Updating Recursive XML Views of Relations

Citation for published version:

Digital Object Identifier (DOI):
10.1109/ICDE.2007.367922

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Abstract

This paper investigates the view update problem for XML views published from relational data. We consider (possibly) recursively defined XML views, 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 intriguing. Furthermore, many issues are still open for relational view updates, and need to be explored. In response to these, we revise the update semantics to accommodate XML side effects based on the semantics of XML views, and present efficient algorithms to translate XML updates to relational view updates. Moreover, we propose a mild condition on XML views, and show that under this condition the analysis of deletions on relational views becomes PTIME while the insertion analysis is NP-complete. Finally, we present an experimental study to verify the effectiveness of our techniques.

1 Introduction

Views provide an abstraction of the data stored in a database and are commonly used in practice. Commercial DBMSs have identified the need for materializing and/or providing ways of updating them, and propagating the updates to the underlying data [13, 20, 23]. Indeed, the study of relational views and their update mechanisms has received considerable attention (see, e.g., [10, 14, 18]). Recently, a number of systems have been developed to publish relational data to XML [1, 4, 11, 13, 20, 23]; the published data is effectively XML views of the relational data. Thus, the problem of transparently updating the XML views needs to be revisited. Given an XML view of a relational database, we want to propagate updates of the XML view to the original relational tables, without compromising the integrity of neither the XML nor the relational data.

While several commercial systems [13, 20, 23] allow users to define XML views of relations, their support for XML view updates is either very restricted or not yet available. Previous work on XML view updates [2, 25, 26] has focused on translating XML view updates to relational view updates and delegating the problem to the relational DBMS; however, most commercial DBMSs only have limited view-update capability [13, 20, 23]. The state of the art in XML view updates research explicitly focuses on non-recursively defined XML views and XML updates defined without recursive XPath queries. These restrictions are unfortunate since the recent proposals on XML update languages (e.g., [24]) employ recursive XPath queries while DTDs (and thus XML view definitions) found in practice are often recursive [6]. Given these requirements, we consider more general XML views and updates: possibly recursive XML view definitions and XML updates specified in terms of XPath expressions with recursion and complex filters, as illustrated below.

Example 1.1: Consider a registrar database with the following schema (keys are 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. As depicted in Fig. 1 (the dotted lines will be explained shortly), an XML view T of the relational database is published for the CS department. The view is required to conform to the DTD below (the definition of elements whose type is PCDATA is omitted):

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

The view is defined recursively since the DTD is recursive (course is indirectly defined in terms of itself via prereq). Consider an XML update $\Delta_X$, which inserts the subtree for course CS240, as a prerequisite of all courses given by the recursive XPath query $course[cno=CS650]//course[cno=CS320]/prereq$. To propagate $\Delta_X$ means that we need to find an equivalent $\Delta_R$ over the relational database that inserts the same information in the underlying tables so that if the data is re-published in...
XML it leads to the same XML view as the one we have after applying \( \Delta_X \) on \( T \).

The analysis becomes complicated since there are three sub-problems that cannot be treated in isolation, namely: (i) how are the XML views efficiently materialized, (ii) what are the correct update semantics for XML views of relational data over the materialization primitives, and (iii) how are the new semantics implemented and the updates propagated to the materialized XML views and the relational database.

**Efficient materialization of XML views.** An XML document published from a relational database has high compression potential. In the document of Fig. 1 (Example 1.1), certain subtree instances can be shared; one can materialize 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 — e.g., the subtree for course CS320). The compressed view becomes a directed acyclic graph (DAG), which is often significantly (at times even exponentially) smaller than the original tree. Moreover, one may want to store the view (DAG) in relations itself. Further, the aim is to use recursive XPath expressions for denoting the parts of the document to be updated. Translation from (recursive) XPath queries over recursive XML views to SQL queries is hard [17]. To our knowledge, no efficient algorithm exists for evaluating XPath queries with complex filters on DAGs stored in relations. To this end, we present an efficient algorithm for evaluating XPath queries with complex filters on DAGs, based on a new and incrementally maintained indexing structure to handle recursion and a technique for handling filters.

**XML update semantics.** Update semantics should be revised given the XML view materialization primitives. In Example 1.1, we are to insert CS240 as a prereq of only those CS320 nodes below CS650; however, CS320 nodes also occur elsewhere. As the XML view is published from the same relational database, all courses have unique prereq hierarchies. An insertion on selected paths of the hierarchy will result in side effects that should be detected. The users should then be consulted and, if they insist on continuing, the insert semantics needs to be revised so that the insertion will be performed at every CS320 node. The details of side effects on deletions are even more subtle and call for a new semantics. In light of this we refine the update semantics for XML views of relations to accommodate XML side effects. In addition, we develop an algorithm to translate recursive updates on a possibly recursively defined XML view to updates on the relational representation of the XML view.

**Update propagation.** Since the XML view is materialized in relations there is substantial work to be carried out in the relational realm. To this end, we identify a key-preservation condition on SPJ views, which is less restrictive than the conditions imposed by previous work [10, 14]. We es-

![Figure 2. Overview of XML view updates](image-url)

establish complexity results for the updatability problem to extend the few existing ones [3, 8]. 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).

**Problem statement and proposed solution.** Given an XML view defined as a mapping \( \sigma : R \rightarrow D \) from relations of a schema \( R \) to XML documents (trees) of a DTD, a relational instance \( I \) of \( R \), the XML view \( T = \sigma(I) \), and updates \( \Delta_X \) on the XML view \( T \), we want to compute relational updates \( \Delta_R \) such that \( \Delta_X(T) = \sigma(\Delta_R(I)) \). That is, the relational updates \( \Delta_R \) when propagated to XML via the mapping \( \sigma \), yield the desired XML updates \( \Delta_X \) on the view \( T \). We propose a framework for processing XML view updates, as shown in Fig. 2. For each XML view definition \( \sigma : R \rightarrow D \), we maintain a relational database \( I \) of \( R \), and the relational views \( V \) that encode the DAG compression of \( T = \sigma(I) \). The users pose updates on \( T \) (Section 2). Given a single XML update \( \Delta_X \) on \( T \) as input, we generate a group update \( \Delta_R \) on \( I \) such that \( \Delta_X(T) = \sigma(\Delta_R(I)) \) if such \( \Delta_R \) exists; otherwise reject \( \Delta_X \) as early as possible. Specifically, the framework processes an XML update \( \Delta_X \) on \( T \) in three phases, namely, DTD validation (see [7]), translation from \( \Delta_X \) to \( \Delta_V \) (Section 3), and translation from \( \Delta_V \) to \( \Delta_R \) (Section 4). If our algorithm detects a side effect, we report it to the user. After the relational update \( \Delta_R \) is computed, we update the underlying database \( I \) using \( \Delta_R \), the relational views \( V \) using \( \Delta_V \), and finally, in the background, invoke our incremental algorithm to maintain our auxiliary structures. An experimental study is presented in Section 5, followed by related work in Section 6 and future work in Section 7. See the full version [7] for details.

## 2 View Updates Revisited in the XML Setting

We give a brief overview of publishing XML from relational data and present a way of efficiently materializing the XML view in relations. We then define the syntax and semantics of XML updates over this representation.

### 2.1 Schema-Directed XML View Definition

Our techniques are applicable to XML views published from relations via any system (e.g., Attribute Translation Grammars—ATG [1], SilkRoute, XPERANTO). We first briefly review ATG, a DTD-directed method for defining XML views; we then present a way of materializing the published XML view in a relational database.
DTDs. A DTD $D$ is a triplet $(E, P, r)$, where $E$ is a finite set of element types; $r \in E$ is called the root type; $P$ defines the element types: for each $A \in E$, there is a production $A \rightarrow a$, where $a$ is a regular expression of the form:

$$\alpha ::= \text{PCDATA} | \epsilon | B_1, \ldots, B_n | B_1 + \ldots + B_n | B^*$$

where $\epsilon$ is the empty word, $B$ is a type in $E$ (a child type of $A$), and `+`, `*`, and `|` denote disjunction, concatenation and the Kleene star, respectively. A DTD is recursive if a type is defined (directly or indirectly) in terms of itself.

**XML views.** A publishing system implements a mapping $\sigma : R \rightarrow D$ from instances of a relational schema $R$ to documents of the target DTD $D$. (a) For each element type $A$ of $D$, $\sigma$ 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 $\alpha$, $\sigma$ specifies a SPJ query, $Q(A,B)($\$A$), which extracts data from a relational database $I$, using $\$A$ as a constant; it generates the $B$ children of an $A$ element and their $\$B$ values. For example, for the production $\text{prereq} \rightarrow \text{course}$*, the SPJ query $Q_{\text{prereq}, \text{course}}($\text{prereq}$)$ can be specified as:

```sql
select distinct c.cno, c.title
from prereq p, course c
where p.cno = \$prereq and p.cno2 = c.cno
```

Intuitively, at a `$\text{prereq}$` node $v$ with `$\$\text{prereq}$` value $p$, the subtree of $v$ is constructed as follows: (1) $Q_{\text{prereq}, \text{course}}(p)$ is evaluated on the database $I$; (2) for each distinct tuple $c$ in the result of the query, a `course` child $v_c$ of $v$ is generated, which carries $c$ as the value of its semantic attribute `$\text{course}$`; and (3) $c$ is then used in a similar fashion to expand the subtree rooted at $v_c$. The entire XML view is generated top-down starting from the root $db$, and conforms to the DTD of Example 1.1 [see [1, 7] for details].

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

As noted in Section 1, a subtree $\text{ST}(A, \$A)$ 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 $\text{ST}(A, \$A)$ no matter how many times it occurs in the XML view. This leads to a DAG representation of XML view $\sigma(I)$. In Fig. 1, for example, $\text{course}_1$ and $\text{student}_2$ are shared subtrees (see dashed lines).

### 2.2 XML View Updates: Side Effects, Semantics

**Syntax.** We consider a class of XML updates [24] specified in terms of XPath: (a) insert $(A, t)$ into $p$, (b) delete $p$. Here, $A$ is an element type, and $t$ is an instantiation of the semantic attribute $\$A$ of $A$. Given the instantiation we can uniquely identify the root of a subtree of type $A$. We define $p$ as an XPath expression ($q$ in $p[q]$) is called a filter:

$$p ::= \epsilon | A | * | / | p/p | p[q],$$

$$q ::= p | p = 's' | label() = A | q \land q \lor q \land -q,$$

**Side effects.** On detecting side effects, users can choose either to abort the update, or to carry on under the semantics we provide in the sequence. Detection of side effects will be further elaborated in Section 3.2.

Recall that each subtree in the XML tree is uniquely identified by the value of the semantic attribute of its root. Moreover, under DAG compression, a single subtree may be shared among multiple parents. Therefore any changes to the subtree must be reflected to all instances of the subtree, irrespective of the XPath specified in the update operation. This forms the very basis for the appearance of side effects.

**Example 2.1:** In Example 1.1, a new subtree was to be inserted to change the prerequisite hierarchy of only those CS320 nodes below CS650. However, since there is a unique CS320 subtree, all changes to its prerequisite hierarchy must be reflected to all CS320 nodes, rather than only to those below CS650, leading to side effects.

Side effects are more subtle for deletions. Consider delete $\text{course}[\text{cno}=\text{CS650}]/\text{prereq}/\text{course}[\text{cno}=\text{CS320}]$ on the same XML tree, that aims to remove course CS320 from the prerequisites of course CS650. This cannot be simply performed by physically removing all CS320 nodes as in previous work on XML view updates [2, 25, 26]: CS320 is itself an independent CS course and may be a prerequisite of other courses. For a correct deletion we need to find, for the root of the subtree to be removed, all its parents such that they are reachable via the XPath of the delete statement, i.e., those $\text{prereq}$ nodes (prereq2) below CS650 nodes, and then remove CS320 from the children list of only those parent nodes. Note that CS320 is not removed from the children list of node db since it is not reachable via the XPath.

**The semantics of XML view updates.** It is obvious that a new semantics should be developed to cope with side effects. This semantics needs to respect the hierarchical nature of XML views. Note that this semantics is different than the semantics of updates on XML data [24]. Given an XML view $T$ with root $r$, an insert operation: (a) 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 \( ST(A,t) \) as the rightmost child of \( u \) as required by the semantics of XML views.

A deletion on XML views (a) computes \( r[p] \); (b) for each node \( v \in r[p] \), it removes the subtree \( ST(A,t) \) from the children list of the parent node \( u \) of \( v \) such that \( u \) is reachable via XPath \( p \), where \( A \) is the type of \( v \) and \( t \) is the value of \( $A \) at \( v \); and (c) for any node \( u' \) of the same type and semantic attribute value as the parent \( u \) of \( v \), it removes \( ST(A,t) \) from the children list of \( u' \).

Compared to previous work [2, 25, 26], we support XML view updates that (a) are defined with much richer XPath expressions with recursion and complex filters, (b) operate on (possibly) recursively defined XML views, and (c) possess a new semantics that captures side effects, if any, of XML view updates. We also provide techniques to detect whether there are side effects and, in those cases, allow the users to cancel the update; otherwise, the operation will carry on with the semantics described earlier.

### 2.3 Relational Coding of Recursive XML Views

To reduce the update problem to a strictly relational one, we employ relational views to represent the XML views defined by a mapping \( \sigma : R \to D \) from a relational schema \( R \) to a DTD \( D \). This is nontrivial: (a) \( \sigma \) 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 compressions of XML views, i.e., a DAG representation of \( \sigma(I) \) where \( I \) is an instance of \( R \) as opposed to trees assumed in previous work. To this end we define a relational representation \( V_\sigma \) for the mapping \( \sigma \) by means of the edge relations in \( \sigma(I) \) as follows.

(a) We assume a compact, unique value associated with the tuple value of semantic attribute \( $A \) in \( \sigma(I) \). We assume w.l.o.g. the existence of a Skolem function \( [1] \) \( gen_{\text{id}} \) that, given the tuple value of \( $A \), computes a unique \( id_{A} \). We use \( gen_{A} \) to denote the set of the identities of all \( $A \) tuples.

(b) We encode an XML view definition \( \sigma \) in terms of \( V_\sigma \) as a set of SPJ queries \( Q_{edge_{A,B}} \) materializing the edge relations of \( \sigma \). More specifically, for each production \( A \to P(A) \) in the DTD of \( \sigma \), 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 SPJ query in \( \sigma \). Then \( edge_{A,B} \) is the set of pairs \( (ia,ib) \) such that \( ia = gen_{\text{id}}(a) \), \( ib = gen_{\text{id}}(b) \), 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 for the example of Fig. 1 is \( Q_{edge_{\text{prereq\_course}}} \):

- \( \text{select } gen_{\text{id}}(gp), gen_{\text{id}}(c.no), c.title \)
- \( \text{from } gen_{\text{prereq}} gp, prereq p, course c \)
- \( \text{where } p.cno1 = gp.cno \) and \( p.cno2 = c.cno \)

Observe the following about \( V_\sigma \). (1) \( V_\sigma \) encodes the DAG compression of XML view \( \sigma(I) \). Indeed, for any subtree \( ST(A,$A) \) in \( \sigma(I) \), each edge \( (ia,ib) \) in \( ST(A,$A) \) is stored only once in a relation \( edge_{A,B} \) no matter how many times \( ST(A,$A) \) (and thus the edge) appears in \( \sigma(I) \).

(2) Each \( Q_{edge_{A,B}} \) in \( V_\sigma \) is defined by a SPJ query. Thus \( V_\sigma \) consists of only SPJ views. (3) \( V_\sigma \) consists of a bounded number of relational views even if \( \sigma \) is recursively defined.

### 3 Mapping XML View Updates to Relations

We present a technique for translating XML updates on an XML view to updates on relational views representing the DAG compression of the XML view. The technique consists of four parts: (a) indexing structures for checking ancestor-descendant relationships, (b) an efficient algorithm for evaluating XPath queries on DAGs and detecting side effects, (c) algorithms to translate updates on the XML view to updates on its relational representation, based on the indexing structures and the evaluation algorithm, and (d) incremental algorithms for maintaining the indexing structures.

#### 3.1 Auxiliary Structures

To efficiently process recursion (‘/’ ) and filters in a DAG, we introduce two auxiliary structures: a topological order and a reachability matrix.

**Topological order.** Recall from Section 2 the function \( gen_{\text{id}}() \), which generates a unique id for each node based on the value of its semantic attribute. Given a representation of a DAG \( V \), we create a list \( L \) consisting of all the distinct node ids 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 \( v \). As will be seen shortly, \( L \) is useful in evaluating XPath filters as well as in computing and
The relational view $V$ and topological order $L$.

Output: reachability matrix $M$.

1. $M := \emptyset$;
2. for ($k := |L|; k > 0; k--$) /*process $L$ from right to left*/
3. \[ d := L[k]; \]
4. $A_d := \{a_{2} | a_{2} \in \text{anc}(a_1), a_1 \in \text{parent}(d) \}$;
5. insert $(a, d)$ into $M$ for each $a \in A_d$;
6. return $M$

Figure 3. Algorithm

Reach

maintaining the reachability matrix. The list $L$ can be computed in $O(|V|^2 \log |V|)$ time (see, e.g., [9]), 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 efficiently evaluate ancestor-descendant relationship between pairs of nodes in a DAG, we use a conceptual reachability matrix encoded as a relation $M(\text{anc}, \text{desc})$, where $\text{anc}$ is an ancestor node, and $\text{desc}$ a descendant. We use $\text{desc}(a)$ (resp. $\text{anc}(a)$) to denote the descendants (resp. ancestors) of node $a$ retrieved from $M$.

Relation $M$ can be computed in $O(|V|^2 \log |V|)$ time from $V$ (see, e.g., [9]). Capitalizing on the topological order $L$ we give Algorithm Reach, shown in Fig. 3, that computes $M$ in $O(n |V|)$ time. It is based on dynamic programming: for a node $d$, the ancestors of the nodes in the set of parents of $d$, denoted by $\text{parent}(d)$, 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). Given the topological order guaranteed by $L$, this can be achieved by traversing $L$ backwards (line 2). Note that $\text{parent}(d)$ can be computed from the edge relations in $V$.

Algorithm Reach runs in $O(n |V|)$ time: (a) for each node in $L$ we visit its parents once and thus any node $v$ is visited as many times as its in-degree, i.e., the number of incoming edges to $v$ in the DAG; (b) the sum of incoming edges to all nodes $v$ is $|V| |V|$, (c) each visit takes at most $O(n)$ time. In practice, $|M| \ll n^2 \ll |V|^2$, where $V$ is even up to an exponential factor smaller than the XML tree $T$.

3.2 Evaluating XPath Queries on DAGs

To translate updates $\Delta_X$ on XML views to updates $\Delta_R$ on relational views and detect whether the update will yield side effects, we must evaluate the XPath expression used in $\Delta_X$. 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, 21], they cannot be applied in our setting because they (a) either do not deal with complex filters which, as will be seen shortly, require a separate pass of the input DAG, or (b) do not address maintenance of the indexing structures they 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 on an XML tree that is (a) compressed as a DAG, and (b) stored in edge relations $V$. The algorithm takes as input an XPath query $q$ over XML tree $T$, 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; (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 such that $p$ reaches $v$ through $u$, and $C$ is the type of $u$; the set $E_p(r)$ is needed for handling deletions; and (c) the set of nodes $S$ in $T$ which are affected by the update but are not reachable via $p$. If the set $S$ is not empty, the update will generate XML side effects.

For XML data stored as a tree $T$, [16] developed an algorithm that evaluates an XPath query $q$ in two passes of $T$. The basic idea of [16] 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 [16]. We next show that a comparable complexity can be achieved when evaluating XPath queries on a DAG.

Our algorithm uses the following variables: (a) A list $Q$ of filters including all the sub-expressions of filters in $p$, sorted such that for any $q_i, q_j$ in $Q$, $q_j$ precedes $q_i$ if $q_i$ is a sub-expression of $q_j$. (b) For each $q_i$ in $Q$ and each node $v$ in $L$, two Boolean variables $\text{val}(q_i, v)$ and $\text{desc}(q_i, v)$ to denote whether or not the filter $q_i$ holds at $v$ and at any descendant $u$ of $v$, respectively. The algorithm has two phases: a bottom-up phase that evaluates filters in $p$ and computes $\text{val}(q, v)$ and $\text{desc}(q, v)$ for each node $v \in L$, followed by a top-down phase that computes $r[p]$ and $E_p(r)$. Due to lack of space we only outline the algorithm below.

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 $\text{val}(q, v)$ and $\text{desc}(q, v)$. This can be done by structural induction on the form of $q$. For example, when $q$ is $\text{label}() = A$, $\text{val}(q, v)$ is true if and only if $v$ is in $\text{gen}_A$. When $q$ is $q_1 \lor q_2$, $\text{val}(q, v) := \text{val}(q_1, v) \lor \text{val}(q_2, v)$. When $q$ is a path expression $p$, $p$ can be rewritten into a “normal form” $\eta_1/\ldots/\eta_n$, where each $\eta_i$ is either (a) $\epsilon[\eta_i]$, (b) a label $A$, (c) wildcard ‘*’, or (d) ‘/’. The normal form can be obtained in $O(|p|)$ time. Then, if $q$ is rewritten as $//\eta_2/\ldots//\eta_n$ with $\eta_i = /$, $\text{val}(q, v)$ is true if either $\text{val}(\eta_2/\ldots//\eta_n, v)$ or $\text{desc}(\eta_2/\ldots//\eta_n, u)$ is true for some child $u$ of $v$; correspondingly, $\text{desc}(q, v)$ is true.
Input: an insertion of the form $\Delta X = \text{insert} (A, t)$ into $p$ over $T$, and the relational view $V$.

Output: a group insertion $\Delta V$ over $V$.

1. $\Delta V := \emptyset$;
2. $E_A := \{(B, \text{gen}_{\text{id}}(\$u)), (C, \text{gen}_{\text{id}}(\$v))\}$ if $(u, v)$ is an edge in $\text{ST}(A, t, u, v)$ with type $B, C$ resp.;
3. $r_A := \text{id}$ of $\text{ST}(A, t)$'s root as generated by $\text{gen}_{\text{id}}(t)$;
4. for each $((B, ui), (C, vi)) \in E_A$
5. $\Delta V := \Delta V \cup \{ \text{insert} (ui, vi) \}$ along edge $B, C$;
6. for each $(B, ui) \in r[p]$
7. $\Delta V := \Delta V \cup \{ \text{insert} (ui, r_A) \}$ along edge $B, A$;
8. return $\Delta V$;

Figure 4. Algorithm Xinsert

if either $\text{val}(q, v)$ or $\text{desc}(q, u)$ holds. The algorithm proceeds in the topological orders $L$. Thus the truth values of $\text{val}((\eta_1 / \ldots / \eta_n, v))$ and $\text{desc}(\eta_1 / \ldots / \eta_n, u)$ are already available before evaluating $\text{val}(q, v)$ and $\text{desc}(q, v)$.

Top-down. We compute $r[p]$, $E_p(r)$ and $S$ as follows.

As mentioned, $p$ can be rewritten as $\eta_1 / \ldots / \eta_n$, in which all the filters have already been evaluated to a truth value at each node. Starting from the root $r$, we find nodes $C_i$ reached after each step $\eta_i$ and maintain a set of nodes $S$ in $T$ that are not reachable via $p$ but will be affected by the update. When $\eta_i$ is $'l'$ (resp. $'ll'$), $S$ is extended with the parent (resp. ancestor) nodes of $C_i$ that are not reached via $p$. These nodes can be found by using indexes on the edge relations $V$ when $\eta_i$ is $A$ or $\ast$, and by means of the reachability matrix $M$ when $\eta_i$ is $'ll'$. The nodes reached by the last step $\eta_n$ are put in $r[p]$, along with their types. The parents through which they are reached via $p$ are put in $E_p(r)$ along with their types. There is a side effect iff $S$ is not empty. At that point, users may either abort the update, or continue using our update semantics.

Complexity. In the bottom-up phase, each node $v$ is visited at most as many times as its incoming edges. 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 as many times as its the fan-out. The complexity of the algorithm is therefore $O(|p| + |V|)$.

Observe the following: (a) When the DAG is a tree each node has one incoming edge and our algorithm visits each node at most twice, i.e., it has the same complexity as that of $[16]$. When dealing with DAGs that do not have a tree structure, it is necessary to visit all the edges in a DAG in the worst case and thus our algorithm is optimal. (b) In contrast to [16], 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 trees) while [16] cannot work on DAGs.

3.3 Translating Updates from XML to Relations

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

Insertion. Algorithm Xinsert is presented in Fig. 4. Given $\Delta X = \text{insert} (A, t)$ into $p$ on the XML view $T$, the algorithm returns the group insertions $\Delta V$ over $V$ (which will then be tested for acceptance). We first compute the set of edges in the newly inserted subtree $\text{ST}(A, t, ui)$ at $r_A$, according to the publishing mapping (lines 2-3), through function $\text{gen}_{\text{id}}()$. We then generate the relational view updates: for each edge $(ui, vi)$ in the newly inserted subtree, we add $(ui, vi)$ to $\Delta V$ (lines 4-5); moreover, for each $(B, ui) \in r[p]$, we add $(ui, r_A)$ as a new edge in $\Delta V$ (lines 6-7). The set $r[p]$ of pairs $(B, ui)$ of node identifiers along with their types reached by XPath $p$ from the root of $T$ (line 6) is computed using our XPath evaluation algorithm.

Deletion. Given $\Delta X = \text{delete}$, Algorithm Xdelete (not shown due to space constraints – see [7]) returns the group of deletions $\Delta V$ over the edge relations, which will be tested for acceptance. For each node $vi$ in $r[p]$ and each parent $ui$ of $vi$ in $E_p(r)$, Xdelete removes the edge $(ui, vi)$ from $V$ (lines 2-3). The parent-child relation is computed by using the set $E_p(r)$, whose computation is coupled with that of $r[p]$ (see Section 3.2).

Example 3.1: Consider the XML update $\Delta X_1 = \text{delete} //\text{course}[\text{cno=}CS320]//\text{student}[\text{sid=}S02]$ on the XML tree in Fig. 1, which is to delete student S02 from the CS320 subtree. Given this as input, Algorithm Xdelete yields $\Delta V_1 = \{(\text{takenBy}_{t_1}, \text{student}_{t_2})\}$. □

Complexity. Alg. Xinsert takes $O(|E_A| + |r[p]|)$ time at most ($|E_A|$ is the number of edges in $\text{ST}(A, t)$). Alg. Xdelete takes $O(|E_p(r)|)$ time. Added to $O(|p| + |V|)$ for evaluating $p$, this is the cost of generating $\Delta V$ from $\Delta X$.

3.4 Maintaining Auxiliary Structures

The maintenance of auxiliary structures $L$ and $M$ is performed in the background in parallel with the processing of relational updates. What we ideally would like is to incrementally update $M$. Existing incremental techniques [12, 15] for updating reachability information are not applicable since they rely on special auxiliary structures which are themselves expensive to construct and maintain (e.g., [12] requires the computation of a spanning tree, taking $O(n + |V|)$ time for each node insertion). Incremental algorithms of updating topologically ordered lists (e.g., [19]) take $O(|V|)$ time per edge insertion. We give a maintenance algorithm for $M$ with $O(n + |V|)$ complexity by using $L$, and for $L$ with $O(n)$ time for each edge insertion using $M$.

Deletion. Incremental maintenance in response to XML view deletions is given in Algorithm $\Delta_{\text{M, L, delete}}$ (Fig. 5). The algorithm efficiently produces the following by scanning the elements of an XML deletion $\Delta_X$: (a) deletions $\Delta M$ over $M$, (b) an updated $L$, and (c) the set of edges
Input: a deletion of the form $\Delta_X = \text{delete } p$ over $T$, the rel. view $V$, reachability matrix $M$ and topological order $L$.

Output: deletions $\Delta'_V$ over $V$, $\Delta_M$ over $M$, and updated list $L$.

1. $\Delta'_V := \emptyset$; $\Delta_M := \emptyset$; $L_R := \emptyset$;
2. $L_R :=$ the sorted list $\text{desc}(r[p])$ according to topological order $L$;
3. keep$(d) = \text{true}$ for each $d \in T$; *initialize state $\gamma*$
4. for each $d \in L_R$ traversed backwards
5. $P_h := \emptyset$;
6. for each $a \in \text{parent}(d)$
7. if $(a, c), (a, d) \in E_R(r)$ and keep$(a)$ is true
8. then $P_a := P_a \cup \{a\}$; $A_a := \{a_2 \mid a \in \text{ancestors}(a_2), a_1 \in P_a\}$;
9. for each $a \in \text{anc}(d) \setminus A_d$
10. $\Delta_M := \Delta_M \cup \{\text{delete } (a, d) \text{ from } M\}$;
11. if $P_a = \emptyset$; $\phi$; compute $\Delta'_V$ and update $L/\gamma$;
12. then keep$(d)$ := false;
13. delete $d$ from list $L$;
14. for any child $d'$ (of type $H$) of $d$ (of type $G$)
15. $\Delta'_V := \Delta'_V \cup \{\text{delete } (d, d') \text{ from } \text{edge} L, H\}$;
16. return $(\Delta'_V, \Delta_M, L)$

Figure 5. Maintenance algorithm $\Delta_{(M,L)}$ delete

$\Delta'_V$ 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. The set $\Delta'_V$ is a consequence of deletions $\Delta_M$ computed by $\Delta$-delete. The need arises when a node $d \in \Delta_V$ is to be completely removed.

The algorithm progresses by populating deletions $\Delta_M$ while, simultaneously, removing elements from $L$ and populating $\Delta'_V$. The first step is arranging all nodes in all deleted subtrees in a list $L_R$ (line 2): we compute $\text{desc}(r[p])$, 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 the XML tree $T$ we associate a state keep$(d)$, initialized to true, that keeps track of whether the node should be ultimately deleted or not (line 3). $L_R$ is then traversed backwards (line 4); this processing order 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 $\text{anc}(d)$ 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 keep$(d)$ to false and delete it from $L$ (lines 13-14). According to the semantics of $L$, an element removal does not affect the topological order. In addition, all outgoing edges from a deleted node $d$ are deleted from $V$ (lines 15-16); children $d'$ of $d$ can be readily identified from the edge relation determined by the types of $d$ and $d'$.

Example 3.2: Recall $\Delta_X$, of Example 3.1. Algorithm $\Delta_{(M,L)}$ delete returns (1) $\Delta'_V_1 := \emptyset$, (2) an unchanged $L$, and (3) $\Delta_M = \{(\text{prereq}_2, \text{student}_2), (\text{prereq}_2, \text{sid}_2), (\text{prereq}_2, \text{name}_2), \ldots\}$, i.e., the reachability information from nodes $\text{prereq}_2$, course1 and takenBy1 to nodes in the S02 subtree (student2, sid2 and name2). Note that the set of edges $\{(\text{takenBy}_2, \text{student}_2), (\text{takenBy}_2, \text{sid}_2), (\text{takenBy}_2, \text{name}_2), \ldots\}$, i.e., the edges between takenBy2 (and thus course2) and the S02 subtree are still valid and are therefore not included in $\Delta_M$.

Insertion. Algorithm $\Delta_{(M,L)}$ insert is shown in Fig. 6. Given $\Delta_X = \text{insert } (A, t)$ into $p$, it finds the $\Delta_M$ over $M$ to maintain the reachability information, and updates the topological order $L$ in response to the insertion of $st(A, t)$.

It is simple to compute $\Delta_M$, which consists of two parts: (a) the reachability matrix for the newly inserted DAG $\text{st}(A, t)$ is computed by invoking Algorithm Reach (line 3); (b) for each $a \in \text{anc}(r[p])$ (ancestors of nodes in $r[p]$) and each $d \in \text{st}(A, t)$, we add $(a, d)$ to $\Delta_M$ (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 $\text{st}(A, t)$, 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 $L$: $\{\ldots, d_u, u, a_{u1}, a_1, d_{v1}, a_{v1}, a_{v2}, v, \ldots\}$, where $a_{u1}$ and $a_{v1}$ are ancestors of $u$, $d_{v1}$ 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 ($d_{v1}$) between $u$ and $v$ such that they precede $v$. This yields $\{\ldots, d_u, d_{v1}, u, a_{u1}, a_1, d_{v1}, a_{v1}, a_{v2}, v, \ldots\}$. Note that $d_{v1}$ 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 $L[u : v] \cap \text{desc}(v)$ to be immediately in front of $u$ in $L$. The procedure of changing $L$ to reflect the insertion of $(u, v)$ is denoted as swap$(L, u, v)$, where $u$ precedes $v$ in $L$ before the move.

We next explain the algorithm for updating $L$ when inserting st$(A, t)$ (lines 6-14). Let $L_A$ be the topological order for $\text{st}(A, t)$ (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 order $L_{N_C}$ for nodes in $N_C$ by considering the edges that connect nodes of $N_C$ in either $T$ or $\text{st}(A, t)$ (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 swap$(L, u, v)$ is called in subsequent alignment (they are not descendants of $v$). Furthermore, if the root of $\text{st}(A, t)$ is already in $T$, we may need to change the order of $L$ in response to the inserted edge $(u, r_A)$, where
4 Updating Relational Views

We briefly outline the techniques for processing SPJ view updates under key preservation. Details can be found in [7].

**Key preservation.** Consider an 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 $R(\bar{a})$. We say that $Q$ is key preserving if for each $R_i$, the primary key of $R_i$ is included in $\bar{a}$ (with possible renaming).

Key preservation is far less restrictive than other conditions proposed in earlier work for handling relational view updates (e.g., [10, 14]). A mapping $\sigma : \mathcal{R} \rightarrow D$ from a relational schema to a DTD employs SPJ queries [1]. Every SPJ query can be made key-preserving by extending its projection-attribute list to include the primary keys.

**Analysis.** Given 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 $\Delta_V$, is there a group update $\Delta_R$ on the database $I$ such that $\Delta_V(\mathcal{V}(I)) = \mathcal{V}(\Delta_R(I))$? In this setting, $\Delta_V$ consists of either only tuple deletions or only tuple insertions, as produced by the translation algorithms of the last section. These deletions and insertions in $\Delta_V$ are translated to deletions and insertions in $\Delta_R$, respectively. We use $\bar{V}$ to denote the view $\mathcal{V}(I)$. We refer to this problem as the **view updatability problem**.

It is known [3] that without key preservation, the updatability problem is NP-hard for a single deletion and a single PJ view, i.e., when $\Delta_V$ consists of a single deletion and $\bar{V}$ is a view defined with projection and join operators only. We show that key preservation simplifies the updatability analysis for a collection of SPJ views and group deletions. More complexity results of view updates can be found in [8].

**Theorem 4.1:** For group view deletions $\Delta_V$, the SPJ view updatability problem is in PTIME.

The problem is intractable for insertions under key preservation; the lower bound is verified by reduction from the non-tautology problem, which is NP-complete.

**Theorem 4.2:** The SPJ view updatability problem is NP-complete even when $\Delta_V$ has a single insertion and $\bar{V}$ has a single view.

We give a PTIME algorithm for computing database tuple deletions $\Delta_R$ from a group of view deletions $\Delta_V$ in [7]. We also provide in [7] a heuristic algorithm for handling group view insertions 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 [22] to efficiently compute $\Delta_R$ if it exists.

5 Experimental Study

We conducted a preliminary experimental study of our proposed view update mechanism in order to verify its effectiveness.

All experiments were conducted on a dataset of four base relations: $C(c_1, \ldots, c_{16}), F(f_1, \ldots, f_{16}), H(h_1, h_2)$ and $C_U(c_1, \ldots, c_{16})$, where underlined attributes indicate keys. The domain of $f_1$ was equal to that of $c_1$ and $c_{17}$. The
remaining $C$ and $F$ attributes controlled how many joining
$C$ and $F$ tuples were filtered out. The domains of $h_1$
and $h_2$ were the same as that of $c_1$. In addition (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 $C_U$, consisting of
100M $C$-tuples, ensured that whenever $h_2$ joined with $c_1$
it always yielded a $C$-tuple. The sizes of $F$ and $H$ were
proportional to the size of $C$, used for reporting the size of
the database; specifically, we report $|C|$, which ranges from
1,000 to 1,000,000 tuples, while $|F| = |C|$ and $|H| \simeq 3|C|$. We
defined an XML view of $C$, $F$ and $H$; as indicated in
Fig. 7(a), the $C$ nodes in the view were recursively defined,
and a recursion of $C$ in the view can be understood as
$\pi_{c_1, f_1, h_1, h_2} (\sigma_{C_1=I_1 \land h_1=I_2 \land c_2=I_3 \land C_3=I_4} (C \times F \times H \times C_U))$. Here $C$ subtrees are shared, and subtree
sharing accounted for 31.4% of $C$ instances. Figure 7(b)
lists some statistics on the number of published $C$ subtrees
and their compressed DAGs, and the corresponding sizes of
the reachability matrix $M$ and topological order $L$.

Varying database size. We generated two random up-
date 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. Class $W_1$ used XPath queries using the descen-
dant axis and value filters; XPath queries in $W_2$ used
the child axis and value filters; finally, $W_3$ contained XPath
queries using the child axis and both structural and value
filters. The times we report include: (a) the time to evaluate
XPath queries; (b) the time to translate $\Delta_X$ to $\Delta_V$ (Algo-
rithms $Xinsert$ and $Xdelete$) and subsequently $\Delta_V$ to $\Delta_R$,
and the time to execute the update; and (c) the time to main-
tain the auxiliary structures in the background (Algorithms
$\Delta_{(M,L)}insert$ and $\Delta_{(M,L)}delete$).

Figures 8(a), 8(b) and 8(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 database sizes.

Note that both axes use a logarithmic scale. The algorithms
scale linearly with the size of the relational database. As
shown, deletion time is dominated by XPath evaluation. Al-
though the cost for auxiliary structure maintenance 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., a greater $|E_p(r)|$),
which are then examined by Algorithm delete.

Similar results are reported for insertions, as shown in
Figures 8(d), 8(e) and 8(f) for $W_1$, $W_2$ and $W_3$, respectively
(again, using logarithmic scales). The size of the inserted
tree was fixed. The SAT solver [22] 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 for deletions, our insertion algorithms
scale linearly with the size of the relational database.

Varying update size. For these experiments, we fixed $|C|
$ to 100K tuples. Figure 8(g) shows the performance of
each algorithm as we varied $|E_p(r)|$ (see Section 3.2) for
deletions and $|E_p(r)|$ for insertions, while keeping ST$(A,t)$
constant to a single $C$-subtree. The runtimes for Algo-
rithms $Xinsert$, $Xdelete$, delete and insert are shown on
the left $y$-axis and the runtimes for algorithms $\Delta_{(M,L)}insert$ and
$\Delta_{(M,L)}delete$ are shown on the right one. The translation
time from $\Delta_X$ to $\Delta_V$ for Algorithm $Xinsert$ (resp. Algo-
rithm $Xdelete$) increases slightly as $|E_p(r)|$ increases. The slope for Algorithm delete is large, as the in-
crease of $|E_p(r)|$ involves more queries to determine the
source tuples to be deleted. The performance of Algo-
rithm insert is dominated by the coding time. As $|C|$ is
far larger than $|ST(A,t)|$ and $|E_p(r)|$, and the number of database
queries required remains fixed, the coding time rem-
ains roughly constant though the size of the resulting cod-
ing increases; that only results in a non-observable increase
in the SAT solver’s runtime keeping the curve relatively flat.
The performance of Algorithm $\Delta_{(M,L)}insert$ (which can be
found in [7]) and Algorithm $\Delta_{(M,L)}delete$ is almost unaf-
fected by $|E_p(r)|$ since $|ST(A,t)|$ is fixed.

Similar results are shown in Fig. 8(h) where we var-
Effectiveness of incremental maintenance. The cost of incrementally maintaining the reachability matrix $M$ and the topological order $L$ is shown in Table 1. The first column is the size of the database. The total time needed for incrementally maintaining both auxiliary structures is given in the second column for Algorithm $\Delta_{(M,L)}^{\text{insert}}$ and in the third column for Algorithm $\Delta_{(M,L)}^{\text{delete}}$. The time for recomputing each structure is shown in the last two columns. The advantages of incremental maintenance become more prominent as the size of the data increases.

<table>
<thead>
<tr>
<th>Sizes</th>
<th>Incremental (Sec.)</th>
<th>Recomputation (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>14.0</td>
<td>14.0</td>
</tr>
<tr>
<td>10K</td>
<td>14.0</td>
<td>14.0</td>
</tr>
<tr>
<td>100K</td>
<td>14.0</td>
<td>14.0</td>
</tr>
<tr>
<td>1M</td>
<td>14.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>

Table 1. Incremental maintenance of $L$ and $M$

6 Related Work

Commercial database systems [13, 20, 23] provide support for defining XML views of relations and restricted view updates. IBM DB2 XML Extender [13] supports only operation of updates from relations to XML but not vice-versa. Oracle XML DB [20] does not directly allow updates on XML (XMLType) views. In SQL Server [23], users 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, 26]. In [2], XML views are defined as query trees and are mapped to relational views. XML view updates are propagated 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. An analysis on deciding whether or not an update on XML views is translatable to relational updates, along with detection algorithms, are provided in [26] and demonstrated in [25].

There has been a host of work ([10, 13, 14, 18, 20, 23]) on relational view updates. [10] provides algorithms for handling restricted view updates without side effects in the presence of functional dependencies. The algorithm in [14] studies updates (with side effects) on a restricted class of SPJ view: key-foreign key joins and join attributes must be preserved. Our key preservation condition is less restrictive than that of [10, 14]. Commercial DBMSs [13, 20, 23] allow updates on very restricted views.

7 Conclusions

We have proposed new techniques for updating XML views published from relational data. We plan to extend our techniques to handle more general XML updates in [24].

References

[23] SQL server. MSDN Library.