B child is generated, carrying t as the value of its $\$B$ attribute.

(4) If α is PCDATA, then the rule specifies formatting of the values of $\$B$ for presentation (string/PCDATA).

(5) If α is ϵ, then no $rule(p)$ is defined and no action is taken.

The *element* children of node b become new buds and are also processed. The process proceeds until the partial tree cannot be further expanded. The final XML tree does not expose attribute values $\$A$, which are used in the relational storage of the tree.

Example 2.1: The ATG σ_0 given in Fig. 2 defines the XML view described in Example 1.1. Given a *registrar* database I, σ_0 computes an XML view $\sigma_0(I)$ as follows. It first generates the root element (with tag *db*), and then evaluates query Q_1 to extract CS courses from I (case (3)). For each distinct tuple c in the output of Q_1, it generates a *course* child v_c of *db*, which is a bud carrying c as the value of its attribute $\$course$. The subtree of the bud v_c is then generated by using c (case (1) above). Specifically, it creates the *cno*, *title*, *prereq* and *takenBy* children of v_c, carrying the corresponding fields of c. It then creates a text node carrying *c.cno* as its PCDATA, as the child of the *cno* node (case (4)); similarly for *title*. It creates the children of the *prereq* node by evaluating Q_2 to find prerequisites of the course, and again for each tuple in the output of Q_2 it generates a *course* node; similarly it constructs the *takenBy* subtree by extracting *student* data via Q_3 (case (3)). Note that Q_2 and Q_3 take *c.cno* as a constant. Since *course* is *recursively defined*, the process proceeds until it reaches courses that do not have any prerequisites, *i.e.*, when Q_2 returns empty at the *prereq* children of those course nodes. When the computation terminates the ATG generates an XML view as shown in Fig. 1, which conforms to the DTD D_0 of Example 1.1. □

db → course*
$\$course \leftarrow Q_1$
Q_1: **select** distinct c.cno, c.title **from** course c
where c.dept = "CS"

course → cno, title, prereq, takenBy
$\$cno$ = $\$course$.cno, $\$title$ = $\$course$.title,
$\$prereq$ = $\$course$.cno, $\$takenBy$ = $\$course$.cno

prereq → course*
$\$course \leftarrow Q_2(\$prereq)$
$Q_2(c_1)$: **select** distinct c.cno, c.title **from** prereq p, course c
where p.cno1 = c_1 and p.cno2 = c.cno

takenBy → student*
$\$student \leftarrow Q_3(\$takenBy)$
$Q_3(c)$: **select** distinct s.ssn, s.name **from** enroll e, student s
where e.cno = c and e.ssn = s.ssn

Figure 2: Example ATG σ_0

Observe that an ATG $\sigma : \mathcal{R} \rightarrow D$ defines a *recursive* XML view if its embedded DTD D is recursive. As a result, given a database I of $\mathcal{R}$, the depth of the XML view $\sigma(I)$ is decided at run-time, rather than statically, by the database I following a data-driven semantics.

2.2 DAG Compression of XML Views

We next describe the DAG compression of XML views.

The subtree property. An XML view of a relational database is determined by the underlying relational data. In ATG this is reflecetd as the *subtree property*. More specifically, consider an ATG $\sigma : \mathcal{R} \rightarrow D$. For any database I of $\mathcal{R}$ and any type A of D, an A-element (subtree) T_A in the XML view $\sigma(I)$ is *uniquely determined* by the value of the semantic attribute $\$A$ at the root of T_A. Thus, the ATG in fact defines a function ST() such that, given an element type A and a value t of $\$A$, ST$(A, t)$ returns a subtree rooted at a node tagged A and carrying t as its attribute.

DAG compression. As noted in Section 1, a subtree ST$(A, \$A)$ may appear at multiple places in the XML view $\sigma(I)$. It is natural and more efficient to *compress* the XML tree by storing a *single copy* of ST$(A, \$A)$ no matter how many times it occurs in the XML view. This leads to a DAG representation of the XML view $\sigma(I)$. In Fig. 1, for example, *course*$_1$ and *student*$_2$ are shared subtrees (see the dashed lines). Note that the DAG is rooted: the root of $\sigma(I)$ is also *the root of the* DAG. This DAG *compression* of $\sigma(I)$ may be *exponentially* smaller than $\sigma(I)$ stored as a tree. In this paper we consider XML views compressed into DAGs.

3. View Updates Revisited in the XML Setting

In this section we define the XML updates studied in this paper, revise the notion of side effects of XML view updates, and provide relational views to characterize DAG compression of XML views. Finally, we outline our approach to processing XML view updates.

3.1 XML View Updates: Side Effects and Semantics

We first define XML view updates and their new semantics.

Syntax. Following [22, 29] we specify XML updates in terms of XPath expressions: (a) insert (A, t) into p, (b) delete p. Here A is an element type, and t is a tuple value of the same type as the semantic attribute $\$A$ of A. We use the value t of the semantic attribute $\$A$ of the root of a subtree $\text{ST}(A, t)$ to uniquely identify $\text{ST}(A, t)$, based on the subtree property mentioned earlier. We define p as an XPath expression:

$$
\begin{array}{lll}
p & ::= & \epsilon \ \mid \ A \ \mid \ * \ \mid \ // \ \mid \ p/p \ \mid \ p[q], \\
q & ::= & p \mid p = \text{`}s\text{'} \mid label() = A \ \mid \ q \wedge q \mid q \vee q \mid \neg q,
\end{array}
$$

where ϵ, A, $*$ and '/' denote the *self-axis*, a label (tag), a wildcard and the *child*-axis, and '//' stands for /*descendant-or-self::node()*/, respectively; q in $p[q]$ is called a *filter*, in which s is a constant (string value), and '$\wedge$', '$\vee$' and '$\neg$' denote conjunction, disjunction and negation, respectively. For //, we abbreviate $p_1/ //$ as $p_1//$ and $// /p_2$ as $//p_2$.

Side effects. Before we define the semantics of XML updates on views, we first study the side effects on XML view updates. Recall from Example 1.1 the update Δ_X, which is to change the subtrees (prerequisite hierarchy) of only those CS320 nodes below CS650. However, the subtree property of the XML view tells us that the subtree of a CS320 node is *uniquely determined* by the value of its semantic attribute $\$course$, which is determined by the same set of relational records for *all* CS320 nodes. As a result, *all* CS320 nodes must have the *same* subtree. In other words, changes incurred to the subtree of any CS320 node must also be reflected to *all* CS320 nodes, rather than only to those below CS650.

The side-effect issue is more subtle for deletions. As an example, consider delete *course[cno=CS650]/prereq/course[cno=CS320]* on the XML tree of Fig. 1. The deletion aims to remove course CS320 from the prerequisites of course CS650. Again the subtree property tells us that we should remove all CS320 nodes, but not only the CS320 node under the CS650 node. On the other hand, this cannot be simply done by removing all CS320 nodes physically as done in previous work on XML view updates [2, 30, 31, 32]: CS320 is itself an independent CS course and moreover, may be a prerequisite of other courses. For the delete operation to make sense, we need first to find all the *parents* of the nodes to be removed, *i.e.*, those *prereq* nodes under CS650 nodes, and then remove CS320 from the *children* list of only those *parent* nodes.

These suggest that we have to refine the notion of *side effects* to capture the semantics and the hierarchical nature of XML views. More specifically, if a change is to be made to the subtree $\text{ST}(A, t)$ of an A element with the tuple t as its semantic attribute $\$A$, the same change has to be made to the subtrees of *all* the A elements with the same semantic attribute t. While this is generally considered a "side effect" in the setting of relational view updates, it is necessarily the semantics of XML view updates.

The semantics of XML view updates. The semantics of XML views call for a new semantics of XML view updates *different* from that of updates on XML data [22, 29]. The semantics of the insert operation on XML views is described as follows. Given an XML view T with root r, (a) it finds the set of all *elements* reachable from r via p in T, denoted by $r[\![p]\!]$; (b) for each element v in $r[\![p]\!]$, it adds the new subtree $\text{ST}(A, t)$ as the rightmost child of v; and moreover, (c) for each element u that has the same type and semantic attribute value as v, it also adds $\text{ST}(A, t)$ as the rightmost child of u as required by the semantics of XML views.

The delete operation on XML views is carried out as follows: (a) it computes $r[\![p]\!]$; (b) for each node $v \in r[\![p]\!]$, it removes the subtree $\text{ST}(A, t)$ from the children list of the parent node u of v, where A is the type of v and t is the value of $\$A$ at v; and (c) for any node u' that has the *same type and semantic attribute value* as the *parent* u of v, it removes $\text{ST}(A, t)$ from the children list of u'.

Compared to the previous work [2, 30, 31, 32], we support XML view updates that (a) are defined with much richer XPath expressions with *recursion and complex filters*, (b) operates on (possibly) recursively defined XML views, and (c) possess a new semantics that capture *side effects* of XML view updates. To avoid side effects a brute-force solution is adopted in [2]: no elements are allowed to appear more than once in an XML view; "conditional translatable updates" address the issue in [32] albeit in a more restrictive setting in terms of expressiveness (no recursively defined XML views or recursive XPath expressions).

3.2 Relational Coding of Recursively Defined XML Views

Consider an ATG $\sigma : \mathcal{R} \to D$ that defines XML views of relational databases of $\mathcal{R}$. To reduce the update problem for XML views defined by σ to its relational counterpart, we define relational views $\mathcal{V}_\sigma$ to characterize σ. This is nontrivial: (a) σ is possibly recursively defined; on such views the encoding methods of previous work (*e.g.*, [2]) may lead to *infinitely* many relational views; (b) we consider DAG compressed XML views, *i.e.*, a DAG representation of $\sigma(I)$ *as opposed to trees* assumed in previous work. To this end we define $\mathcal{V}_\sigma$ by means of the edge relations of $\sigma(I)$ as follows.

(a) We assume a compact, unique value associated with each tuple value of semantic attribute $\$A$ in $\sigma(I)$. We abstract away the implementation of this identity value by assuming w.l.o.g. the existence of a Skolem function gen_id that, given the tuple value of $\$A$, computes id_A that is unique among all identities associated with *all* semantic attributes. We use gen_A to denote the set of the identities of all $\$A$ tuples, which is computed once.

(b) We encode an XML view definition σ in terms of $\mathcal{V}_\sigma$, a set of SPJ queries $Q_{edge\_A\_B}$ coding the edge relations of σ. More specifically, for each production $A \to P(A)$ in the DTD embedded in σ, and for each child type B in $P(A)$, we create a relation $edge\_A\_B$ with two columns, id_A and id_B. Consider productions of the form $A \to B^*$, where $\$B \leftarrow Q(\$A)$ is the associated query in σ. Then $edge\_A\_B$ is the set of pairs (ia, ib) such that $ia = gen\_id(a)$, $ib = gen\_id(b)$, where $a \in gen\_A$, $b \in Q(a)$. The definition of $Q_{edge\_A\_B}$ is similar for productions of other forms. One example of an edge-relation query derived from the σ_0 ATG of Fig. 2 is $Q_{edge\_prereq\_course}$:

select	gen_id(gp), gen_id(c.cno, c.title)
from	gen_prereq gp, prereq p, course c
where	p.cno1 = gp.cno and p.cno2 = c.cno

Observe the following about $\mathcal{V}_\sigma$. First, $\mathcal{V}_\sigma$ encodes the DAG *compression* of XML view $\sigma(I)$. Indeed, for any subtree $\text{ST}(A, \$A)$ in $\sigma(I)$, each edge (ia, ib) in $\text{ST}(A, \$A)$ is stored *only once* in a relation $edge\_A\_B$ no matter how many times $\text{ST}(A, \$A)$ (and thus the edge) appears in $\sigma(I)$. This is because the tuple (ia, ib) is uniquely determined by the semantic-attribute values of the corresponding nodes, which are the same in different occurrences of $\text{ST}(A, \$A)$. Second, each $Q_{edge\_A\_B}$ in $\mathcal{V}_\sigma$ is defined by a SPJ query. Thus $\mathcal{V}_\sigma$ consists of only SPJ *views*. Third, $\mathcal{V}_\sigma$ consists of a *bounded* number of *relational views* even if σ is *recursively* defined. Indeed, each $edge\_A\_B$ relation codes edges from A-nodes to B-nodes

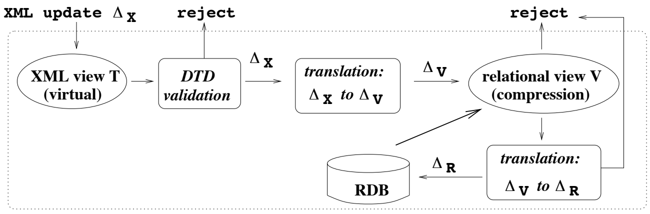


Figure 3: Overview of XML view updates

that may appear at an arbitrary depth of the tree, and the number of edge relations in $\mathcal{V}_\sigma$ is *bounded* by the size of the DTD D.

Updates on relational views. Given an update Δ_X on a DAG compressed XML view $\sigma(I)$, we propagate it to updates Δ_V on the relational view $V = \mathcal{V}_\sigma(I)$. The relational view updates Δ_V consist of edge tuples of the form $t = (ia, ib)$ to be inserted into or deleted from an edge relation $edge\_A\_B$.

The DAG compression of XML views also complicates the processing of view updates: (a) the XPath query embedded in an XML update has to be evaluated on a DAG rather than a tree; (b) a shared tree cannot be simply removed, as illustrated by the example below.

Consider again the delete operation on the XML view of Fig. 1, as described earlier. Suppose now the XML view is compressed into a DAG. We cannot simply remove the subtree of CS320 physically even if all CS320 nodes are in the *prereq* subtree of some CS650 nodes. This is because some subtrees inside CS320 (*i.e.,* certain *students*) may be shared and referenced by other nodes.

In response to this, we compute the relational view updates Δ_V such that (a) a newly inserted subtree is only stored once in V no matter how many times it appears in the updated view, and (b) a deleted subtree is not physically removed: only the tuple (ia, ib) in V representing the corresponding parent-child edge is deleted from its edge relation $edge\_A\_B$. More specifically, the tuple corresponding to ia is not removed from gen_A because ia is a parent node $v \in r[\![p]\!]$ and needs to be kept in the XML view. To cope with subtree sharing, ib is not removed from gen_B when the edge t is removed from $edge\_A\_B$; instead, upon the completion of processing Δ_V, our incremental maintainance algorithm runs in the *background* to remove tuples from gen_B's that are no longer linked to any node; it is at the completion of Δ_V when gen_B's are updated (similarly for insertions). Note that gen_B's are not defined as a view; they are derived from V (*i.e.,* the edge relations $\mathcal{V}_\sigma$) and maintained in the background.

3.3 Processing XML View Updates

We propose a framework for processing XML view updates, as shown in Fig. 3. For each ATG (XML view definition) $\sigma : \mathcal{R} \rightarrow D$, we maintain a relational database I of $\mathcal{R}$, and the relational views V that encode the DAG compression of $T = \sigma(I)$. The users pose updates on (the virtual view) T. Given a single XML update Δ_X on T as input, we are to generate a group update Δ_R on I such that $\Delta_X(T) = \sigma(\Delta_R(I))$ if such Δ_R exists; and otherwise *reject* Δ_X as early as possible. Specifically, the framework processes an XML update Δ_X on T in three phases, namely, DTD *validation*, *translation from* Δ_X *to* Δ_V, and *translation from* Δ_V *to* Δ_R. The DTD validation phase is simple and its discussion is deferred to [13].

From XML view updates to relational view updates. Given an update (insertion or deletion) Δ_X on the (virtual) XML view T, this phase translates Δ_X to a *group* relational view update Δ_V on V (See Section 4).

From relational view updates to base relation updates. Given a group update Δ_V on relational views V, this phase translates Δ_V to a group update Δ_R on the database I, if Δ_R exists; it rejects Δ_X otherwise. Instead of relying on the limited support for relational view updates of commercial DBMSs, in Section 5 we present an effective technique for processing relational view updates.

Conducting updates. After the relational update Δ_R is computed, we update the underlying database I using Δ_R, update the relational views V using Δ_V, and finally, *in the background,* invoke our incremental algorithm to maintain the indexing structures and to remove from gen_A those node ids that are no longer reachable from the root of the XML view T.

Input: the relational view V and topological order L.
Output: reachability matrix M.

```
M := ∅;
for(k := |L|; k > 0; k- -) /*process L from right to left */
    d := L[k];
    A_d := {a_2 | a_2 ∈ anc(a_1), a_1 ∈ parent(d) };
    insert (a, d) into M for each a ∈ A_d;
return M
```

Figure 4: Algorithm Reach

4. Mapping XML View Updates to Relations

In this section we present a technique for translating XML update Δ_X on an XML view T to updates Δ_V on relational views V, which represent the DAG compression of T. The technique consists of four parts: (a) indexing structures for checking ancestor-descendant relationships (Section 4.1), (b) an efficient algorithm for evaluating XPath queries on DAGs (Section 4.2), (c) algorithms to translate Δ_X to Δ_V (Section 4.3), based on the indexing structures and the evaluation algorithm, and (d) incremental algorithms for maintaining our indexing structures (Section 4.4).

4.1 Auxiliary Structures

To efficiently process '//' and filters on a DAG, we introduce two auxiliary structures: a topological order and a reachability matrix.

Topological order. Recall from Section 3 the function $gen\_id()$, which generates a unique id for each node based on its semantic-attribute value. Given a DAG stored in relations V, we create a list L consisting of all the distinct node identities in V topologically sorted such that u precedes v in L only if u is not an ancestor of v in the DAG, *i.e.,* there is no path from u to v in the DAG. As will be seen shortly, while based on L alone one cannot determine the ancestor-descendant relation, L is useful in evaluating XPath filters as well as in computing and maintaining the reachability matrix.

The list L can be computed in $O(|V|)$ time (see, *e.g.,* [8]), where $|V|$ is the size of the relational views. Its size, $|L|$, is the number of *distinct nodes* in the DAG, denoted by n. Note that L is computed once when V is created and it is maintained incrementally.

Reachability matrix. To identify the ancestor-descendant relationship between a pair of nodes in a DAG, we use an $n \times n$ *reachability matrix* $\mathcal{M}$: a cell in $\mathcal{M}$ is a bit. Given a row i denoting node n_i and a column j indicating node n_j, if cell $\mathcal{M}_{ij}$ is set, n_i is an ancestor of n_j in the XML view (or n_j is a descendant of n_i).

To store $\mathcal{M}$, we conceptually need as many bits as n^2. The cost for that is prohibitive. To overcome this, we store only information about the set bits of the reachability matrix. That is, $\mathcal{M}$ is physically stored as a table $M(anc, desc)$, where *anc* denotes an ancestor node, and *desc* a descendant. We use $\mathsf{desc}(a)$ (resp. $\mathsf{anc}(a)$) to denote the descendants (resp. ancestors) of node a retrieved from M.

Table M can be computed in $O(|V|^2 \log |V|)$ time from V (see, *e.g.,* [8]). Capitalizing on the topological order L we give Algorithm Reach, shown in Fig. 4, that computes M in $O(n\,|V|)$ time. It is based on dynamic programming: it ensures that for a node d the ancestors of the nodes in the set of parents of d, denoted by $\mathsf{parent}(d)$, are already known before we compute ancestors A_d, such that we can compute A_d by using those previously computed ancestors (lines 4-5). This can be achieved by processing the nodes in the order of L from right to left (line 2). Note that $\mathsf{parent}(d)$ can be computed from the edge relations in V.

To see that Algorithm Reach is in $O(n\ |V|)$ time, observe the following: (a) for each node in L we visit its parents once and thus any node v is visited $\mathsf{in}(v)$ times, where $\mathsf{in}(v)$ is the in-degree of v, *i.e.*, the number of incoming edges to v in the DAG; (b) the sum of $\mathsf{in}(v)$'s for all v is $|V|$; and (c) each visit takes at most $O(n)$ time. In practice, $|M| \ll n^2 \ll |V|^2$, where V is typically much smaller than the XML tree T, even up to an exponential factor.

We remark that L is very useful in maintaining M, and on the other hand M helps in maintaining L as to be shown in Section 4.4.

4.2 Evaluating XPath Queries on DAGs

To translate updates Δ_X on XML views to updates Δ_R on relational views, we have to evaluate the XPath expression embedded in Δ_X. The DAG compression of XML views introduces new challenges: previous work on XPath evaluation has mostly focused on trees rather than DAGs. While evaluation algorithms were developed for path queries on DAGs [5, 26], they cannot be applied here because (a) they do not deal with complex filters which, as will be seen shortly, require a separate pass of the input DAG, and (b) they do not address maintenance of the indexing structures the employ, which is necessary when the DAG is updated. Path-query evaluation algorithms were also developed for semi-structured data (general graphs). However, these algorithms neither treat DAGs differently from cyclic graphs (and thus may not be efficient when dealing with DAGs), nor consider XPath queries used in XML view updates.

To this end we outline an efficient algorithm for evaluating an XPath query p on an XML tree T that is (a) compressed as a DAG, and (b) stored in relations V. The algorithm takes as input an XPath query p over T, the relational views V, and the reachability matrix M. It computes (a) a set $r[\![p]\!]$ consisting of, for each node reached by p, a pair (B, v), where v is the id and B the type of the node respectively; and (b) a set $E_p(r)$ consisting of, for each v reached by p, tuples of the form $((C, u), v)$, where u is the id of a parent of v in the DAG (*i.e.*, there is an edge from u to v) such that p reaches v through u, and C is the type of u. We shall see that the set $E_p(r)$ is needed for handling deletions. Note that for each v there are possibly multiple (C, u) pairs, since we are dealing with a DAG (in which a node may have multiple parents) rather than a tree.

For XML data stored as a tree T, [19] developed an algorithm that evaluates an XPath query p in two passes (linear scans) of T. The basic idea of [19] is to first convert T to a binary-tree representation (before the two-pass process is invoked), and then run a bottom-up tree automaton on the binary tree to evaluate filters, followed by a run of a top-down tree automaton to identify nodes reached by p. It has linear-time complexity, the 'optimal' one can expect [19]. We next show that *a comparable complexity* can be achieved when evaluating XPath queries on a DAG stored in relations.

Our evaluation algorithm uses the following variables: (a) A list Q of filters including all the sub-expressions of filters in p, topologically sorted such that for any q_1, q_2 in Q, q_1 precedes q_2 if q_1 is a sub-expression of q_2. (b) For each q in Q and each node v in L, two Boolean variables $\mathsf{val}(q, v)$ and $\mathsf{desc}(q, v)$ to denote whether or not the filter q holds at v and at any descendant u of v, respectively.

Using these variables, we present a two-pass algorithm to evaluate p on V: a bottom-up phase that evaluates *filters* in p and computes the Boolean variables associated with each node v in L, followed by a top-down phase that computes $r[\![p]\!]$ and $E_p(r)$ using the filters computed. Due to lack of space we only outline the algorithm below.

Input: an insertion of the form Δ_X = insert (A, t) into p over T, and the relational view V.
Output: a group insertion Δ_V over V.

$\Delta_V := \emptyset$;
$E_A := \{\ ((B, gen\_id(\$u)), (C, gen\_id(\$v)))\ \mid\ (u, v)$ is an edge in $\mathrm{ST}(A, t)$, u, v with type B, C resp.$\}$;
r_A := the id of $\mathrm{ST}(A, t)$'s root as generated by $gen\_id(t)$;
for each $((B, ui), (C, vi)) \in E_A$
  $\Delta_V := \Delta_V \cup \{$ insert (ui, vi) into $edge\_B\_C\}$;
for each $(B, ui) \in r[\![p]\!]$
  $\Delta_V := \Delta_V \cup \{$ insert (ui, r_A) into $edge\_B\_A\}$;
return Δ_V;

Figure 5: Algorithm Xinsert

Bottom-up. The key idea is based on dynamic programming. For each node v in the topological order L, and for each sub-filter q in the topological order Q, we compute the values of $\mathsf{val}(q, v)$ and $\mathsf{desc}(q, v)$. This can be done by structural induction on the form of q. For example, when q is *label() = A*, $\mathsf{val}(q, v)$ is true if and only if v is in gen_A. When q is $q_1 \vee q_2$, $\mathsf{val}(q, v) := \mathsf{val}(q_1, v) \vee \mathsf{val}(q_2, v)$. When q is a path expression p, p can be rewritten into a "normal form" $\eta_1 / \ldots / \eta_n$, where each η_i is either (a) $\epsilon[q_i]$, (b) a label A, (c) wildcard '$*$', or (d) '//'. The normal form can be obtained in $O(|p|)$ time by capitalizing on the following rewrite rules: $p[q] \equiv p/\epsilon[q]$, and $\epsilon[q_1] \ldots [q_n] \equiv \epsilon[q_1 \wedge \ldots \wedge q_n]$. For example, if q is rewritten as $//\eta_2/ \ldots \eta_n$ with $\eta_1 = //$, $\mathsf{val}(q, v)$ is true if either $\mathsf{val}(\eta_2/ \ldots /\eta_n, v)$ or $\mathsf{desc}(\eta_2/ \ldots /\eta_n, u)$ is true for some child u of v; correspondingly, $\mathsf{desc}(q, v)$ is true if either $\mathsf{val}(q, v)$ or $\mathsf{desc}(q, u)$ holds. Note that the children of v can be efficiently identified by using the indexes on V. In addition, the algorithm proceeds in the topological orders L and Q. Therefore, the truth values of $\mathsf{val}(\eta_2/ \ldots /\eta_n, v)$ and $\mathsf{desc}(\eta_2/ \ldots /\eta_n, u)$ are already available before having to assign a value for $\mathsf{val}(q, v)$ and $\mathsf{desc}(q, v)$. Similarly $\mathsf{val}(q, v)$ can be computed for all other possible rewrites of q.

Top-down. Upon the completion of the bottom-up phase, we compute $r[\![p]\!]$ and $E_p(r)$ as follows. As mentioned earlier p can be normalized in the form of $\eta_1/ \ldots /\eta_n$, in which all the filters have already been evaluated to a truth value at each node satisfying p. Starting from the root r, we find nodes reached after each step η_i. These nodes can be easily found by using indexes on the edge relations V when η_i is A or $*$, and by means of the reachability matrix M when η_i is '//'. We now have all the information we need: upon the very last step η_n we accumulate all nodes reachable in that step into $r[\![p]\!]$, along with their types. Correspondingly, and whenever the last step leads to a node to be inserted in $r[\![p]\!]$ we accumulate the originating parent in $E_p(r)$ along with its type.

Complexity. In the bottom-up phase, each node v is visited at most $\mathsf{in}(v)$ times, where $\mathsf{in}(v)$ is the in-degree of v. In the top-down phase, each node is visited only once, except the final step when a node u may be included in $E_p(r)$ at most $\mathsf{out}(u)$ times, where $\mathsf{out}(u)$ is the out-degree of u. Putting these together, the complexity of the algorithm is $O(|p|\ |V|)$ time.

Compared to the algorithm of [19], observe the following. (a) When the DAG is a tree, our algorithm visits each node at most twice, *i.e.*, it has the same complexity as that of [19]. When dealing with DAGs that do not have a tree structure, it is necessary to visit all the edges in the DAGs in the worst case and thus our algorithm is optimal. (b) In contrast to [19], our algorithm does not require the conversion to binary trees and the construction of tree automata, which are potentially very large. (c) Our algorithm works on DAGs including but not limited to trees while [19] cannot work on DAGs.

4.3 Translating Updates from XML to Relations

On account of the relational representation (DAG) of XML views, a single XML update may be mapped to multiple relational updates (a group update) over the edge tables V. We next give two algorithms, Xinsert and Xdelete, for translating XML view insertions and deletions to relational view updates Δ_V, respectively.

Insertion. Algorithm Xinsert is presented in Fig. 5. Given Δ_X = insert (A, t) into p on the XML view T, the objective is to return the group of insertions Δ_V over V (which will then be tested for acceptance). The first step is to find the set of edges in the newly inserted subtree $\text{ST}(A, t)$ with the root r_A, which is computed by the algorithm of [1] and the function $gen\_id()$ (lines 2-3). We then generate the relational view updates: for each edge (ui, vi) in the newly inserted subtree, we add (ui, vi) to Δ_V (lines 4-5); moreover, for each $(B, ui) \in r[\![p]\!]$, we add (ui, r_A) as a new edge to Δ_V (lines 6-7). The set $r[\![p]\!]$ of nodes (pairs (B, ui) of node ids along with their types) reached by XPath p from the root of T (line 6) is computed using the evaluation algorithm of Section 4.2.

Deletion. Algorithm Xdelete is shown in Fig. 6. Given Δ_X = delete p, it returns the group of relation view deletions Δ_V over V, which will be passed to subsequent steps for acceptance test (Section 5.2). For each node vi in $r[\![p]\!]$ and each parent ui of vi in $E_p(r)$, it removes the edge (ui, vi) from V (lines 2-3). Here the parent-child relation is computed by using the set $E_p(r)$, whose computation is coupled with that of $r[\![p]\!]$ (See Section 4.2).

Observe that these algorithms implement *the new semantics* of XML view updates given in Section 3. This is achieved by leveraging the characterization of the XML view T in terms of relational views V. Indeed, for two edges $(u, v), (u', v)$ in T, if two parents u and u' of the same node v have the same element type A and the same value of the semantic attribute $\$A$, the two edges are represented by a *single* tuple in some edge relation $edge\_A\_B$. Thus there is no need to search V to find different nodes sharing (A, t), *i.e.*, XML side effects described in Section 3 do not incur extra cost. Furthermore, the set semantics of V ensures that a newly inserted subtree is stored *only once*. In addition, Algorithm Xdelete does *not* physically remove a deleted subtree; instead, only the corresponding parent-child edge is removed. These naturally comply to the requirements of DAG update semantics given in Section 3.

Example 4.1: Consider the XML update Δ_{X_1} = delete *//course [cno=CS320]//student[sid=S02]* on the XML tree in Fig. 1, which is to delete student S02 from the CS320 subtree. Given this as input, Algorithm Xdelete yields Δ_{V_1} = {(takeBy$_1$, student$_2$)}. As another example, given Δ_{X_2} = delete *//student[sid=S02]*, we get Δ_{V_2} ={(takeBy$_1$, student$_2$), (takeBy$_2$, student$_2$)}. □

Complexity. Algorithm Xinsert takes $O(|E_A| + |r[\![p]\!]|)$ time at most, which is the cost of inserting the "inner" connections of $\text{ST}(A, t)$ into V and connecting $\text{ST}(A, t)$ to the rest of V, where $|E_A|$ is the number of edges in $\text{ST}(A, t)$. Algorithm Xdelete takes $O(|E_p(r)|)$ time. Together with the complexity $O(|p|\ |V|)$ of evaluating p, this is the cost of generating Δ_V from Δ_X.

4.4 Maintenance of Auxiliary Structures

We next outline how to maintain the reachability matrix M and the topological order L in response to updates over V. We should remark that the maintenance of M and L is computed in the *background* in parallel with the processing of relational updates Δ_R; as a result, in our framework (Fig. 3), maintenance does not slow down the process of carrying out XML view updates.

The maintenance is nontrivial, as illustrated by the next example.

Example 4.2: Recall the XML update Δ_{X_1} from Example 4.1 This entails that all reachability information to S02 be deleted from the root of the CS320 subtree and from *all nodes* on the path to S02. Moreover, this course may be a prerequisite of other courses, *e.g.*,CS650; since CS320's subtree is shared, the reachability information from CS650 to S02 should be updated. □

Input: a deletion Δ_X = delete p over T and the rel. view V.
Output: a group deletion Δ_V over V

1. $\Delta_V := \emptyset$;
2. **for each** $((C, ui), vi) \in E_p(r)$, where $(B, vi) \in r[\![p]\!]$
3. $\quad\Delta_V := \Delta_V \cup \{$ delete (ui, vi) from $edge\_C\_B\}$;
4. **return** Δ_V;

Figure 6: Algorithm Xdelete

Recomputing M from the updated V bears a prohibitive cost. What we ideally would like is to *incrementally* update M. Existing incremental techniques [15, 18] for updading reachability information are not applicable since they rely on special auxiliary structures which are themselves expensive to construct and maintain (*e.g.*, [15] requires the computation of a spanning tree, taking $O(n\ |V|)$ time for each node insertion). On the other hand, incremental algorithms of updating topologically ordered lists (*e.g.*, [24]) take $O(|V|)$ time per edge insertion. Given these high individual complexities we follow a hybrid approach by maintaining both auxiliary structures at once.

Maitenance of auxiliary structures in response to XML view deletions takes place in the form of Algorithm $\Delta_{(M,L)}$delete, shown in Fig. 7. Due to space constraints we omit the maintenance algorithm $\Delta_{(M,L)}$insert for insertions, which can be found in [13]. The algorithm efficiently produces the following by scanning the elements of an XML deletion Δ_X: (a) deletions Δ_M over M, (b) an updated L, and (c) as an added bonus, the set of edges Δ'_V in the deleted subtree that are no longer connected to any nodes in the DAG and are to be passed to the garbage collector for *background* processing (see Section 3.) The set Δ'_V is a direct consequence of deletions Δ_V computed by Algorithm Xdelete. The need arises when a node $d \in \Delta_V$ is to be completely removed from the subtree. This happens when either all its incoming edges are in $E_p(r)$ (described in Section 4.2), or all its parent nodes are deleted.

The algorithm progresses by populating deletions Δ_M while, at the same time and whenever applicable, removing elements from L and populating Δ'_V. The first step is arranging all nodes in all deleted subtrees in a list L_R (line 2). To do so, we compute desc$(r[\![p]\!])$, *i.e.*, the descendants of all nodes in $r[\![p]\!]$; we then sort L_R according to L; this is always possible since $L_R \subseteq L$. For each node d in T we associate a state keep(d), initialized to true, and keeping track of whether the node should be ultimately deleted or not (line 3). L_R is then traversed backwards (line 4); this processing order of L_R ensures that each d in L_R is processed after its ancestors thus guaranteeing correct deletion semantics. For each d in L_R we compute its undeleted parents (lines 6-8) P_d (*i.e.*, any node a in its parent set for which keep(a) is true) and then its *new* ancestors A_d (line 9). If there is a node in d's current ancestors anc(d) that is not in A_d, it should be removed from M (lines 10-11). If d does not have any parents (*i.e.*, $P_d = \emptyset$) we set its keep state to false and delete it from L (lines 13-14). Observe that according to the semantics of L, an element removal does not affect the topological order of the rest of its elements. In addition, all outgoing edges from a deleted node d are deleted from V (lines 15-16); chidlren d' of d can be readily identified from d's type.

Example 4.3: Recall Δ_{X_1} from Example 4.1. Given Δ_{X_1}, Algorithm $\Delta_{M,L}$delete returns (1) $\Delta'_{V_1} = \emptyset$, (2) unchanged L, and (3) Δ_{M_1} = {(prereq$_2$, student$_2$), (prereq$_2$, sid$_2$), (prereq$_2$, name$_2$), ...}, *i.e.*, the reachability information from nodes prereq$_2$, course$_1$ and takenBy$_1$ to nodes in the S02 subtree, *i.e.*, nodes student$_2$,

Input: a deletion of the form Δ_X = $\mathsf{delete}\ p$ over T, the rel. view V, reachability matrix M and topological order L.
Output: deletions Δ'_V over V, Δ_M over M, and updated list L.

$\Delta'_V := \emptyset$; $\Delta_M := \emptyset$;
L_R := the sorted list $\mathsf{desc}(r[\![p]\!])$ according to topological order L;
$\mathsf{keep}(d) := \mathsf{true}$ for each $d \in T$; /*initialize state */
for each d in L_R *traversed backwards*
  $P_d := \emptyset$;
  for each $a \in \mathsf{parent}(d)$
    if $((C, a), d) \notin E_p(r)$ and $\mathsf{keep}(a) = \mathsf{true}$
    then $P_d := P_d \cup \{a\}$;
  $A_d := \{a_2 \mid a_2 \in \mathsf{anc}(a_1), a_1 \in P_d\}$;
  for each $a \in \mathsf{anc}(d) \setminus A_d$
    $\Delta_M := \Delta_M \cup \{\ \mathsf{delete}\ (a, d)\ \mathsf{from}\ M\}$;
  if $P_d = \emptyset$ /*compute Δ'_V and update L*/
  then $\mathsf{keep}(d) := \mathsf{false}$;
    delete d from list L;
    for any child d' (of type H) of d (of type G)
      $\Delta'_V := \Delta'_V \cup \{\ \mathsf{delete}\ (d, d')\ \mathsf{from}\ edge\_G\_H\}$;
return (Δ'_V, Δ_M, L)

Figure 7: Maintenance algorithm $\Delta_{(M,L)}$delete for deletions

sid_2 and name_2. Note that { $(\text{takeBy}_2, \text{student}_2)$, $(\text{takeBy}_2, \text{sid}_2)$, $(\text{takeBy}_2, \text{name}_2)$, ...}, *i.e.*, the connection between node takeBy_2 (and thus course_2) and the S02 subtree still holds and is not included in Δ_{M_1}. Given Δ_{X_2} in Example 4.1, Algorithm $\Delta_{M,L}$delete returns (1) Δ'_{V_2} ={$(\text{student}_2, \text{sid}_2)$, $(\text{student}_2, \text{name}_2)$}, (2) the new L by removing student_2, sid_2 and name_2 from the old L, and (3) Δ_{M_2} composed of the connections between nodes in the S02 subtree and all its ancestor nodes including db, course_1, takenBy_1, course_2, takenBy_2 and prereq_2. □

Complexity. The worst-case time complexity of the algorithm is $O(n\ |V|)$, which is the cost of computing new ancestors for nodes in L_R. For each node in L_R we visit its parents once, which in total takes at most $O(|V|)$ time (in practice it is much smaller than $|V|$); at each visit, the algorithm takes at most $O(n)$ time.

Observe the following: (a) The analysis given above is the worst-case complexity. In practice the updated XML view $\Delta_X(T)$ differs only slightly from the old view T, and the cost of maintaining M and L is much smaller than what worst-case complexity indicates. (b) As remarked earlier, all maintenance is conducted *in the background* and thus does not become a bottleneck. (c) As will be seen in Section 6, our experimental study verifies that the incremental approach is far more efficient than its batch counterpart.

5. Updating Relational Views

In this section we extend the study of relational view updates by providing complexity results (Proofs in [13]) and techniques for processing SPJ view updates under key preservation. These results are not only important for updating XML views defined in terms of ATGs, but are also useful for studying relational view updates.

5.1 Key Preservation and Relational View Updates

We propose a mild condition on SPJ views, and show that this condition simplifies the analysis of relational view updates.

Key preservation. Consider a SPJ query $Q(R_1, \ldots, R_k)$ that takes base relations $R_1, \ldots, R_k$ of $\mathcal{R}$ as input, and returns tuples of the schema $R(\vec{a})$. We say that Q is *key preserving* if for each R_i, the primary key of R_i is included in $\vec{a}$ (with possible renaming). That is, the primary keys of all the base relations involved in Q are included in the projection fields of (the SPJ query) Q.

Observe the following. First, key preservation is far less restrictive than other conditions proposed in earlier work for handling relational view updates (*e.g.,* [11, 17]; see Section 7). Second, every SPJ query in the definition of an ATG view σ can be made key-preserving by extending its projection-attribute list to include the primary keys. The extension does not affect the expressive power of ATGs. For example, Q_3 in σ_0 of Fig. 2 can be made key-preserving by adding *e.cno* to its select clause. Thus, in the sequel we assume w.l.o.g. that all the queries in ATGs are key-preserving.

Analysis. We consider the following decision problem:

PROBLEM:	SPJ View Updatability Problem
INPUT:	A collection of views $\mathcal{V}$ defined as SPJ queries *under key preservation*, a relational database I of schema $\mathcal{R}$, and a group view update Δ_V.
QUESTION:	Is there a group update Δ_R on the database I such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$?

Here Δ_V consists of either only tuple deletions or only tuple insertions, as produced by the translation algorithm of the last section. These deletions and insertions in Δ_V are translated to deletions and insertions in Δ_R, respectively. We use V to denote the view $\mathcal{V}(I)$.

It is known [3] that without key preservation, the updatability problem is already NP-hard for a single deletion and a single PJ view, *i.e.,* when Δ_V consists of a single deletion and $\mathcal{V}$ is a view defined with projection and join operators only. In contrast, we show that key preservation simplifies the updatability analysis for a collection of SPJ views and group deletions.

Theorem 5.1: *For group view deletions Δ_V, the* SPJ *view updatability problem is in PTIME.* □

However, the problem is intractable for insertions under key preservation; the lower bound can be verified by reduction from the non-tautology problem, which is NP-complete (cf. [14]).

Theorem 5.2: *The* SPJ *view updatability problem is NP-complete even when Δ_V has a single insertion and $\mathcal{V}$ has a single view.* □

These are the *first* complexity results for relational view updates under key preservation. In Section 5.2 we present a PTIME algorithm for computing database deletions Δ_R from view deletions Δ_V, which suffices to prove Theorem 5.1. In light of Theorem 5.2, we present a heuristic algorithm for computing database insertions Δ_R from view insertions Δ_V in Section 5.3.

5.2 Handling Group Deletions

We give a PTIME algorithm for computing database tuple deletions Δ_R from a group of view deletions Δ_V. Consider an instance of the view-tuple deletion problem: multiple views $\mathcal{V}$ defined in terms of SPJ queries under key preservation, a database I of schema $\mathcal{R}$, and a group view deletion Δ_V consisting of pairs (Q, t), which denote that the view tuple t is to be deleted from the view $Q(I)$ for some Q in $\mathcal{V}$ (note that the output of the algorithms in the last section can be expressed in this format). Assume that $\mathcal{R}$ consists of relation schemas $R_1, \ldots, R_k$, and I is $I_1, \ldots, I_k$. Each view Q in $\mathcal{V}$ is of the form $\pi_{\vec{a}}(\sigma_C(S_1 \times \ldots \times S_l))$, where $\vec{a}$ is a list of columns of $\mathcal{R}$, C is a *conjunctive condition*, and S_j is (a renaming of) some R_i. Note that the key preservation condition assures that $\vec{a}$ contains the primary key of S_j for $j \in [1, l]$. Given these, the algorithm is to find a collection Δ_R of tuples to be deleted from I such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$ if Δ_R exists; otherwise it rejects Δ_V, where $\Delta_V(\mathcal{V}(I))$ denotes $\mathcal{V}(I) \setminus \Delta_V$.

Let V_Q be the view $Q(I)$, and consider a tuple t in Δ_V that is to be deleted from V_Q. The key preservation condition allows us to identify, for each S_j, a *unique* tuple t_j via its key in t, such that $t_1, \ldots, t_l$ produce t via Q. Let us use $Sr(Q, t)$ to denote the set consisting of all the pairs (S_j, t_j), referred to as the *deletable source* of t in V_Q. Observe the following. (a) Deleting any t_j from

Input: a view definitions $\mathcal{V}$, a relational database I, the view $V_Q = Q(I)$ for each $Q \in \mathcal{V}$, and a group deletion Δ_V.
Output: a group update Δ_R on I if it exists.

1. $\Delta_R := \emptyset$;
2. **for each** (Q, t) in Δ_V
3.   compute $Sr(Q, t)$, the deletable source of t in V_Q;
4. **for each** Q' in $\mathcal{V}$ and **each** t in $V_{Q'}$ but not in Δ_V
5.   compute $Sr(Q', t')$;
6. **for each** (Q, t) in Δ_V
7.   **if** there exists (S_j, t_j) in $Sr(Q, t)$ such that (S_j, t_j) is not in $Sr(Q', t')$ for any Q' in $\mathcal{V}$ and any t' in $V_{Q'}$ but not in Δ_V
8.   **then** $\Delta_R := \Delta_R \cup \{(S_j, t_j)\}$;
9.   **else** reject Δ_V and **exit**;
10. **return** Δ_R

Figure 8: Algorithm delete

S_j suffices to remove t from V_Q. (b) Deletion of a source tuple t_j from V_Q is *side effect free* if and only if (S_j, t_j) is not in the deletable source of any tuple $t' \in \mathcal{V}(I) \setminus \Delta_V$ that is to remain in the view after Δ_V is carried out. From these one can see that t can be deleted from V_Q if and only if there exists $(S_j, t_j) \in Sr(Q, t)$ such that for all $Q' \in \mathcal{V}$ and all t' that are in $Q'(I)$ but not in Δ_V, (S_j, t_j) is not in $Sr(Q', t')$. Note that when *only the updatability problem* is concerned, deleting any of such t_j suffices, *i.e.,* one can choose an arbitrary t_j from $Sr(Q, t)$ satisfying the condition (b) given above, if there exists any.

Based on this we give Algorithm delete in Fig. 8. It first computes the deletable source $Sr(Q, t)$ for each view tuple t in Δ_V and each tuple that is in $\mathcal{V}(I)$ but not in Δ_V (lines 2-5). It then checks, for each (Q, t) in Δ_V, whether or not there is a source tuple in $Sr(Q, t)$ that can be deleted without violating condition (b) given above, and if so it updates Δ_R; it rejects Δ_V otherwise (lines 6-9). It returns Δ_R if all view tuples in Δ_V can be deleted without side effects (line 10). One can verify that the view V can be updated by Δ_V if and only if such a Δ_R exists.

Complexity. Observe that $Sr(Q, t)$ can be computed in $O(|Q|)$ time; the size of $Sr(Q, t)$ is bounded by $O(|Q|)$. Checking the side-effect free condition (line 7) takes at most $O(|\mathcal{V}(I)| - |\Delta_V|)$ time even if no indexes on I are used, while the worst-case data complexity of Algorithm delete is in $O(|\Delta_V|(|\mathcal{V}(I)| - |\Delta_V|))$ time. Note that we focus on data complexity in this section (*i.e.,* ignoring the view size), since the evaluation of a SPJ query $Q(I)$ may already take exponential time when the combined complexity is considered, *e.g.,* when $Q = R \times \ldots \times R$ for n Cartesian products.

Minimal deletions. The focus of Algorithm delete is to solve the updatability problem, *i.e.,* whether or not there exists Δ_R such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$. It does not address, however, which Δ_R to select if multiple valid Δ_R's exist. In the presence of multiple Δ_R's it is natural for one to choose the *smallest* set Δ_R of tuples to delete, *i.e.,* a set Δ_R such that $|\Delta_R|$ is the smallest. The *minimal view deletion problem* is thus to find, given a collection $\mathcal{V}$ of view definitions, a database I and view deletions Δ_V, the smallest set of tuple deletions Δ_R such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$.

However desirable, the minimal view deletion problem is intractable, even under the key preservation condition. The lower bound can be verified by reduction from the minimal set cover problem, which is known to be NP-complete (cf. [14]).

Theorem 5.3: *For* SPJ *views under key preservation, the minimal view deletion problem is NP-complete.* □

5.3 Processing Group Insertions

Theorem 5.2 tells us that any practical algorithm for handling group view insertions is necessarily heuristic. We approach this by reducing the SPJ view insertion problem to SAT, one of the most studied NP-complete problems. This allows us to leverage a well-developed SAT solver [27] to efficiently compute Δ_R if it exists.

An instance of SAT (cf. [14]) is $\phi = \bigwedge_{i \in [1,n]} C_i$, where C_i is a disjunction of literals, *i.e.,* propositional variables or their negation. It is to find a truth assignment μ that satisfies ϕ, if such a μ exists.

Below we outline our heuristic algorithm, referred to as Algorithm insert. The algorithm takes the same input as that of Algorithm delete given in Fig. 8, namely, $\mathcal{V}$, I, $V_Q(I)$ for each $Q \in \mathcal{V}$, and Δ_V, except that tuples in Δ_V are to be inserted into the views. It either finds a set of insertions D_R such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$, or it rejects Δ_V. It does the following:

- Compute a propositional logic formula ϕ (*i.e.,* a SAT instance) from $\mathcal{V}$, I, $V_Q(I)$'s, and Δ_V, such that ϕ is satisfiable if and only if there exists D_R such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$.
- Utilize an existing heuristic tool [27] for SAT to process ϕ.
- If the tool returns a truth assignment μ that satisfies ϕ, compute Δ_R from μ; otherwise reject the view updates Δ_V as well as Δ_X.

We next illustrate each of the three steps.

Deriving ϕ. The encoding is a little involved. It takes four steps.

First, we derive tuples that have to be present in base relations so that Δ_V can be computed through queries in $\mathcal{V}$. Consider (Q, t) in Δ_V, which indicates that tuple t is to be inserted into the view $Q(I)$, as illustrated in Section 5.2. For each t and each relation R_i involved in Q, we derive an R_i *tuple template* $t_i = (\vec{a_i}, \vec{b_i}, \vec{z_i})$ from t and Q, where $\vec{a_i}$ corresponds to the (primary) key of R_i, $\vec{b_i}$ to the other columns of R_i whose values can be determined from t, and $\vec{z_i}$ to variables whose values are unknown. Note that $\vec{a_i}$ is known due to the key preservation condition. If there is no tuple t' in the instance I_i of R_i with the key $\vec{a_i}$, we add t_i to a set X_i. Note that no more than $|Q|\,|\Delta_V|$ many tuple templates are in these X_i's.

Example 5.1: Consider two relations R_1, R_2 and a SPJ view Q given below, where keys are underlined:

R_1 = (<u>A: int</u>, B: bool), R_2 = (<u>C: int</u>, D: bool),
Q = $\pi_{A,C}\ (\sigma_{B=D}(R_1 \times R_2))$.

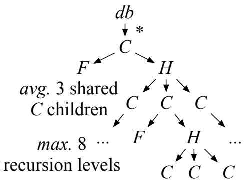
Suppose that tuples (a, c) and (a, c') are to be inserted into $Q(I)$. Then X_1 contains a tuple template (a, x_1) and X_2 contains (c, x_2) and (c', x_3), if no tuple bearing the key a is already in I_1 and no c, c' tuples are in I_2. For $(a, c), (a, c')$ to be inserted into the view, it is necessary that (a, x_1) is inserted into I_1 after x_1 is instantiated to a truth value, and that $(c, x_2), (c', x_3)$ are added to I_2. □

Second, we "evaluate" each view query Q on the database I incremented by adding X_i to I_i. Due to lack of space we defer the detail of the evaluation to [13]. In the evaluation we "instantiate" variables in the tuple templates, as well as the selection (conjunctive) condition in Q. In Example 5.1, for instance, the evaluation yields view tuples (a, c) with condition $x_1 = x_2$, and (a, c') with condition $x_1 = x_3$. We then inspect the result of Q to determine whether or not tuple templates may yield side effects. Specifically, for each tuple t in the result, if it is in neither the view nor Δ_V, we consider the following cases.

(a) If t is not associated with any condition, *i.e.,* it certainly has side effect, then we *reject* the view updates Δ_V and Δ_X immediately.

(b) If t has a condition in which at least one variable represents an attribute with an infinite domain, we can always pick a distinct value for the variable that makes the condition false. This eliminates t from the result and thus t does not yield a side effect.

(c) If t has a condition ϕ_t in which all variables correspond to attributes with a finite domain, we add the negation $\neg\phi_t$ as a conjunct to the logic formula ϕ that we are constructing.



(a) XML view

$\|C\|$	DAG	Tree	$\|L\|$	$\|M\|$
1K	25K	36.6K	25K	88K
10K	251K	366K	251K	900K
100K	2.5M	3.7M	2.5M	9.64M
1M	25.1M	36.6M	25.1M	102M

(b) Statistics of the datasets

Figure 9: Description of the datasets

Furthermore, for each t that is in Δ_V, we also add its associated condition ϕ_t as a conjunct to ϕ. Observe that these conjuncts are bounded by $|\Delta_V|$, and those in case (c) involve only attributes with a finite domain (with a fixed cardinality, a *constant*).

Example 5.2: Referring to Example 5.1, the conjuncts added to ϕ in the second step are $x_1 = x_2$ and $x_1 = x_3$. □

Third, to complete the construction of ϕ, for each variable x bounded to a finite domain, we add the following formula to ϕ as a conjunct: $x = c_1 \vee \ldots \vee x = c_k$, where $c_1, \ldots, c_k$ are all the values in that domain. In Example 5.1, for instance, we add $x_i = \mathsf{true} \vee x_i = \mathsf{false}$ for $i \in [1, 3]$.

Finally, we convert ϕ to a propositional formula (*i.e.*, a SAT instance). We use propositional variables and their negation to code variables introduced in the encoding: p for $x = c$ and $\bar{p}$ for $y \neq c$. We also add conjuncts $(\bar{p} \vee \bar{p'})$ to ensure that p and p' cannot be both true if, *e.g.*, p codes for $x = c$, p' for $x = c'$, and $c \neq c'$.

The correctness of the reduction is ensured by the following.

Theorem 5.4: *If Δ_V is not rejected during the coding, then ϕ is satisfiable iff there is Δ_R such that $\Delta_V(Q(I)) = Q(\Delta_R(I))$.* □

Processing ϕ. We invoke Walksat [27] with ϕ as the input. Walksat, an extension of GSAT, employs an efficient approximation algorithm to solve the maximum satisfiability problem. If ϕ is satisfiable, it finds a truth assignment μ for ϕ above a certain percentage.

Computing Δ_R. If μ is found, we derive Δ_R, *i.e.*, the set of tuples to be inserted into each I_i, by instantiating variables in the tuple templates in X_i's based on μ and the interpretation of propositional variables given above. More specifically, for each tuple template t in X_i, we assign a value to each variable z in t based on μ: if z is bounded in ϕ by $(z = c)$ for some constant c and $(z = c) \leftrightarrow x$, then we let $z = c$ if $\mu(x)$ is true. After this process if z is not assigned any value, then either (a) z ranges over an infinite domain and thus we can always pick a value c' for z that is not in the active domain of the database, or (b) the value of z does not have any impact on the satisfaction of ϕ; in both cases we can find a value for z without violating ϕ. Then Δ_R consists of query templates instantiated by these values.

If μ is not found, we reject Δ_V and Δ_X. Note that Walksat [27] may not find a truth assignment for ϕ even if ϕ is satisfiable, since SAT is intractable and so is the view insertion updatability problem (Theorem 5.2). However, this only happens within a certain percentage given the excellent performance of Walksat [20].

Complexity. From the construction of ϕ one can see that its size $|\phi|$ depends on $|\Delta_V|, \mathcal{R}$ and $|Q|$ only, whereas the size of the database I is irrelevant. Our algorithm has a low (data) complexity, and is effective in practice as verified by our experimental study.

6. Experimental Study

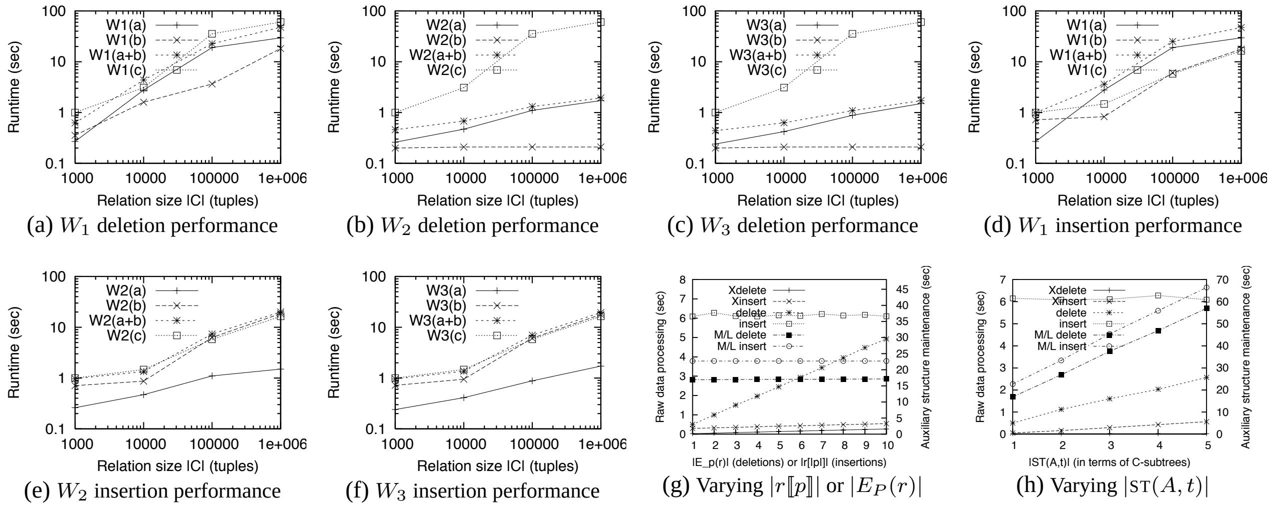
We conducted a preliminary experimental study of our proposed view update mechanism in order to verify its effectiveness. Our experiments were conducted on a Linux box running Redhat 9 and a commercial DBMS. The CPU was a 1.8Hz Pentium 4, while the machine had 2GB of physical memory; of those, 1GB was used as the buffer pool of the DBMS. The reported numbers are warm numbers and are the average of five runs per query. Reporting warm numbers is reasonable in this application context, as we can expect publishing systems to be continuously online and caching to take place. The standard deviation of the reported numbers is 5%.

All experiments were conducted on a synthetic dataset. This allows us to produce highly nested XML views with diverse structure and to have more control over the experimental settings.The dataset consists of four base relations: $C(\underline{c_1}, \cdots, c_{16})$, $F(\underline{f_1}, \cdots, f_{16})$, $H(\underline{h_1, h_2})$ and $C_U(\underline{c'_1}, \cdots, c'_{16})$, where underlined attributes indicate keys. The domain of f_1 is equal to the domain of c_1 and c'_1. The remaining C and F attributes were used to control how many joining C and F tuples were filtered out. The domains of h_1 and h_2 are the same as the domain of c_1. The generator ensured that (1) for each $c \in C \cup C_U$ there would be on average three tuples $h \in H$, where $c_1{=}h_1$, and (2) $h_1{<}h_2$, where $(h_1, h_2) \in H$. The universe of C, namely C_U, consisted of 100M C-tuples, ensuring that whenever h_2 joined with c_1 a C-tuple was always output. The sizes of F and H were proportional to the size of C, which we use for reporting the size of the synthetic database; specifically, the size we report is $|C|$, which ranges from 1,000 to 1,000,000 tuples, while $|F| = |C|$ and $|H| \simeq 3|C|$. We defined an ATG view of the relations C, F and H; as indicated in Fig. 9(a), the C nodes in the view were recursively defined, and a recursion of C in the view can be understood as $\pi_{c_1,f_1,h_1,h_2}(\sigma_{c_1=f_1 \wedge f_1=h_1 \wedge h_2=c'_1 \wedge c_2=f_2 \wedge c_3=f_3 \wedge c_4=f_4}(C \times F \times H \times C_U))$. Recall that [2, 30] cannot handle recursions of C in the view. Compression was achieved by sharing C subtrees, while dataset subtree sharing accounted for nearly 31.4% of C instances. Figure 9(b) lists some statistics on the number of published C subtrees, their compressed DAGs, and the corresponding sizes of the reachability matrix M and topological order L.

Varying database size. We generated two random update workloads over the XML view, one for insertions, and one for deletions; each workload consisted of three update classes, each class including ten operations. The classes were characterized by the XPath queries used to define the updates. Specifically, class W_1 involved XPath queries using '//' and value-based filters; XPath queries in W_2 used '/' and value-based filters; finally, W_3 contained XPath queries with '/', and both structural and value filters. The times we report include the following: (a) the time to evaluate XPath queries (Section 4.2); (b) the time to translate Δ_X to Δ_V (Algorithms Xinsert and Xdelete) and subsequently Δ_V to Δ_R (Section 5), and the time to execute the update; and (c) the time to maintain the auxiliary structures (Algorithms $\Delta_{(M,L)}\mathsf{insert}$, which can be found in [13], and $\Delta_{(M,L)}\mathsf{delete}$). Note that (c) is executed in the *background*.

Figures 10(a), 10(b) and 10(c) show the performance of the deletion algorithms for W_1, W_2 and W_3, respectively. We plot the runtime of performing the updates broken into their (a), (b) and (c) above constituents for various relational database sizes. Note that both x- and y-axes use a logarithmic scale. As shown, the algorithms scale linearly with the size of the relational database. It is evident that deletion time is dominated by XPath evaluation. Observe that although the cost for (c) is relatively high, it is performed in the background. $W_1(b)$ is the highest reported time among the three workloads since its XPath queries generate more edges (*i.e.*, $E_p(r)$), which are then examined by Algorithm delete.

Similar results are reported for insertions, as shown in Figures 10(d), 10(e) and 10(f) for W_1, W_2 and W_3, respectively



(a) W_1 deletion performance (b) W_2 deletion performance (c) W_3 deletion performance (d) W_1 insertion performance

(e) W_2 insertion performance (f) W_3 insertion performance (g) Varying $|r[\![p]\!]|$ or $|E_P(r)|$ (h) Varying $|\mathrm{ST}(A,t)|$

Figure 10: Update performance as a function of the size of the underlying relational database and the view update size

(again, using logarithmic scales). The size of the inserted subtree was fixed. The SAT solver [27] we used returned a truth assignment in 78% of the cases and we only report the time for insertions where the SAT solver successfully returned a truth assignment. As in the case of deletions, our insertion algorithms also scale linearly with the size of the database.

Varying update size. We then fixed $|C|$ to be 100K tuples. Figure 10(g) shows the performance of each algorithm as we varied $|E_p(r)|$ (see Section 4.2) for deletions and $|r[\![p]\!]|$ for insertions, while keeping $st(A, t)$ a constant single C-subtree. The runtimes for Algorithms Xinsert, Xdelete, delete and insert are measured on the left y-axis, while the runtimes for algorithms $\Delta_{(M,L)}$insert and $\Delta_{(M,L)}$delete are measured on the right y-axis. As expected, the translation time from Δ_X to Δ_V for Algorithm Xinsert (resp. Algorithm Xdelete) increases slightly as $|r[\![p]\!]|$ (resp. $|E_p(r)|$) increases. The slope of the curve for Algorithm delete is large, as the increase of $|E_p(r)|$ involves more database queries to determine the source tuples to be deleted. The performance of Algorithm insert, which models the translation of Δ_V to Δ_R for insertion workloads, is dominated by the coding time. As $|C|$ is far larger than $|\mathrm{ST}(A,t)|$ and $|r[\![p]\!]|$, and the number of database queries required remained fixed, the coding time remains roughly constant, though the size of the resulting coding increases; however, that only results in a non-observable increase in the SAT solver's runtime keeping the curve relatively flat. The performance of Algorithm $\Delta_{(M,L)}$insert (See [13]) and Algorithm $\Delta_{(M,L)}$delete is almost unaffected by $|r[\![p]\!]|$ (resp. $|E_p(r)|$) since $|\mathrm{ST}(A,t)|$ is fixed.

Similar results are shown in Fig. 10(h) where we varied the size of $|\mathrm{ST}(A,t)|$ while fixing $|E_p(r)| = 1$ and $|r[\![p]\!]| = 1$. The performance of Algorithm Xdelete remains unchanged and its runtime is negligible as it nearly overlaps with the x-axis for a fixed $|E_p(r)|$. Algorithm Xinsert scales linearly with the update size $|\mathrm{ST}(A,t)|$ as it needs to process $\mathrm{ST}(A,t)$ to generate Δ_V. Algorithms $\Delta_{(M,L)}$insert and $\Delta_{(M,L)}$delete evidently scale linearly w.r.t. the update size for reasons similar to the ones outlined earlier.

Effectiveness of incremental maintenance. The cost of incrementally maintaining the reachability matrix M and the topological order L as opposed to recomputing them is shown in Table 1. The first column presents the size of the relational datasets. The total time needed for incrementally maintaining both auxiliary structures is given in the second column for Algorithm $\Delta_{(M,L)}$insert (given in [13]) and in the third column for Algorithm $\Delta_{(M,L)}$delete.

Sizes	**Incremental (Sec.)**		**Recomputation (Sec.)**	
$\lvert C\rvert$	**Insertion**	**Deletion**	L	M
1K	1.0	1.0	6.3	9.8
10K	4.6	3.1	86	288
100K	22.7	16.9	631	3,600
1M	84.2	61.5	8611	14,000

Table 1: Incremental maintenance of L and M *vs.* recomputation

The time for recomputing each structure is shown in the last two columns. As expected, the advantages of incremental maintenance become more prominent as the size of the data increases.

7. Related Work

Commercial database systems [16, 25, 28] provide support for defining XML views of relations and restricted view updates. IBM DB2 XML Extender [16] supports only propagation of updates from relations to XML but not vice-versa. Oracle XML DB [25] does not allow updates on XML (XMLType) views. In SQL Server [28], users are allowed to specify the "before" and "after" XML views using *updategram* instead of update statements; the system then computes the difference and generates SQL update statements. The views supported are very restricted: only key-foreign key joins are allowed; neither recursive views nor updates defined in terms of recursive XPath expressions are supported.

There have been recent studies on updating XML views published from relational data [2, 30, 32]. In [2], XML views are defined as *query trees* and are mapped to relational views. XML view updates are translated to relations only if XML views are well-nested (*i.e.*, key-foreign key joins), and if the query tree is restricted to avoid duplication. [30] requires a *round-trip* mapping that shreds XML data into relations in order to ensure that XML views are always updatable. A detailed analysis on deciding whether or not an update on XML views is translatable to relational updates, along with detection algorithms, are provided in [32]. A framework for [32] is presented in [31]. The limitations of previous work [2, 30, 31, 32] are discussed in Section 1.

There has been a host of work ([9, 10, 11, 16, 17, 23, 25, 28]) on relational view updates. [11] provides algorithms for translating restricted view updates to base-table updates without side effects in the presence of certain functional dependencies. The algorithm in [17] handles translation (with side effects) for a restricted class of SPJ view: 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 is less restrictive than those in [11, 17]. There has also been work ([9, 23]) on relational view complements. However, finding a minimal view complement is NP-complete [9]. An algorithm for deletion translation is given in [10], which is very different from Algorithm delete of Fig. 8. Commercial DBMSs [16, 25, 28] allow updates on very restricted views (while users may specify updates manually with INSTEAD OF triggers). For example, for views to be deletable IBM DB2 [16] restricts the FROM clause to reference only one base table.

Few complexity bounds are known for (relational) view updates. The complexity of view complement computation is analyzed in [9, 23], and the complexity of deletion on views is given in [3]. To our knowledge, our work is the first to establish complexity bounds for both deletion and insertion on views under key preservation.

A number of XPath evaluation algorithms have been proposed (*e.g.,* [7, 19, 5, 26]). Except [5, 26], these techniques, however, are developed for trees and cannot answer XPath queries on DAGs. As mentioned in Section 4, our evaluation algorithm is inspired by [19], but differs from it in that we use dynamic programming based on indexing structures instead of converting XML data to binary trees and constructing (potentially expensive) tree automata. Path query evaluation has been studied in [5, 26] for DAGs. [5] extends stack-based algorithms to evaluate path-pattern queries on DAGs. Their algorithms cannot be directly used in the context of XML view updates because (a) pattern queries of [5] do not allow complex filters (*e.g.,* Boolean operations and nested filters) and thus cannot express XPath expressions embedded in XML updates considered here; (b) the maintenance method for their indexing structures is not yet in place, which is necessary in the study of XML updates. [26] explores the use of reachability information by means of a 2-hop cover index to process '//' on arbitrary graphs. However, the path queries of [26] do not allow filters, and moreover, more expensive queries are employed to search for ancestors and descendants (*i.e.,* 2-hop joins instead of single scans/index lookups used in our approach). To our knowledge, our update translation algorithm is among the first solutions for both (a) processing XPath queries with complex filters on DAGs stored in relations, and (b) incrementally maintaining indexing structures in response to updates.

8. Conclusions

We have proposed new techniques for updating XML views published from relational data. The novelty of our technique consists of (a) the ability to handle XML updates defined with *recursive* XPath queries over (possibly) *recursively defined* XML views; (b) the first method to rewrite XML updates into group updates on relational views that represent a DAG *compression* of an XML view, capturing XML view-update side effects; (c) a key-preservation condition on SPJ views that is less restrictive than constraints imposed by previous work but simplifies the analysis of relational view updates; and (d) efficient (heuristic) algorithms for handling *relational* SPJ view updates under key preservation, along with complexity results. Our results contribute to the study of view updates in *both* an XML and a relational setting. On the XML side, these yield an effective approach to dealing with XML view updates without relying on the limited view-update support of relational DBMSs. On the relational side, our complexity results and algorithms extend the line of research for processing relational view updates.

We plan to extend our techniques to handle more general XML updates such as those proposed in [22, 29]. We are also investigating the problem of finding minimal, side-effect-free relational updates in response to XML view updates.

9. References

[1] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.

[2] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, 2004.

[3] P. Buneman, S. Khanna, and W. Tan. On propagation of deletions and annotations through views. In *PODS*, 2002.

[4] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *VLDB*, 2000.

[5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, 2005.

[6] B. Choi. What are real DTDs like. In *WebDB*, 2002.

[7] E. Colen, H. Kaplan, and T. Milo. Labeling dynamic XML tree. In *PODS*, 2002.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. McGraw-Hill, 2001.

[9] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. In *PODS*, 1983.

[10] Y. Cui and J. Widom. Run-time translation of view tuple deletions using data lineage. *Technical Report, Standford University*, 2001.

[11] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3), 1982.

[12] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD*, 2001.

[13] full paper. Updating recursive XML views of relations. *http://homepages.inf.ed.ac.uk/wenfei/papers/viewfull.pdf.*

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

[15] G.F.Italiano. Finding paths and deleting edges in directed acyclic graphs. *Inf. Process. Lett.*, 28, 1988.

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

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

[18] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *ACM Symposium on Theory of Computing*, 1999.

[19] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, 2003.

[20] E. Koutsoupias and C. H. Papadimitriou. On the greedy algorithm for satisfiability. *Inf. Process. Lett.*, 43(1), 1992.

[21] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, 2003.

[22] A. Laux and L. Martin. XUpdate - XML Update Language, 2000. *http://www.xmldb.org /xupdate/xupdate-wd.html.*

[23] J. Lechtenborger and G. Vossen. On the computation of relational view complements. *TODS*, 28(2):175–208, 2003.

[24] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1), 1996.

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

[26] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *ICDE*, 2005.

[27] B. Selman and H. Kautz. Walksat home page, 2004. *http://www.cs.washington.edu/homes/kautz/walksat/.*

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

[29] G. Sur, J. Hammer, and J. Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[30] L. Wang, M. Mulchandani, and E. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *Xsym*, 2003.

[31] L. Wang, E. A. Rundensteiner, and M. Mani. Ufilter: A lightweight xml view update checker. In *ICDE*, 2006.

[32] L. Wang, E. A. Rundensteiner, and M. Mani. Updating XML views published over relational databases: Towards the existence of a correct update mapping. *DKE*, to appear.

Appendix

DTD validation

Given XML updates Δ_X, we first perform static optimization by validating the predefined DTD D with respect to Δ_X, and reject the updates if $\Delta_X(T)$ does not conform to D as required by the schema-directed definition of σ.

The validation is conducted at the schema level by leveraging the DTD normalization given in Section 2, as follows. Let Δ_X be defined in terms of an XPath query p. We first "evaluate" p on the DTD D to find the types of the elements reached via p. We then check whether the insertion or deletion of subtrees of these elements (types) violates their productions in the DTD D. Note that an insertion (resp. deletion) of a B child under an A element does not violate D only if the production of A is of the form $A \to B^*$. Thus updates of other forms can be immediately rejected. This can be checked in $O(|p|\ |D|^2)$ time, where $|p|$ and $|D|$ are the sizes of the XPath query p and the DTD D respectively. We omit the details of the validation algorithm due to lack of space. Compared to previous work on incremental DTD validation (*e.g.,* [**?**]) our algorithm is capable of handling XML updates defined in terms of XPath expressions rather than a single subtree insertion (or deletion) defined in terms of an absolute node-id path.

Maintenance of Auxiliary Structures

We give the incremental maintenance algorithm in response to XML view insertion.

Insertion. Algorithm $\Delta_{(\mathsf{M,L})}\mathsf{insert}$ is shown in Fig. 11. Given $\Delta_X = \mathsf{insert}\ (A, t)\ \mathsf{into}\ p$, it finds the Δ_M over M to maintain the reachability information, and moreover, updates the topological order L in response to the insertion of $st(A, t)$.

It is simple to compute Δ_M, which consists of two parts: (a) the reachability matrix for the newly inserted DAG $\mathrm{ST}(A, t)$ is computed by invoking Algorithm Reach (line 3); (b) for each $a \in \mathsf{anc}(r[\![p]\!])$ (ancestors of nodes in $r[\![p]\!]$) and each $d \in \mathrm{ST}(A, t)$, we add (a, d) to Δ_M (lines 4-5).

Maintaining L is a bit cumbersome. As will be shown, M is useful in maintaining L. Before considering to insert a DAG ($st(A, t)$), we first consider how to maintain L when one edge is inserted. For an edge insertion (u, v), if v is already in front of u in L, L remains valid without any change; otherwise, special care is needed to update node positions in L. We illustrate this by an example. Consider part of L: $\langle \ldots, d_u, u, a_{u_1}, a_1, d_{v_1}, a_{u_2}, v, \ldots \rangle$, where a_{u_1} and a_{u_2} are ancestors of u, d_{v_1} is a descendant of v, d_u is a descendant of u, and a_1 is neither an ancestor of u nor a descendant of v. After (u, v) is inserted, we can obtain a correct topological order by moving v and its descendants (d_{v_1}) between u and v such that they precede u. This yields $\langle \ldots, d_u, d_{v_1}, v, u, a_{u_1}, a_1, a_{u_2}, \ldots \rangle$. Note that d_{v_1} must be neither an ancestor of u (otherwise there is a cycle) nor an ancestor of a_1. To formalize this, we denote the nodes between u and v in L as $L[u : v]$. Given an edge insertion (u, v), the correct topological order can be obtained by moving nodes in $L[u : v] \cap \mathsf{desc}(v)$ to be *immediately* in front of u in L. The procedure of changing L to reflect the insertion (u, v) is denoted as $\mathsf{swap}(L, u, v)$, where u precedes v in L before the move.

We next explain the algorithm for updating L when inserting $\mathrm{ST}(A, t)$ (lines 6-14). Let L_A be the topological order for $\mathrm{ST}(A, t)$ (line 2) and N_C be the set of common nodes in L and L_A. The basic idea of the algorithm is to make the relative orders of nodes in N_C consistent in lists L and L_A before we merge L and L_A to obtain the updated L. To do this, we compute the topological oders L_{N_C} for nodes in N_C by considering the edges that connect nodes of N_C in either T or $\mathrm{ST}(A, t)$ (line 7), and then align L and L_A with L_{N_C} to make their positions consistent with L_{N_C} (lines 8-11). One subtlety is worth mentioning: when performing the alignment we follow the order of L_{N_C} from the right to the left. This processing order ensures that the position of aligned nodes will not be changed by subsequent alignment. To be specific, the aligned nodes are not descendants of nodes to be aligned and thus will not be moved any more when $\mathsf{swap}(L, u, v)$ is called in subsequent alignment (they are not descendants of v). Furthermore, if the root of $\mathrm{ST}(A, t)$ is already in T, we may need to change the order of L in response to the inserted edge (u, r_A), where $u \in r[\![p]\!](u \notin L_A)$ (lines 12-13). After we obtain two consistent lists L and L_A, we can merge L_A into L to generate the updated L (line 14). This can be done by regarding the nodes in N_C as "pivots" and inserting the new nodes (i.e. $L_A \setminus N_C$) into L before their respective "pivots".

Input: an insertion of the form Δ_X = $\mathsf{insert}\ (A, t)\ \mathsf{into}\ p$ over T, the rel. view V, reachability matrix M and topological order L.
Output: insertions Δ_M over M, and updated list L.

1. compute N_A and r_A, as lines 2-4 in Algorithm $\mathsf{Xinsert}$;
2. L_A := the topological order of nodes in $\mathrm{ST}(A, t)$;
3. Δ_M := reachability matrix for $\mathrm{ST}(A, t)$; /*using Algorithm Reach*/
4. **for each** $a \in \mathsf{anc}(r[\![p]\!])$ and **each** $d \in N_A$ /* computing Δ_M */
5. $\quad\Delta_M := \Delta_M \cup \{\ \mathsf{insert}\ (a, d)\ \mathsf{into}\ M\}$;
6. N_C := the set of common nodes in lists L and L_A; /*update L*/
7. L_{N_C} := the topological order of nodes in N_C;
8. **for** $(k = |L_{N_C}|; k > 1; k--)$ /*align L_A and L with L_{N_C} */
9. $\quad u := L_{N_C}[k]$; $\quad v := L_{N_C}[k-1]$;
10. $\quad$**if** $ord_{L_A}(u) < ord_{L_A}(v)$ **then** $\mathsf{swap}(L_A, u, v)$;
11. $\quad$**if** $ord_L(u) < ord_L(v)$ **then** $\mathsf{swap}(L, u, v)$;
12. **if** $r_A \in L$ **then for each** u in $r[\![p]\!]$
13. $\quad$**if** $ord_L(u) < ord_L(r_A)$ **then** $\mathsf{swap}(L, u, r_A)$;
14. L := merge L_A into L;
15. **return** (Δ_M, L);

Figure 11: Maintenance algorithm $\Delta_{(\mathsf{M,L})}\mathsf{insert}$ for insertions

Complexity. The worst-case time complexity of Algorithm $\Delta_{(\mathsf{M,L})}\mathsf{insert}$ is $O(|E_A| + |E_{N_C}| + (|N_C| + |r[\![p]\!]|)\ n + |N_A||E_A| + |N_A|\ n)$, where (a) $|N_A|$ is the number of distinct nodes, and $|E_A|$ is the number of edges in the inserted subtree $\mathrm{ST}(A, t)$, (b) $|N_C|$ is the number of common nodes in L and L_A, $|E_{N_C}|$ is the number of those edges that connect nodes of N_C in either T or $\mathrm{ST}(A, t)$, and (c) n is the number of distinct nodes in T. In practice $|N_C| < |N_A| < |E_A| \ll n \ll |V|$. The first and second factors are the cost of computing L_A and L_{N_C}, respectively, and the third factor is the cost of maintaining L, where $\mathsf{swap}()$ is called at most $2|N_C| + |r[\![p]\!]|$ times and each takes at most $O(n)$ time. Note that $\mathsf{swap}(L, u, v)$ is in $O(|L[u : v]|)$ time, which is usually much smaller than n. The fourth factor is the cost of computing the reachability matrix for $\mathrm{ST}(A, t)$, while the last factor is the cost of maintaining the reachability between nodes in $\mathrm{ST}(A, t)$ and the nodes in T.

Observe the following. (a) The analysis given above is the worst-case complexity. While it seems no better than the complexity of re-computing M and L from scratch, in practice the updated XML view $\Delta_X(T)$ typically differs slightly from the old view T, and $|r[\![p]\!]|$ and $|\mathsf{anc}(r[\![p]\!])|$ are often far smaller than n. (b) L_A and L_R are typically much smaller than L; this makes the fourth factor of the complexity of $\Delta_{(\mathsf{M,L})}\mathsf{insert}$ and the complexity of $\Delta_{(\mathsf{M,L})}\mathsf{delete}$ much smaller than $n\ |V|$ in practice. (c) As mentioned earlier, the computation of Δ_M and updating of L is in fact conducted in the background.

Proof of Theorem 5.2

A NP algorithm for checking CQ view updatability works as follows: it first guesses a group insertion Δ_R and then checks whether

$\mathcal{V}(\Delta_R(I)) = \Delta_V(V)$, which can be done in PTIME (data complexity).

We next show the problem is NP-hard, by reduction from the non-tautology problem. Consider an instance of the problem: $\phi = C_1 \vee \ldots \vee C_n$, where all the variables in ϕ are $x_1, \ldots, x_k$, C_j is of the form $l_{j_1} \wedge l_{j_2} \wedge l_{j_3}$, 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 such that ϕ is false, i.e., ϕ is not valid. This problem is known to be NP-complete.

Given ϕ, we define a relational database I, a single CQ view $\mathcal{V}$ under key preservation, and a single view insert Δ_V on $V = \mathcal{V}(I)$, such that ϕ is not valid iff there exists Δ_R and $\mathcal{V}(\Delta_R(I)) = \Delta_V(V)$.

Relational database I**.** The database consists of three base relations, R, R_ϕ and R_E, defined as follows.

- $R(A, B)$, where A is the key of the relation and B is a boolean. Intuitively, A is to hold a number in $[1, k]$ encoding a variable, and B is a truth value (T or F). That is, $R(A, B)$ is a truth assignment for ϕ. Initially $R(A, B)$ consists of a single special tuple $(0, T)$.
- $R_\phi(j, j_1, X_1, j_2, X_2, j_3, X_3)$, where j is the key of the relation. Initially, for each $C_j = l_{j_1} \wedge l_{j_2} \wedge l_{j_3}$, there is a tuple $(j, l_{j_1}, X_1, l_{j_2}, X_2, l_{j_3}, X_3)$ in R_ϕ such that l_{j_i} is s if $l_{j_i} = x_s$ or $l_{j_i} = \bar{x_s}$, X_i is T if $l_{j_i} = x_s$, and X_i is F if $l_{j_i} = \bar{x_s}$. Intuitively, each of these tuples in R_ϕ codes a clause in ϕ. A special tuple $(0, 0, T, 0, T, 0, T)$ is also in R_ϕ.
- $R_E(e_1, e_2, \ldots, e_k)$, where $e_1, \ldots, e_k$ are the key. Intuitively e_i is to code i in $[1, k]$. Initially, R_E consists of a single special tuple $(0, \ldots, 0)$.

View. We define a single view $\mathcal{V} = V_1 \times V_2$ in terms of conjunctive queries and under key-preservation as follows:

- $V_1 = \pi_{j, j_1, j_2, j_3} \sigma_C (R_1 \times R_2 \times R_3 \times R_\phi)$, where R_1, R_2, R_3 are renaming of R, and C is a boolean condition $c_1 \wedge c_2 \wedge c_3$, in which c_i is $R_i(A) = R_\phi(j_i) \wedge R_i(B) = R_\phi(X_i)$ $(i = 1, 2, 3)$. Intuitively, C holds if and only if one of the C_j 's is true.
- $V_2 = \pi_{e_1, e_2, \ldots, e_k} \sigma_D (R_E \times R_1 \times R_2 \times \ldots \times R_k)$, where $R_1, R_2, \ldots, R_k$ are renamings of R, and D is a boolean condition $\bigwedge_{i=1}^{k} R_i(A) = R_E(e_i)$.

Initially $V = \mathcal{V}(I)$ has a single tuple $(0, \ldots, 0)$ (k+4 0's).

View insert. We define Δ_V to insert a single tuple $(0, 0, 0, 0, 1, \ldots, k)$ into V.

We next verify that Δ_V is side-effect free iff ϕ is not a tautology. Indeed, if ϕ is not a tautology, then there is a truth assignment μ such that ϕ is false, and thus C_j is false w.r.t. μ. We define Δ_R based on μ as follows: insert tuples to $R(A, B)$ such that (i, T) is inserted into $R(A, B)$ iff $\mu(x_i) = T$, and (i, F) is inserted into $R(A, B)$ iff $\mu(x_i) = F$; furthermore, insert $(1, \ldots, k)$ into R_E. Then obviously Δ_V is side-effect free. Conversely, suppose that there is Δ_V that is side-effect free. Then $(1, \ldots, k)$ needs to be inserted into R_E, and a unique tuple of the form (i, X) needs to be inserted into the base relation R for each $i \in [1, k]$ due to the key constraint on R, such that Δ_V is indeed an update on the view V. Here X is either T or F, and thus after the insertion of Δ_V, $R(A, B)$ contains a valid truth assignment for ϕ. Since Δ_V is side-effect free, V_1 will remain $(0, 0, 0, 0)$ after Δ_V is performed. That is, C_j remains false. Thus ϕ is not a tautology. □

Proof of Theorem 5.3

We show the problem is NP-hard by reduction from the minimal set cover problem. An instance of the minimal 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 minimal.

Given S and C, we define an instance of the minimal view deletion problem. Let $S = \{x_i \mid i \in [1, n]\}$. We construct $|C|$ many base tables, n CQ views and a group view deletion, as follows.

1. For each $S_j \in C$, we define a base relation R_j consisting of a single column.

   Let I_j, the instance of R_j, be $\{j\}$, and let the database instance I be the collection of all I_j's defined above.
2. For each x_i, let T_i be the collection of all the subsets in C that contain x_i. Enumerate the elements of T_i as $(S_{i^1}, \ldots, S_{i^{n_i}})$. Define $V_i = R_{i^1} \times \ldots \times R_{i^{n_i}}$. Note that $V_i(I) = (i^1, \ldots, i^{n_i})$. Let $\mathcal{V}$ be the collection of V_i's for $i \in [1, n]$.

   Obviously, the views defined as above are key-preserving.
3. The group deletion Δ_V is to remove all tuples from all the views.

   Note that the tuple is removed from V_i without side effect if and only if the tuple from any R_{ij} is removed.

The minimum view deletion problem is to find a smallest set of the base relations $R_1, \ldots, R_{|C|}$ from which tuples are removed, while ensuring that the view tuples from V_i for $i \in [1, n]$ are deleted without side effect.

We next verify that the construction above is indeed a reduction from the minimum set cover problem. First suppose that C' is a minimal cover of S. We define Δ_R such that it consists of deletion of tuple from each base relation in $\{R_j \mid S_j \in C'\}$. Clearly, $\mathcal{V}(\Delta_R(I)) = \Delta_V(\mathcal{V}(I)) = \emptyset$ since C' is a cover of S. Furthermore, Δ_R is minimal since C' is minimal. Conversely, suppose that Δ_R is a solution to the minimal view deletion problem. Then let C' be the subset of C such that an element S_j of C is in C' if and only if Δ_R involves deletion of the tuple from the corresponding relation R_j. To see that C' is a cover of S, note that $\mathcal{V}(\Delta_R(I)) = \Delta_V(\mathcal{V}(I)) = \emptyset$, and thus for each $i \in [1, n]$, some set R_{ij} is in C'. Moreover, C' is minimal since Δ_R is minimal. □

Proof of Theorem 5.4

We verify that if Δ_V is not rejected during the coding of an instance Q, Δ_V and I of the CQ view insertion problem, then there exists a truth assignment μ that satisfies ϕ_Q if and only if there exists Δ_R such that $\Delta_V(Q(I)) = Q(\Delta_R(I))$.

Assume that there exists a truth assignment μ that satisfies ϕ_Q. Then we define Δ_R as follows. For each X_j and each tuple template t in X_j, we assign a value to each variable z in t based on μ. If z is bounded in ϕ_Q by $(z = c)$ for some constant c and $(z = c) \leftrightarrow x$, then we let $z = c$ if $\mu(x)$ is true; after this process if z is not assigned any value, z must be a free variable that ranges over an infinite domain τ_i and thus we can always pick a value c' for z without violating ϕ. Indeed, our coding distinguishes (bounded) variables with a finite domain from those (free) variables with an infinite domain, and encodes possible value selections of those variables having a finite domain in terms of additional clauses; the coding ensures that the value of z can be picked without causing side effects. For each relation I_i, let Δ_R^i consist of all these instantiated tuple templates from all X_j's that are a renaming of R_i. Let Δ_R be the collection of Δ_R^i's for $i \in k$. Then $\Delta_V(Q(I)) = Q(\Delta_R(I))$. Indeed, these newly inserted tuples do not produce view tuples that have a key of R_i that is not already in Δ_V, since otherwise this had been caught in the coding process and Δ_V would have been rejected. Furthermore, these newly insertions do not yield tuples that are not in Δ_V but share keys of Δ_v, as ensured by the coding ϕ_Q. Finally, all the tuples in Δ_V are coded in ϕ_Q and are guaranteed to be produced by $\Delta_R(I)$. Thus Δ_R carries out the desired view insertions without side effects.

Conversely, assume that there exists a group update Δ_R to I such that $\Delta_V(Q(I)) = Q(\Delta_R(I))$. Then by reversing the derivation of Δ_R given above we can define a truth assignment μ to propositional variables in ϕ_Q; indeed, we let $\mu(x)$ be true iff $(z = c)$ and $(z = c) \leftrightarrow x$ are in ϕ_Q, if z has the value c in Δ_R. It is easy to verify that μ satisfies the formula ϕ_Q. □

Evaluation of query on database with variables

Given the original database I_i ($i = 1, ..., n$), the set of relational tuples to be inserted X_i ($i = 1, ..., n$) and the conjunctive query Q=$\pi_P(\sigma_C(T_1, ..., T_n))$, where C is a conjunction of equalities and P is a set of projected attributes, the problem is how to evaluate query Q on database I_i incremented by X_i that contains variables to capture whether insertions X_i will yield side effects. The challenge here is that the selection conditions of Q cannot be evaluated on tuples with variables and thus SQL queries cannot work directly on tuples enriched with variables.

Before analyzing how side-effects are generated and discussing how to evaluate Q to capture side-effects, we will do some preprocessing in order to (1) guarantee that Δ_R can be generated from the conjunctive query (view) on $I_i \cup X_i$ for any instantiation of the variables in X_i; and (2) reduce the number of variables. The preprocessing consists of several steps: (1) If there is a selection condition such that $z_{ik} = z_{jl}$, $z_{ik} \in \vec{z_i}, z_{jl} \in \vec{z_j}$, we use one variable to rename z_{ik} and z_{jl}; (2) If a variable is not involved in selection conditions, it can be filled with a dummy value because the instantiation of the variable is not relevant to side-effects; and (3)If there already exits a base tuple r' sharing key with r in X_i, we fill the missing values in r according to r'.

We observe that there are only two types of side-effects.

1. A view tuple is a side effect if it contains at least one key from $I_i \setminus X_i$ and at least one key from $X_j \setminus I_j$.
2. A view tuple is a side effect if it is generated from X_i ($i = 1, ..., n$), but is not a tuple in $\Delta_R \cup Q(I_1 \cap X_1, ..., I_n \cap X_n)$

The above two kinds of side effects cover all possible side effects raised by the insertion of Δ_R while other possibility, such as $Q(I_1, ..., I_n)$, will not generate any side effect tuples. For convenience of presentation, we divide $I_i \cup X_i$ into three non-overlapping subsets for each $i \in [1, n]$:

- $U_i = X_i \setminus I_i, i \in [1, n]$
- $A_i = I_i \setminus X_i, i \in [1, n]$
- $B_i = X_i \cap I_i, i \in [1, n]$

To capture the first kind of side effect, for all possibilities of $T_1, ..., T_n$, where $T_i \in \{U_i, A_i, B_i\}$, such that there exist an $i, j \in [1, n]$, $T_i = U_i$ and $T_j = A_i$, we rewrite Q to accommodate the variables in U_i and thus to capture side effects. More specifically, we rewrite the selection conditions and projected attributes. We illustrate the rewriting using an example: given $Q := \pi_P(\sigma_C(R_1, R_2, R_3))$ and one combination (U_1, U_2, A_3), to capture the side effects from the combination we rewrite the Q into Q'=$\pi_{P_1}(\sigma_{C_2}(U_1, U_2, A_3))$. The selection conditions C in Q are discomposed into C_1 and C_2, where C_1 only contains equality conditions involving variables (must be in U_1 and U_2 in this example) while C_2 contains the other selection conditions. P_1 contains only the attributes contained in C_2. Observe that (1) the selection conditions in C_2 that do not contain variable can be imposed on Q', and (2) the projection on P_1 ensures that any two of generated side effect tuples produce different encoding. The second kind of side effect is captured similarly.

Input: relations $I_1, ..., I_n$, view V, a group insertion Δ_R,
the view definition $\pi_P(\sigma_C(R_1 \times ... \times R_n))$,
Output: side-effect encode or reject (exception)

1. Compute X_i from Δ_R w.r.t R_i, for $i \in [1, n]$;
2. Preprocess X_i;
3. $\Theta := \emptyset$ /* SAT instance */
4. $U_i := X_i \setminus I_i, i \in [1, n]$
5. $A_i := I_i \setminus X_i, i \in [1, n]$
6. $B_i := X_i \cap I_i, i \in [1, n]$

/* detect the first type of side-effect */
7. **for each** combination of $T_1, ...T_n$, s.t. $\exists i\ \exists j\ [T_i = U_i \wedge T_j = A_j]$,
   $\wedge\ \forall\ k\ [(k \neq i \wedge k \neq j) \rightarrow (T_k = U_k \vee T_k = R_k)]$
8. C_1 := selection conditions involving variables in T_i
9. $C_2 := C \setminus C_1$
10. P_1: = attributes involved in conditions in C_1
11. $\Delta V_1 := \pi_{P_1}(\sigma_{C_2}(T_1, ..., T_k))$
12. **for each** $t' \in \Delta V_1$
13. **if** t' does not contain variable **then** reject Δ_R **return**
14. **else** $\Theta := \Theta \wedge (\bigvee_{c_j \in C_1}((x_{k_j} \neq z_{k_j})))$
15. **endfor**
16. **endfor**

/* detect the second type of side-effect */
17. **for each** combination of $T_1, ...T_n$, s.t. $\exists i\ [T_i = U_i]$
    $\wedge\ \forall\ k\ [(k \neq i) \rightarrow (T_k = X_k)]$
18. C_1 := selection conditions involving variables in T_i
19. $C_2 := C \setminus C_1$
20. $\Delta V_2 := \sigma_{C_2}(T_1, ..., T_k)$
21. **for each** $t' \in \Delta V_2 \wedge t' \notin U$
22. **if** t' does not contain variable **then** reject Δ_R **return**
23. **else** $\Theta := \Theta \wedge (\bigvee_{c_j \in C_1}(x_{k_j} \neq z_{k_j}))$
24. **endfor**
25. **endfor**
26. return Θ

Figure 12: The `insert` algorithm

The algorithm is given in Fig. 12. Its input consists of (1) a set of base relations $\{I_1, ...I_n\}$, (2) a view V defined in terms of conjunctive query $V = \pi_P(\sigma_C(R_1 \times ... \times R_n))$, and (3) a group insertion $\Delta_R = \{t_1, ..., t_k\}$ against V. The first kind of side-effect is encoded in lines 7-16. If a returned tuple does not contain any variable, it is a side-effect tuple(line 13); If it contains some variables, we need to instantiate the variables such that the selection conditions in C_1 are not satisfied in order to avoid side-effect. More specifically, for each return tuple t_k containing variable, we construct for each condition c_j in C_1 one inequality $x_{k_j} \neq z_{k_j}$, where x_{k_j} is a variable and z_{k_j} can be either a constant or a variable. Side-effect tuple t_k can be avoided only if at least one of the above inequalities holds. Similarly we encode the second kind of side-effect (lines 17-25).