# Updating Views Over Recursive XML

Ming Jiang, Ling Wang, Murali Mani, and Elke Rundensteiner

Worcester Polytechnic Institute, 100 Institute Road, Worcester MA 01609, USA
{jiangm|lingw|mmani|rundenst}@cs.wpi.edu

**Abstract.** We study the problem of updating XML views defined over XML documents. A view update is performed by finding the *base updates* over the underlying data sources that achieve the desired view update. If such base updates do not exist, the view update is said to be *untranslatable* and rejected. In SQL, determining whether a view update is translatable is performed using *schema level analysis*, where the view definition and the base schema are used. XML schemas are more complex than SQL schemas, and can specify recursive types and cardinality constraints. In this paper, we propose a solution based on schema level analysis for determining whether an update over XML views is translatable and for finding the translation if one exists, while considering the features of XML schemas.

## 1 Introduction

In databases systems, a user sees a portion of the base data called a view. Therefore he/she may need to update base data through these views (view updates). Especially in shared databases, it is essential to provide the capacity to support view updates. In the relational scenario, there have been many studies on determining whether a view update is *translatable* [5]. A common semantics used for determining whether a view update is translatable is *side-effect free semantics*. In this semantics, a view update is said to be translatable if there exists base updates that achieve the desired view update without affecting any other portion of the view. Current relational/SQL systems use *schema level analysis* for determining whether a view update is translatable, where the view definition and the base schemas are used.

Nowadays, as XML is becoming the standard format for data exchange, database community is exploring its ability to store data. In fact, view updates become more common as many XML databases are available on the internet, and a large number of users have access to such databases. In this paper, we study how to perform XML view updates over XML data sources, using schema level analysis. This problem is much harder than for relational schemas because of the complex features in XML schema, such as recursive types and cardinality constraints.

Let us consider an example XML document with its schema as in Figure 1. Note the base schema element $course$ is recursive, as a course may have a child element $pre$, which stands for pre-requisite for this $course$, and $pre$ in turn can have $course$ elements as its children. Similarly, the base element $pre$ is also recursive. Now consider two queries over $D$, as shown in Figure 2 and Figure 3.

In Figure 2, (a) is the XQuery statement which defines the view. (b) is the view schema tree that corresponds to the XQuery. (c) is the view instance tree generated by the XQuery and XML document $D$. The same goes with Figure 3. [1]

[1] The subscripts *a*, *b*, *c* in Figure 1 and *1,2,3* in Figure 2(c) and Figure 3(c) are for illustration purpose only. They do not appear in the actual documents or views.

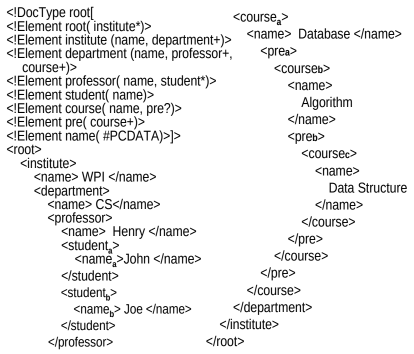


**Fig. 1.** XML document $D$ with Schema(D)

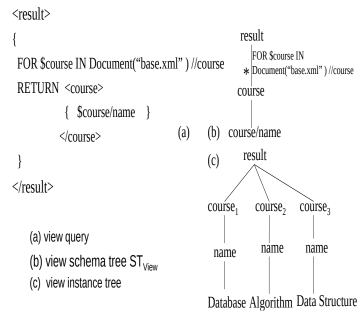


**Fig. 2.** Query $Q_1$ and corresponding view

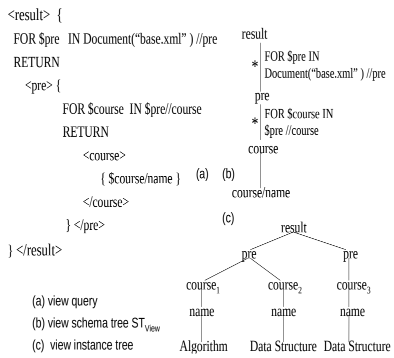


**Fig. 3.** Query $Q_2$ and corresponding view

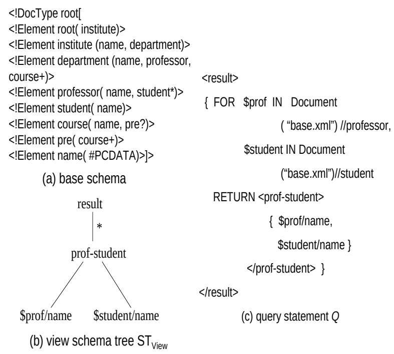


**Fig. 4.** Query $Q_3$

A user may want to delete $course_1$ in Figure 2(c). If we delete $course_a$ in $D$, this update would cause $course_2$, $course_3$ and their descendants to be removed in Figure 2(c). This is a side-effect and therefore it is not a correct translation. Now let us consider Figure 3(c) and try to delete $course_2$. We can achieve this by deleting the base element $course_c$ which has the $name$ child. However, doing so will also delete $course_3$ in the view and therefore it is also not a correct translation.

Intuitively, recursive base schemas and queries cause the above problems. However, are the above two scenarios the only cases that recursion may have side-effects? If not, how can we effectively check out all such side-effects? This problem has not been studied, to the best of our knowledge.

There are also other XML features that need to be considered for XML view update problems, such as cardinality constraints in the base schema. Will these features make the problem different from the relational scenario? Let us take a look at the query in Figure 4(c). It indicates that each $professor$ element in the base will join with every $student$ element. Therefore each $professor$ and $student$ may be used more than once and we cannot delete $prof$-$student$ view element. However, let us reconsider this query, given the base schema as shown in Figure 4(a). It indicates that there is only one $professor$ in the base. We now know that each $student$ will be used only once and we can delete a certain $prof$-$student$ by deleting the corresponding $student$ in the base XML document. From this example, we can observe that utilizing cardinality information provided in the base schema may give a better translation for the view update. How to fully handle cardinality is also discussed in this paper.

Our main technical contributions include: we study how features in XML schemas, such as recursive types and cardinality constraints, impact the XML view update problem. We propose an algorithm for determining whether a view update over XML data sources is translatable and for finding the translation if one exists, based on schema level analysis. Our algorithm is sound (a translation returned by our algorithm is guaranteed to not cause side-effects) and complete (a translation is guaranteed to be returned by our algorithm if there exists one). We believe these results go a long way towards understanding the XML view update problem and provide the capacity to efficiently update XML views.

**Outline.** The rest of the paper is organized as follows. Section 2 defines view update translatability and then defines the scope of the problem we consider. Section 3 introduces notations and background. Section 4 discusses how to handle unique features in XML schemas when solving the view update problem over XML data sources. Section 5 proposes our three-step checking algorithm and Section 6 gives a conclusion and discusses the future work.

## 2 Related Work

There are many studies on view updates in relational scenario, such as [6, 5, 9, 4]. [6] introduces the concept of a complementary view. The authors argue that when changing the data in the base corresponding to the updates on the view, the rest of the database that is not in the view should remain unchanged. This solution tends to be too strict, as many view updates are not translatable by this theory. In [5], the authors argue that we can perform a view update by deleting base tuples that contribute to the existence of this view element. Also such base tuples are required not to contribute to other view elements to avoid side-effects. Similarly, in [9], Keller proposes an algorithm to check whether 1-1 mapping exists between a set of base tuples and a set of view tuples. This mapping indicates that a certain view element can be deleted without side-effects.

While [6, 5, 9] study the view update problem on the schema level, there are other works such as [4] that study the problem on the instance level. Therefore in [4], more updates can be performed without side-effects. However, because of the large size of the database, such data-centric algorithms tend to be more time-consuming.

In order to utilize the maturity of relational database techniques and also adapt to the current required web applications, people tend to build XML views over relational databases, such as [12, 13]. There are some research that consider XML views as compositions of flat relational views, such as [7], for the purpose of querying relational databases. Some other work further study the updatability of XML views over relational databases. In [15], the authors discuss how to check side-effects for updating XML view elements over a relational database. In [3] the authors use the nested relational algebra as the formalism for an XML view of a relational database to study the problem of when such views are updatable. However, given an XML view over XML data, how to check the updatability of the view elements and further give the correct, efficient translation of this view update remains unsolved.

Language for updating XML documents is being studied by [1]. [2] discusses updates in XML scenarios. [14] presents some interesting problems in XML view updates. [10] considers virtual updatable views for a query language addressing native

XML databases, including information about intents of updates into view definitions. [11] studies type checking in XML view updates.

## 3 View Update Translatability and Problem Scope

### 3.1 View Update Translatability Definition

A view update operation *u* can be a delete, an insert or a replacement. The corresponding update on the XML base is said to be the translation of the view update.

**Definition 1.** *Let D be an XML document and V a view defined by $DEF^V$ over D. An XML document update sequence $U^R$ is a correct translation of a view update $u^V$ if $u^V(DEF^V(D))=DEF^V(U^R(D))$.*

This definition is depicted in Figure 5. The update is correct if the diagram in Figure 5 commutes.

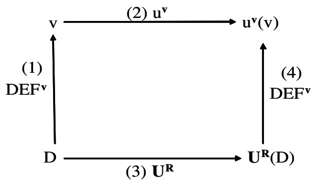


**Fig. 5.** Rectangle rule

### 3.2 Problem Scope

**Update Operations Considered** As we introduced above, a view update operation can be a delete, an insert or a replacement. Deletions are typically considered to be different from insertions. For instance, consider an SQL view defined as a join between $student$ table and $professor$ table, where a $student$ row joins with at most one $professor$ row. The SQL standard [8] supports deleting a row in this view by deleting a corresponding $student$ row, whereas inserts are rejected as they might need to insert into $student$ table, or $professor$ table or even both, which is more complex and hard to decide. As the first work considering view updates over XML data sources, we consider only deletions and inserts are out of our scope. Further, we study single view element deletion, as opposed to deleting a set of view elements. In addition, we do not use a view update language. As we focus on updates of a single view element, how the view element is specified (by the view update language) is not significant.

**Base Schema Language** We use DTD (Document Type Definition) as schema language to describe the underlying databases. DTD is a very expressive and complex language. The two most significant features in DTD that we consider are recursion and cardinality. The cardinality information is obtained from the content model in DTD, which uses "*", "+", "?", "," or "|". We will not consider other features in XML schema languages, for doing so will make the algorithm extremely complicated and hard to understand. More specifically, we will not consider ID/IDREF constraints in DTD, and sub-typing and key/foreign key constraints in XML schema.

**View Definition Language** We will use a subset of XQuery as the view definition language described as follows:

1. The XQuery we consider could have FOR, WHERE and RETURN clauses and dirElemConstructor [1] in the statement.

2. In each FOR clause, there can be multiple variable binding statements.
3. In an XPath expression, multiple ”//” and ”|” can exist. Further, a node test [1] can be specified as a wildcard.
4. RETURN can contain nested XQuery statements.

Even though we consider WHERE clause, the predicates specified in the WHERE clause are not used to determine whether a view update is translatable. Though considering such predicates might result in more view updates being translatable, it can be handled similarly as in relational scenario and we want to focus on the unique XML features. Also, the LET clause is not considered as an XQuery that uses LET can be rewritten into one without the LET clause. Similar to SQL solutions, we do not consider aggregation, user-defined functions and Orderby clauses.

**Restrictions on Translations Considered** There are various strategies for translating view updates. For those base XML elements corresponding to the view element to be deleted, we can set its value to null, or delete it but keep its descendants, etc. However, we consider only the translations where we delete an XML view element by deleting the corresponding base elements and also the descendants. This keeps the problem tractable, and is similar to existing solutions in SQL/relational scenarios. Now the problem we study can be described as:

**Problem Statement:** Let $Schema(D)$ be an XML schema and $Q$ a view query over $Schema(D)$. Given a view schema node $n$, $n \in Q$, does there exist a translation for deleting a view element whose view schema node is $n$ that is correct for every instance of $Schema(D)$?

Note that we study the problem with schema level analysis, which utilizes the view definition and the schema of the base XML data sources. In other words, we do not examine the base data to determine whether there exists a translation. Such schema level analysis is similar to the approach in relational scenarios [5, 9]; data level analysis for the view update problem has been studied in [4].

## 4 Notations

In this section we first introduce some concepts and notations which are the foundation of later discussions. A summary of them can be found in Table 1 [2]. Let $D$ be an XML document(base XML data sources) with schema $Schema(D)$. $Schema(D)$ can be represented as a tree called the base schema tree, denoted as $ST_{Base}$. The $ST_{Base}$ of the XML Document in Figure 1 is shown in Figure 6 [3].

The XML view is defined as a query $Q$ over $Schema(D)$. The corresponding instance is denoted as $V$. $Q$ specifies a view schema tree, denoted as $ST_{View}$, such as Figure 2(b), Figure 3(b) and Figure 4(b).

$ve_i$ is a view element in $V$ that is to be deleted. The node in $ST_{View}$ corresponding to $ve_i$ is called the view schema node of $ve_i$, denoted as $SN_{View}(ve_i)$. Let us consider the view element $course_1$ in Figure 2(c), $SN_{View}(course_1)$ is the node $course$ in Figure 2(b).

[2] $SN_{View}$ stands for View Schema Node and $ST_{View}$ stands for View Schema Tree. $SN_{Base}$ and $ST_{Base}$ are analogously defined for the base XML document.

[3] Note there is some information not captured by $ST_{Base}$ such as order of elements. We only capture those information that will be utilized by our algorithm, such as cardinality constraints and recursive types.

| Notations | Semantic Meaninig | Notations | Semantic Meaning |
|---|---|---|---|
| $D$ | XML data sources | $Q$ | XQuery Statement defining the view |
| $Schema(D)$ | XML schema of $D$ | $V$ | view instance defined by $Q$ |
| $ST_{Base}$ | schema tree of XML data sources | $ST_{View}$ | schema tree of $Q$ |
| $SN_{Base}$ | a node in $ST_{Base}$ | $SN_{View}$ | a node in $ST_{View}$ |
| $be_j$ | a base element in $D$ | $ve_i$ | a view element in $V$ |
| $source(ve_i)$ | a base element that contribute to the existence of $ve_i$ | $sources(ve_i)$ | All base elements that contribute to the existence of $ve_i$ |
| $Source(ve_i)$ | a $SN_{Base}$ that contributes to the existence of $ve_i$ in $V$ | $Sources(ve_i)$ | all the $SN_{Base}$ that contribute to the existence of $ve_i$ |
| $des(source)$ | The set of base elements that are the descendants of *source* and *source* itself | $Des(Source)$ | The set of schema nodes that are the descendants of *Source* and *Source* itself |

**Table 1.** concepts and notations summary

Let us examine the view element $course_1$ in Figure 3(c) again. It exists in the view only when the following two conditions are both satisfied:

1. In the base XML document, there exists one $pre$ element, demonstrated as $pre_a$, and one $course$ element, denoted as $course_b$.
2. The $course_b$ element is a descendant of the $pre_a$ element.

$course_1$ in Figure 3(c) exists because of $pre_a$ and $course_b$ in base XML Document. Deleting any one of these base elements will lead to deleting $course_1$. Therefore, these base elements are considered as candidates for deleting $course_1$. Let us now define those candidates [4].

Given a $SN_{View}(ve_i)$ in $ST_{View}$, every XPath expression that appears on the path from the root till $SN_{View}(ve_i)$ in $ST_{View}$ corresponds to a base schema node, which is called a $Source$ and denoted as $Source(ve_i)$. The name indicates that it is a way to delete the view element. The set of all such XPath expressions is denoted as $Sources(ve_i)$.

For example, in Figure 7(c), let us consider the view element $name_1$. According to Figure 7(b), There are four path expressions from the $root$ till $name_1$, which are $Document("base.xml")//department$, $\$dept//professor$, $\$prof/student$, $\$student/name$. Therefore, $Sources(name_1)$ = $\{department, professor, student, name_{student}\}$.

For each $Source(ve_i)$, there exists a set of base elements $I(Source(ve_i))$ in $D$ corresponding to it. In $I(Source(ve_i))$, there exists one base element contributing to the existence of $ve_i$ and we call this a $source$, denoted as $source(ve_i)$. For example, in Figure 7(c), $sources(name_1)$ is $\{department, professor, student_a, name_a\}$.

Note while we can delete a source to delete its corresponding view element, it is possible that some other view elements got unexpectedly affected because of this update, which are normally called side-effects. There are two kinds of side-effects. The first kind of side-effects is a descendant of $source(ve_i)$ is a source of another view element. For example, we may want to delete $course_a$ in Figure 1 to delete $course_1$

[4] In fact, deleting an ancestor of any of these base elements can be considered as a candidate for deleting $course_1$ also. Doing this, however, will delete some base elements that are not required to get updated. Further this does not affect translatability. Therefore, we do not include them in our candidates.

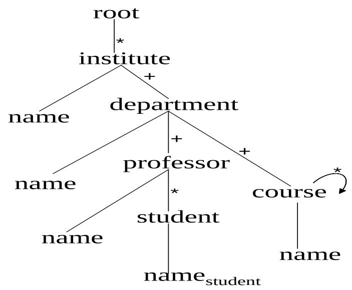


**Fig. 6.** base schema of $D$

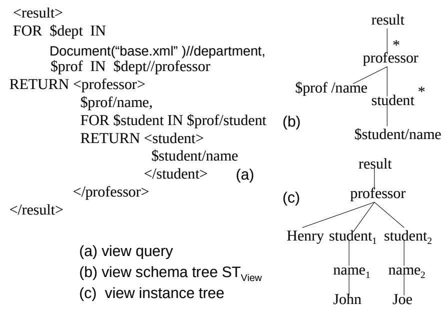


**Fig. 7.** Query $Q_4$ and corresponding view

in Figure 3(c), as $course_a$ is a source of $course_1$. However, $course_b$, which is a descendant of $course_a$, is the source of $course_2$ in Figure 2(c). Therefore, such update will cause side-effects over view element $course_2$,as one of its sources get deleted. The second kind is $source(ve_i)$ is also a source of another view element. For example, $course_b$ in Figure 1 is the source of $course_2$ in Figure 3(c). However, it is also a source of $course_3$. If we want to delete $course_b$ to delete $course_2$, there will be side-effects over $course_3$, as one of its sources get deleted.

Our goal is to find, given a view element $ve_i$, whether there exists a non-empty subset of $sources(ve_i)$ such that deleting any source $source(ve_i)$ in this subset will delete $ve_i$ without affecting any other non-descendant view element of $ve_i$. Deleting $source(ve_i)$ does not affect $ve_j$ if $des(source(ve_i)) \cap sources(ve_j) = \emptyset$. Based on the above concepts, the definition of correctly translating the deletion of a view element problem can be refined as:

**Problem Statement:** Let $Schema(D)$ be an XML schema and $Q$ a view query over it. Given a view schema node $n$, does the following condition hold for every instance of $Schema(D)$ whose corresponding view instance is $V$: For any element $ve_i$, whose schema node is $n$, does there exist $source(ve_i)$ such that $\forall\ ve_j \in V$, $ve_i \neq ve_j$ and $ve_j$ is not descendant of $ve_i$, where des($source(ve_i)$) $\cap$ $sources(ve_j)$ = $\emptyset$.

# 5 Algorithm Analysis

## 5.1 A Naive Algorithm

Using the above concepts, we can observe the following. Consider deleting a view element $ve_i$ by deleting a certain base element $source(ve_i)$. Let this element correspond to the base schema node $Source(ve_i)$. Consider all base schema nodes that could be descendants of $Source(ve_i)$, basically $Des(Source(ve_i))$. If none of these nodes form a $Source(ve_j)$, then deleting $source(ve_i)$ will not affect $ve_j$. This is stated below.

**Lemma 1.** *Deleting a* $source(ve_i)$ *will not affect view element* $ve_j$*, if* $Des(Source(ve_i))$ $\cap$ $Sources(ve_j)$ = $\emptyset$.

For example, consider $course_2$ and $course_3$ in Figure 3(c). Suppose we want to delete $course_2$. As $course$ in Figure 1 is a $Source(course_2)$, $Des(course) \cap Sources$ $(course_3)$ = $\{course, pre\}$ which is not empty. This implies if we delete $course_2$, some base elements contributing to the existence of $course_3$ may also get deleted and therefore there may exist side-effects on $course_3$, which gives the same result as in our previous analysis.

Using Lemma 1, we can come up with a naive algorithm. Let $sum$ be the union of $Sources$ of every non-descendant view element $ve_j$ of $ve_i$, $ve_j \neq ve_i$. If there exists $Source(ve_i)$, such that $Des(Source(ve_i)) \cap sum = \emptyset$, $Source(ve_i)$ is a correct translation of deleting $ve_i$.

However, this algorithm cannot be applied for all view elements. Consider view elements whose view schema nodes are the same. $SN_{View}(ve_i)$, such as $student_1$ and $student_2$ in Figure 7(c). If we want to delete $student_1$, it is easy to observe that we can delete the $student_a$ element in the base document, corresponding to the base schema node $student$ in Figure 1. However, according to the above lemma, $Des(student) \cap Sources\ (student_2) \neq \emptyset$ and thus $student_1$ cannot be updated.

Also, Lemma 1 cannot be applied to detect side-effects on view elements whose schema nodes are descendants of $SN_{View}(ve_i)$. Because for such a view element $ve_j$, we have $Sources(ve_i) \subseteq Sources(ve_j)$, as all the base schema nodes that contribute to the existence of $ve_i$, also contribute to the existence of every view element that is the descendant of $ve_i$. For the above two cases, we need other strategies.

Though Lemma 1 cannot be applied to the above two types of view elements, it can still be applied to detect side-effects on nodes whose schema nodes are non-descendants of $Source(ve_i)$.

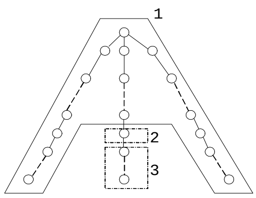


**Fig. 8.** Schema Tree Structure

We therefore partition the view schema tree into three parts, as shown in Figure 8. Let $n = SN_{View}(ve_i)$ be the view schema node for $ve_i$. The first group, marked as 1, is view schema nodes that are non-descendant of $n$. We can apply Lemma 1 to detect side-effects on view elements whose schema nodes are in this group. The second group, marked as 2, is view schema node $n$ itself. We discuss how to detect side-effects on view elements whose schema node is in this group in Section 5.2. The third group, marked as 3, is schema nodes that are descendants of $n$. We discuss how to detect side-effects on view elements whose schema nodes are in this group in Section 5.3. Also, these three groups cover all schema nodes without any overlap. Thus we check all view elements for side-effects effectively, and a correct translation is returned if there exists one.

### 5.2 Detecting Side-Effects in Group 2

Here we check view elements that share the same view schema node as $ve_i$, the view element to be updated. This is similar to the relational view update problem, and we can utilize the solutions from the relational scenario.

**Updating Relational Views** In [9], Keller proposes an algorithm to check whether there is a 1-1 mapping between the set of tuples in the relational view and the set of tuples in a base relation. This algorithm can be used to check whether we can delete a tuple in the view without side-effects in the relational scenario. We use Keller's algorithm as the basis for studying view updates in XML scenario as well. Therefore, in this section, we will introduce and discuss this algorithm.

**Keller's Algorithm**: Given a relational database $D$ and a relational view $V$, in order to find all possible relations $r_1, r_2, \ldots, r_i$ such that there is a 1-1 mapping between the set of tuples in $V$ and the set of tuples in every $r_p$, $1 \leq p \leq i$, construct a directed graph, also called as a **trace graph**, as:

1. every relation used by the view forms a node in the graph. Suppose there are nodes $r_1, r_2, \ldots, r_n$ in the graph.
2. let $r_i, r_j$ be two nodes ($r_i \neq r_j$). There is an edge $r_i \rightarrow r_j$ iff there is a join condition of the form $r_i$.a = $r_j$.k ($r_j$.k is the key for $r_j$. If there is a $r_i$.k = $r_j$.k join, then there are two edges $r_i \rightarrow r_j$ and also $r_j \rightarrow r_i$.).

If there is any node $r$ which can reach all other nodes, then there is a 1-1 mapping from tuples in $V$ to tuples in the relation which is denoted by node $r$. □

**Adapting Keller's Algorithm to XML scenario** In Keller's Algorithm, an edge $r_i \rightarrow r_j$ represents that a tuple in $r_i$ joins with at most one tuple in $r_j$. The same intuition can be applied to XML scenario. Given view element $ve_i$, its trace graph has a $root$ element and one node for every $Source(ve_i)$. Let $Source_i$, $Source_j \in Sources(ve_i)$. We draw an edge from $Source_i$ to $Source_j$ if the XPath expression of $Source_i$ starts with the variable representing $Source_j$. We draw an edge from $Source_i$ to $root$ if the XPath expression of $Source_i$ starts with $Document("base.xml")$. Let us consider element $student$ in Figure 7(b); $Sources(student)$ = $\{department$, $professor$, $student\}$. The corresponding XPath expressions are $Document("base")//department$, \$$dept$ $//professor$, \$$prof/student$ respectively. Every $professor$ will join with at most one $department$. Similarly, every $student$ is guaranteed to join with at most one $professor$. According to Keller's algorithm, there are four nodes in the trace graph: $root, department, professor$ and $student$. We can draw an edge from $student$ to $professor$, one from $professor$ to $department$ and one from $department$ to $root$. $student$ can reach all the other nodes. This implies we can delete view element $student_1$ by deleting base element $student_1$ in $D$, as analyzed before.

However there are differences between relational and XML scenarios. For instance, a node in the trace graph that does not reach all other nodes can still be a correct translation. Consider view schema node $prof$-$student$ in Figure 4(b). A view element of $prof$-$student$ has $Sources = \{professor, student\}$, without any edge between them in the trace graph. However, as base schema in Figure 4(a) implies that there is only one $professor$ element in the base, any view element whose schema node is $prof$-$student$ can be deleted by deleting a base element whose schema node is $student$. So cardinality constraints should be considered to determine whether a $Source$ can be a correct translation.

On the other hand, a node in the trace graph that reaches other nodes might not be a correct translation. Consider $course_1$ in Figure 3(c), $Sources(course_1) = \{pre, course\}$. In the trace graph there is an edge from $course$ to $pre$. However, $course_1$ cannot be deleted by deleting $course_b$ in Figure 1. This is because $course_c$ is a descendant of $course_b$ and is $source$ of both $course_2$ and $course_3$. Also $course_2$ in Figure 3(c) cannot be deleted because it shares the same source as $course_3$. Both of these occur because of recursive types in XML.

In the rest of the section, we study how we can extend Keller's algorithm to handle cardinality constraints and recursive types in XML.

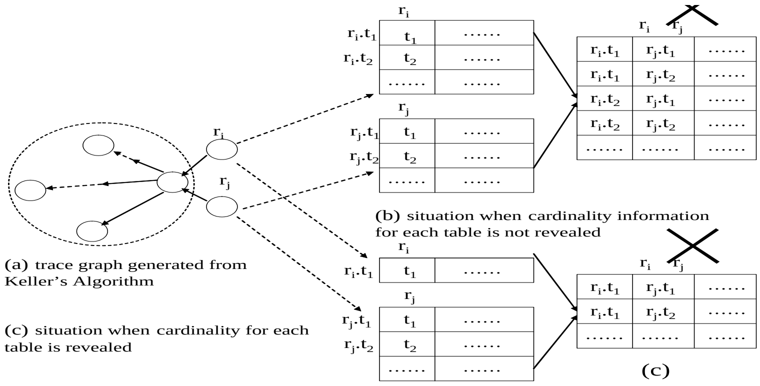


**Fig. 9.** Keller's algorithm and cardinality constraints

**Handling Cardinality Constraints** How cardinality information impacts the translatability of view updates in relational scenario is illustrated in Figure 9, where $r_i$ and $r_j$ can reach all other nodes except each other. Without any cardinality information, a view tuple cannot be deleted either from $r_i$ or $r_j$, as there can be side-effects shown in Figure 9(b). However, if we know the cardinality information that there is only one tuple in $r_i$ [5], then view tuples can be deleted from $r_j$, shown in Figure 9(c).

While such cardinality information cannot be specified easily in relational schema, it does exist in XML schema, as we mentioned in section 3.2. We only capture cardinality constraints *, 1 and 0. Note XML schema can specify more complex cardinality constraints such as MaxOccurs and MinOccurs. However they do not affect whether a view element can be updated or not. So we ignore them in this paper.

Given two base schema nodes $t$ and $t_n$ which are of ancestor-descendant relationship, however, what is the cardinality between them? Here we give the formal definition:

**Definition 2.** *Let $t/a_1 :: t_1/a_2 :: t_2/\ldots/a_n :: t_n$ be a path expression between two nodes $t$ and $t_n$ in the base schema, where $\forall a_i, 1 \leq i \leq n$, can be child, descendant-or-self, or attribute. The cardinality $card(t, t_n)$ between $t$ and $t_n$, which can also be denoted as $card(t, /a_1 :: t_1/a_2 :: t_2/\ldots/a_n :: t_n)$, is defined as:*

1. *if n >1, $card(t, /a_1 :: t_1/a_2 :: t_2/\ldots/a_n :: t_n)$ = $card(t, /a_1 :: t_1) \times card(t_1, /a_2 :: t_2) \times \ldots \times card(t_{n-1}, /a_n :: t_n)$. For the multiplication, please refer to Figure 10.*
2. *if n=1:*
   (a) *if $a_1$ is descendant-or-self, $card(t, /a_1 :: t_1)$ = *.*
   (b) *if $a_1$ is attribute, $card(t, /a_1 :: t_1)$ = 1.*
   (c) *if $a_1$ is child, and the content model of $t$ is $re$. Then $card(t, /a_1 :: t_1)$ = $cardRE(t_1, re)$. $cardRE(t_1, re)$ is defined as follows:*
      i. *if $re$ = $(re_1, re_2)$, $cardRE(t, re)$ = $cardRE(t_1, re_1)$+ $cardRE(t_1, re_2)$.*
      ii. *if $re$ = $(re_1 \mid re_2)$, $cardRE(t_1, re)$ = $max\{ cardRE(t_1, re_1), cardRE(t_1, re_2)\}$.*
      iii. *if $re = (re_1)*$, $cardRE(t, re)$ = $cardRE(t_1, re_1) \times *$.*
      iv. *if $re = t_i$:*
         A. *if $t_i = t_1$, then $cardRE(t_1, re)$ = 1.*
         B. *if $t_i \neq t_1$, then $cardRE(t_1, re)$ = 0.*

| × | 1 | 0 | * |
|---|---|---|---|
| 1 | 1 | 0 | * |
| 0 | 0 | 0 | 0 |
| * | * | 0 | * |

Cardinality Multiplication Table

| + | 1 | 0 | * |
|---|---|---|---|
| 1 | * | 1 | * |
| 0 | 1 | 0 | * |
| * | * | * | * |

Cardinality Addition Table

**Fig. 10.** cardinality tables

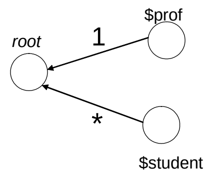


**Fig. 11.** trace graph of $prof$-$student$ in Figure 4(b) with cardinalities

Consider Figure 6, cardinality between $root$ and $department$ can be computed as $card(root, /child :: institute/child :: department) = card(root, /child :: institute)$ $\times card(institute, /child :: department) = *$.

Our proposition below uses the cardinality information in the base schema for deciding whether a base element is a correct translation of deleting the required view element.

**Proposition 1.** *Given $Sources(ve_i)$, draw the trace graph according to Keller's algorithm. Suppose there are n 0-indegree nodes in the trace graph, say $r_1, r_2, \ldots, r_n$. Among $Sources(ve_i)$, find one that is the lowest common ancestor of all 0-indegree nodes, denoted as $SN_{ancestor}$. For each $r_i$, $card(SN_{ancestor}, r_i)$ is called the relative cardinality of $r_i$. Let the number of relative cardinalities whose value is 1 be $l$.*

1. *if $l = n$, we can delete $ve_i$ from any $source(ve_i)$ whose corresponding node in trace graph has 0-indegree.*
2. *if $l = n - 1$, we can delete $ve_i$ by deleting the $source$ whose base schema node is the 0-indegree node with cardinality as "*".*
3. *if $l \leq n - 2$, there is no correct translation.*

Let us consider the query in Figure 4 again. Figure 11 is the trace graph of $prof$-$student$ in Figure 4(b). With Definition 1, $card(result, professor)$= 1, $card(result, student)$ = *. Therefore, to delete the view element whose view schema node is $prof$-$student$, we can delete from Source $student$.

**Handling Recursive Type** Let us first consider the side-effects where $source(ve_j) \in des(source(ve_i))$, $ve_i$ and $ve_j$ share the same view schema node. Consider $course_1$ in Figure 2(c). Deleting it will have side-effects because some descendants of its source, $source_a$, also contribute to the existence of other view elements, such as $course_2$. To identify such side-effects, we define *recursive Source* as below.

**Definition 3.** *Let $Schema$ be an XML schema and $Q$ a view query defined over this schema. Let $S$ be a $Source$ for a view element whose view schema node is $n$. $S$ is said to be a recursive Source if $\exists D$, an XML Document confirming to $Schema$, where the conditions below are all satisfied:*

1. *there exist two view elements in $Q(D)$, $ve_i$ and $ve_j$, such that $i \neq j$ but $SN_{View}(ve_i) = SN_{View}(ve_j) = n$.*
2. *$I(S)$ contains $be_i$ and $be_j$, $be_i$ and $be_j$ is $source$ of $ve_i$ and $ve_j$ respectively, and they have ancestor-descendant relationship.*

[5] This is a quite strict requirement, which will be relaxed in later discussions.

One might think that if a path expression for a Source has "//" operation, then the Source is recursive. However, this need not be the case, such as in the XPath expression $Document("base.xml")//department/course$. To identify *recursive Source*, we define $AbsoluteXPath$ below.

**Definition 4.** *The path in the trace graph from $Source$ to $root$ is called a branch, denoted as $branch_{Source}$. The XPath expression obtained by concatenating all the XPath expressions in $branch_{Source}$ is called the absolute XPath of $Source$.*

To identify whether a Source is recursive, we check its absolute XPath. If the absolute XPath retrieves two base elements that have ancestor-descendant relationship, then the Source is recursive.

**Proposition 2.** *Let $P$ be the absolute XPath of a $Source(ve_i)$ for view element $ve_i$. We call $Source(ve_i)$ as recursive iff the following two conditions are both satisfied:*

1. *$P$ is of the form $/P_1//be_{re}/P_2/be_l$, where $P_1$, $P_2$ are path expressions and $be_{re}$, $be_l$ are base schema nodes.*
2. *the last base element $be_l$ in $P$ can have $be_{re}$ as its descendant.*

Proposition 2 is illustrated in Figure 13(a). Here both the $be_l$'s satisfy $P$ and have ancestor-descendant relationship. The $Source$, $student$, for a $student$ view element in Figure 7 has the absolute XPath $Document("base.xml")//department//professor/student$, which does not match Proposition 2, therefore $student$ is not recursive. However the $Source$, $course$, for a $course$ view element in Figure 2 has the absolute XPath $Document("base.xml")//course$. This matches Proposition 2 where $P_1$ is $Document("base.xml")$, $P_2$ is empty and $be_{re}$ = $be_l$ = $course$, and $course$ has $course$ as descendant.

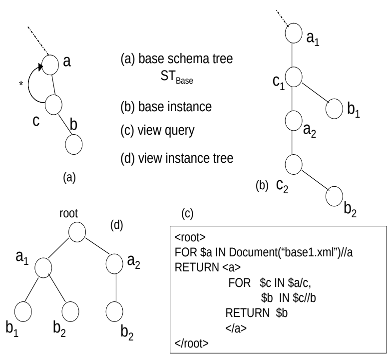


**Fig. 12.** $ST'_{Base}$, Query $Q_4$

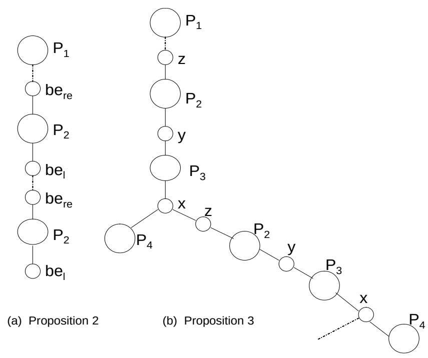


**Fig. 13.** Illustrating Proposition 2 and Proposition 3

Now let us consider the second type of side-effects, where $source(ve_i)$ is also $source(ve_j)$. Consider the query in Figure 3(a). $course_c$ in Figure 1 contributes to two view elements, $course_2$ and $course_3$, in Figure 3(c). A more general example is shown in Figure 12. Figure 12(a) is the base schema and Figure 12(b) is one possible instance. Based on the query in Figure 12(c), we have the view instance tree shown in Figure 12(d). Specified by the query, $b_2$ joins with $a_1$ and $a_2$ and thus appears multiple times in the view. Deleting any of them may cause side-effects over other appearances of $b_2$. For such situations we have the following proposition:

**Proposition 3.** *Consider the trace graph of a view element whose view schema node is $n$. Let $Source_1$ and $Source_2$ be two Sources in this trace graph, with an edge from $Source_2$ to $Source_1$. $I(Source_2)$ may contain a base element that is the source of two view elements, $ve_1$ and $ve_2$, iff all the following conditions below are satisfied:*

1. *The absolute XPath of $Source_1$ is of the form $P_1//z/P_2/y$. Let $y$ be the variable that $Source_1$ binds to and $Source_1$ is marked as recursive using Proposition 2.*
2. *The absolute XPath of $Source_2$ is of the form $\$y/P_3//x//P_4$.*
3. *$z \in Des(x)$.*

Figure 13(b) illustrates Proposition 3. Here, there are two view elements where $Source_2$ binds to the rightmost $P_4$, and where $Source_1$ binds to the two different $y$'s.

Actually this scenario implies a much stronger condition: there exists no correct translation for deleting the view element. Let us examine this. First of all, no $Source_i$ that can reach $Source_2$ can be a correct translation, as an instance of $Source_i$ can be the source of two different view elements. Now, consider a $Source_i$ that cannot reach $Source_2$. Since this node cannot reach $Source_2$, we must consider cardinality constraints. Let $Source_{21}$ be a 0-indegree that reach $Source_2$. As the lowest common ancestor of all 0-indegree nodes, $SN_{ancestor}$, must be a node in the path from root to $Source_1$, $card(SN_{ancestor}, Source_{21})$ = *. Thus $Source_i$ can never be a correct translation. This is stated in the corollary below:

**Corollary 1.** *Consider the trace graph of view element $ve_i$. If $\exists Source_1, Source_2$ in this graph that satisfy Proposition 3, there is no correct translation for deleting $ve_i$.*

With Proposition 1, Proposition 2 and Proposition 3, we can detect all the possible side-effects on view elements whose schema node is in Group 2 when deleting $Source(ve_i)$. Please refer to Section 6 for how to integrate them.

### 5.3 Detecting Side-Effects in Group 3

In this section, we will discuss how to detect side-effects on view elements whose schema nodes are descendants of $n$. Note view elements that are descendants of $ve_i$ will get deleted with $ve_i$, according to the hierarchial structure of XML view. Therefore, we focus on whether any view element, $ve_j$, that are descendants of siblings of $ve_i$, gets affected when deleting $source(ve_i)$.

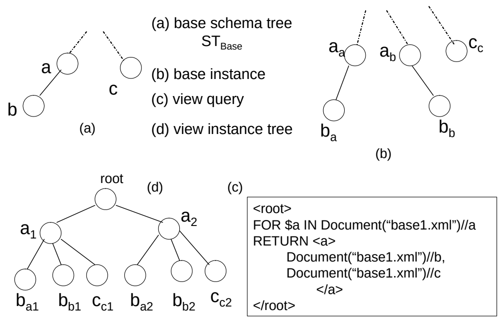


**Fig. 14.** $ST'_{Base}$, $Query\ Q_5$

Figure 14 illustrates side-effects on Group 3. If we delete $a_1$ in Figure 14(d) by deleting $a_a$ in Figure 14(b), then the view element $b_{a2}$, the descendant of $a_2$ in Figure 14(d) is deleted. This is a side-effect. This happens because view element $b_{a2}$ has a $source$, $b_a$, which is the descendant of $source(a_1)$. On the other hand, there is no side-effects on view element $c_{c2}$.

We identify such side-effects as follows. Let $ve_j$ be a descendant of sibling view element of $ve_i$. If $Source(ve_j)$ is not a descendant of $Source(ve_i)$, we need not consider it as it will never get affected. Consider $c_{c2}$ in Figure 14(d) as $ve_j$ and $a_1$ as $ve_i$. As $c$ is $Source(ve_j) \notin Sources(ve_i)$ and also $c$ is not a descendant of $Source(ve_i)$, $a$, no side-effect on view element $c_{c2}$ will appear.

On the other hand, if $Source(ve_j)$ is descendant of $Source\ (ve_i)$ or itself, $source\ (ve_j)$ must contribute to at most one view element that must be a descendant of $ve_i$. This implies there should need an edge from $Source(ve_j)$ to $Source(ve_i)$ in the trace graph of $ve_j$. Consider $b_{a2}$ as $ve_j$ and $a_1$ as $ve_i$. As $Source(ve_j)$, $b$, is a descendant of $Source(a_1)$, there needs to be an edge from $b$ to $a$ in the trace graph of $ve_j$, which actually does not exist. Therefore, there may be side-effects on $b_{a2}$. The above conclusions are formalized in the following lemma:

**Lemma 2.** *For every descendant element $ve_d$ of $SN_{View}(ve_i)$, get its trace graph. Suppose there are n 0-indegree nodes that cannot reach $Source(ve_i)$, say $r_1, r_2, \ldots, r_n$. For some $ve_d$, if $\exists\ r_i$ such that $SN_{Base}(r_i) \in Des(SN_{Base}(Source(ve_i)))$, $Source\ (ve_i)$ cannot be the correct translation of deleting $ve_i$.*

# 6 Algorithm for Correctly Deleting Single View Element in XML Scenario

In this section, we will present the three-step algorithm for finding the correct translation of deleting a view element $ve_i$.

**Step 0**:

0. $Candidates$ = $Sources(ve_i)$

**Step 1**:

1. Let $Sources'$ be the union of $Sources$ of all non-descendant view elements of $ve_i$.
2. For every $Source(ve_i) \in Candidates$, if $Des(Source(ve_i)) \cap Sources' \neq \emptyset$, $Candidates$ = $Candidates - Source(ve_i)$.
3. If $Candidates = \emptyset$, the algorithm terminates; else go to Step 2.

**Step 2**:

4. Draw the trace graph of $ve_i$ and let $Sources_{Keller}$ be the set of 0-indegree nodes.
5. Use Proposition 1 to check $Sources_{Keller}$. Let $l$ be the number of nodes whose relative cardinality is ”1”.
   (a) if $l = n - 1$, $Sources_{Keller} = \{SN_{rest}\}$, where $SN_{rest}$ is the only schema node in $Sources_{Keller}$ whose relative cardinality is ”*”.
   (b) if $l \leq n - 2$, $Candidates = \emptyset$; the algorithm terminates.
6. Use Proposition 2 to check if $Source(ve_i)$ is recursive. If so $Candidates = Candidates - Source(ve_i)$.
7. For every branch of the trace graph, find two consecutive Sources that satisfy the condition in Proposition 3. If there exists such two Sources, $Candidates = \emptyset$; the algorithm terminates.
8. $Candidates = Candidates \cap Sources_{Keller}$. If $Candidates = \emptyset$, the algorithm terminates; otherwise go to Step 3.

**Step 3**:

9. For every $Source \in Candidates$, if deleting $Source$ has side-effects on a descendant according to Lemma 2, $Candidates = Candidates - Source$.
10. The algorithm terminates. If $Candidates = \emptyset$, there is no correct translation of deleting $ve_i$; otherwise each $Source \in Candidates$ is a correct translation.

**Theorem 1.** *After the above algorithm, if $Sources(ve_i)'$ is empty, deleting $ve_i$ is untranslatable. Otherwise deleting $\forall source \in sources'(ve_i)$ is a correct translation of deleting $ve_i$.*

## 7 Conclusion

In this paper we presented an algorithm for correctly translating the deletion of an XML view element as deleting an element in the underlying XML base. Our algorithm uses a schema-level analysis to efficiently find a correct translation and it is based on the previous work for updating relational views, extending this with recursive types and cardinality constraints in XML, and "//" operator in XQuery. Our algorithm is sound and complete.

This paper forms a first major step in studying view updates in XML scenario. Future work needs to consider incorporating other update operations such as insert, replace and XML specific operations and considering updating multiple elements. Further, we need to consider more semantics both in XML Schema and XQuery statements.

## References


1. http://www.w3.org/xml/query/. 2006.
2. S. Abiteboul. On views and xml. *PODS*, 1999.
3. V. Braganholo, S. Davidson, and C. Heuser. the updatability of xml views over relational databases, June 2003.
4. Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *The VLDB Journal*, pages 471–480, 2001.
5. U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, volume 7(3), pages 381–416, Sept 1982.
6. F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems (TODS)*, pages 557–575, 1981.
7. M. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. http://www.www9.org/w9cdrom/202/202.html, May 2000.
8. International Organization for Standardization (ISO) & American National Standards Institute (ANSI).
9. A. M. Keller. Algorithms for Translating View Updates to Database Updates for View Involving Selections, Projections and Joins. In *Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.
10. H. Kozankiewicz, J. Leszczyowski, and K. Subieta. Updatable xml views. *Advances in Databases and Information Systems*, pages 381–399, September 2003.
11. P. Lehti and P. Fankhauser. Towards type safe updates in xquery. *http://www.ipsi.fhg.de/ lehti/Typing*
12. *Oracle Technologies Network. Using XML in Oracle Database Applications. http://technet.oracle.com/tech/xml/htdocs/about_oracle_xml_products.htm, November 1999.*
13. *M. Rys. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In* VLDB, *pages 465–472, 2001.*
14. *R. Vercammen. Updating xml views.* VLDB PhD Workshop, *page 6ł10, 2005.*
15. *L. Wang, E. A. Rundensteiner, and M. Mani. UFilter: A Lightweight XML View Update Checker. In* ICDE, poster paper, *2006.*