Updating Views Over Recursive XML

by

Ming Jiang

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

December 2007

APPROVED:

Professor Murali Mani, Thesis Advisor

Professor Dan Dougherty, Thesis Reader

Professor Michael A. Gennert, Head of Department

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.

There are two kinds of view updates: single view element update, where the user requires for an update over a particular view element, and a set of view elements update, where the user requires for an update over all view elements that satisfy a given XPath over the view. Accordingly, we propose one solution for each kind of view update problems 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.

Acknowledgements

This thesis is by far the most significant scientific accomplishment in my life and it would be impossible without people who supported me and believed in me.

Most of all I would like to thank my research advisor, Professor Murali Mani, whose passion for research, teaching and scientific exploration inspired me in more ways than I can count.

He taught me new ways to look at research and encouraged creativity, took the time to read and carefully comment on my papers, and gave me encouragement and practical advice when I needed it most. When I got stuck with problems, he always discussed with me patiently. His unlimited energy, dedication and enthusiasm was contagious. It was a great privilege to work with him on this thesis.

I have learned a great deal from my fellow graduate students and friends in DSRG and WPI. I particularly want to thank Ling Wang for introducing me so many valuable papers as I first step into XML view update areas and discussing probable solutions for different cases with me. I also want to thank Song Wang, Mr. Know-all who answered me countless questions. I also want to thank Bin Liu, Ming Li, Mingzhu Wei, Luping Ding and others for challenging, on-going exchanges of ideas.

I would also like to thank all of my professors at WPI for inspiring me to question things, think critically, and challenging me. In particular, I would like to thank Professor Dan Dougherty, my Thesis Reader, and Professor Elke A. Rundensteiner for regularly checking on my research progress.

My deepest thanks goes to my girl friend Yingying Lang for her unconditional love and support in everything.

I want to thank my parents for loving me more than anything in this world.

Contents

List of Figures

List of Tables

Chapter 1

Part I:

Introduction

1.1 Motivation

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* [9]. 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.

There are two kinds of view updates based on how the user specifies the required view elements: single view element update and a set of view element update. Sometimes a user may just want to update a particular view element. Then he may prefer to specifically point out a view element and try to update it. This is called single view element update. There are also situations where the user prefer to update those view elements satisfying certain conditions specified by predicates. In this

case, he may prefer to specify a condition expression and required updates.

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 hierarchical structure and other 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 1.1. 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 1.2 and Figure 1.3.

```
<!DocType root[
<!Element root( institute*)>
<!Element institute (name, department+)>
<!Element department (name, professor+,
   course+)>
<!Element professor( name, student*)>
<!Element student( name)>
<!Element course( name, pre?)>
<!Element pre( course+)>
<!Element name( #PCDATA)>]>
<root>
  <institute>
    <name> WPI </name>
    <department>
      <name> CS</name>
      <professor>
        <name>  Henry </name>
        <student_a>
          <name_a>John </name>
        </student>
        <student_b>
          <name_b> Joe </name>
        </student>
      </professor>
```

```
<course_a>
  <name>  Database </name>
    <pre_a>
      <course_b>
        <name>
          Algorithm
        </name>
        <pre_b>
          <course_c>
            <name>
              Data Structure
            </name>
          </course>
        </pre>
      </course>
    </pre>
  </course>
</department>
</institute>
</root>
```

Figure 1.1: XML document D with Schema(D)

In Figure 1.2, (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 1.3. [1]

[1]The subscripts *a*, *b*, *c* in Figure 1.1 and *1*,*2*,*3* in Figure 1.2(c) and Figure 1.3(c) are for illustration purpose only.

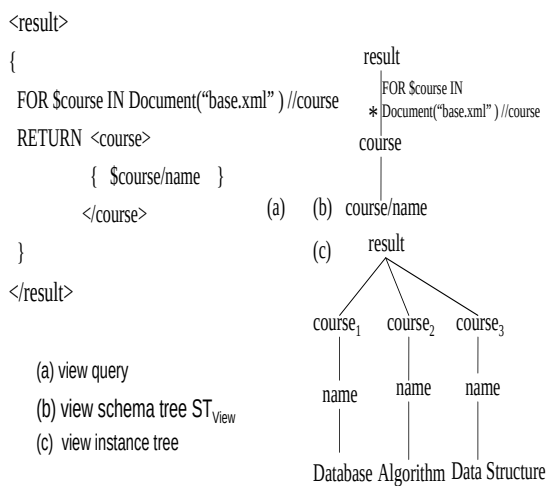


Figure 1.2: Query Q_1 and corresponding view

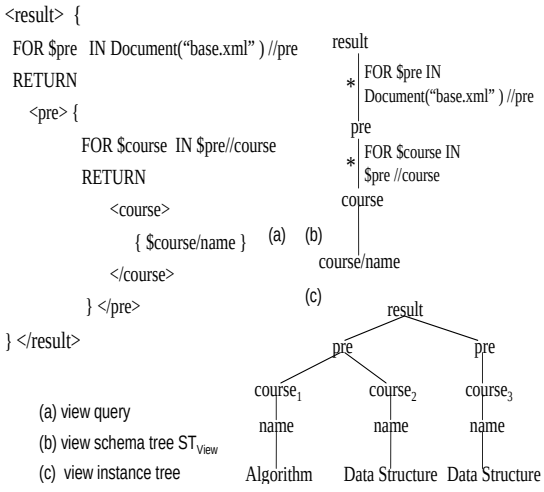


Figure 1.3: Query Q_2 and corresponding view

Let us consider the first kind of view update, e.g. single view element update. A user may want to delete $course_1$ in Figure 1.2(c). If we delete $course_a$ in D, this update would cause $course_2$, $course_3$ and their descendants to be removed in Figure 1.2(c). This is a side-effect and therefore it is not a correct translation. Now let us consider Figure 1.3(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 where 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 1.4(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 1.4(a). It indicates that there is only one $professor$ in the base. We now know that each $student$ will be used only once

They do not appear in the actual documents or views.

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 the thesis.

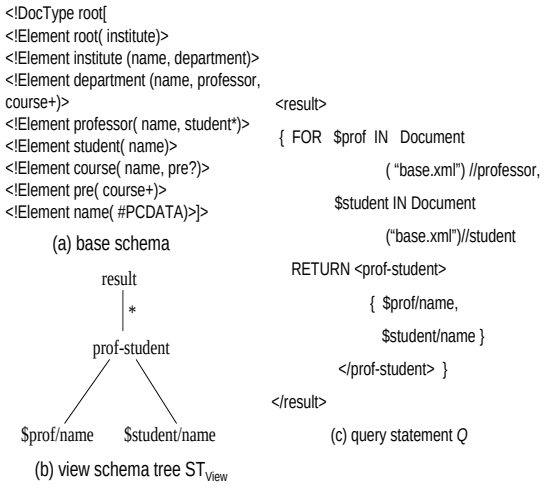


Figure 1.4: Query Q_3 and corresponding view

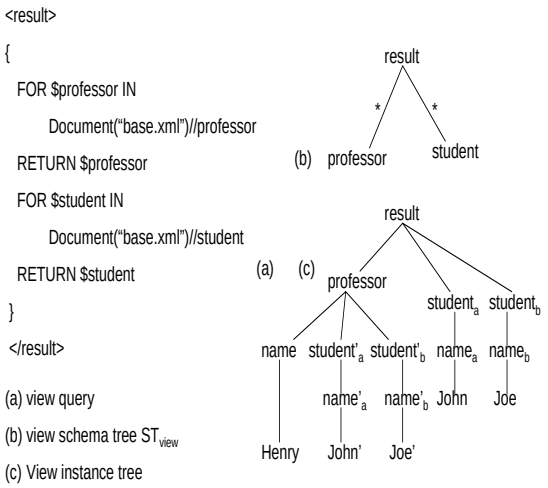


Figure 1.5: Query Q_4 and corresponding view: side-effects over invisible elements

Last but not the least, let us take a look at another example, in which view update problem arises due to the hierarchical structure of XML. Consider the XQuery statement in Figure 1.5(a). It requires to return all the $professor$ and $student$ base elements. For each returned base element, all of its descendants will appear in the view, as we can see in Figure 1.5(c). However, as the view schema tree in Figure 1.5(b) is generated from XQuery, it will display the schema node for $professor$ and $student$ only. Note the base element $student$ appears as two different elements in the view. One of them, $student$, appears because XQuery explicitly extracts the base element $student$; while the other, view element $student'$ [2], appears as XQuery requires to extract the contents of base element $professor$ and base element $student$ is a descendant of $professor$. For the view element $student'$, there is no schema node in Figure 1.5(b) corresponding to it. We call those view elements which has no corresponding view schema node as *invisible elements*. This is quite different from relational scenario, where all the view tuples are visible, in the sense that each of them corresponds to the only view schema node in the view schema tree.

[2] just like subscripts in Figure 1.3, superscripts is for illustration purpose only.

Now let us examine these invisible elements to see if there exist updates that could cause side-effects related to them. As the view elements $student_a$ and $student'_a$ come from the same base element, deleting one of them will have side-effects over the other. This means deleting a visible view element may have side-effects over some invisible elements, and vice versa. In addition, let us consider deleting the view element $name_a$, which is an invisible element. This will obviously have side-effects over view element $name'_a$, which is also an invisible view element. As both the updated view element and affected view element are not visible in the view schema tree, how are we going to detect the side-effects on the schema level needs to be tackled.

View update problems caused by the above three XML features also remain in the scenario where a set of view elements are required to get updated in a batch. Let us examine the difference between scenario in which only one view element needs to get updated and the scenario in which a set of view elements needs to get updated. A naive algorithm can be to check the side-effects and find the translation for each view element update in the set one by one. If any update of the view elements have side-effects, we need to reject the updates, as updating these view elements will cause side-effects. The following example, however, indicates that this is not always the case. Consider a user may want to delete all the course view elements in Figure 1.2(c). Though deleting such a single view element may cause side-effect, as we stated above, the corresponding update of deleting all such view elements is straightforward; we can simply delete all the course elements in the base XML document and this translation causes no side-effects. The reason is those view elements which may have side-effects also belong to the set of view elements that we want to delete. As a result, there is no side-effects on the rest of the view.

Therefore, we can observe that those view elements that may have side-effects, may be able to get updated along with other view elements and leaves no side-effects over the rest of the view. How a required set of view elements can be specified in XML scenario and how to check their update translatability will also be discussed in the thesis.

Our main technical contributions include: we study how features in XML schemas, such as recursive types and cardinality constraints, along with the hierarchical structure of XML, impact

the XML view update problem. We examine two update scenarios, where one is to delete one single view element and the other is to delete a set of view elements. Accordingly, we propose two algorithms. The first one is to determine whether a view update over XML data sources is translatable and to find the translation if one exists, based on schema level analysis. The second one is to determine whether updating a set of view elements specified by some conditions over XML data sources is translatable and to find the translation if one exists, based on schema level analysis. Our algorithms are 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.

1.2 Related Work

There are many studies on view updates in relational scenario, such as [10, 9, 14, 8]. In [10], authors introduce 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 [9], 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 [14], Keller proposes an algorithm to check whether 1-1 mapping exists between a set of view tuples and a set of base tuples. This mapping indicates that a certain view element can be deleted without side-effects. In [18], authors consider the problem of detecting independence of a query expressed by datalog programs from updates.

While in [10, 9, 14] authors study the view update problem on the schema level, there are other works, such as [8], that study the problem on the instance level. Therefore in [8], 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 recent work [15], authors propose a new update semantic for updating relational views. Every update in the view is encoded using special identifier in the database, which ensures the uniqueness of set of base elements generating the updated view element. This uniqueness indicates all view updates are translatable without side-effects. Beyond those special encodings, side effects are hidden in the actual base data via an extended view query. This paper, however, does not study how to check the translatability of the view updates in the context of this special semantic. In [3], authors propose a novel approach to view update problem in relational scenario. It defines a bi-directional query language, in which every expression can be read both as a view definition and as an update policy.

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 [19, 20]. There are some research that consider XML views as compositions of flat relational views, such as [11], for the purpose of querying relational databases. Some other work further study the updatability of XML views over relational databases. In [5], authors study the update over *well-nested* XML views. However, as authors map XML view into relational view updating problem, some of the constraints such as cardinality constraints and recursive types in XML context cannot be captured. In [23], the authors discuss how to check side-effects for updating XML view elements over a relational database. In [4], 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. In [7], authors 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. Also, they provide new techniques to efficiently support XML view updates specified in terms of XPath expressions with recursion and complex filters.

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. In [2], the authors introduce a view architecture and discuss XML view updates for the first time. In [22], authors study both closed and open view update strategies in relational scenarios and discuss

their applicability to an XML setting. In [17], authors study type checking in XML view updates. In [21], the authors study execution cost of updating XML views using triggers versus indices. In [16], authors consider virtual updatable views for a query language addressing native XML databases, including information about intents of updates into view definitions. In [6], authors develop an ER based theory to guide the design of valid XML views, which avoid the duplication from joins and multiple references to the relations. In [13], authors consider finding a correct translation of a given update in a user-defined XML views over XML documents, avoiding side-effects over other view elements. This thesis will extend the algorithm in this paper and try to handle more kinds of views and updates.

Chapter 2

Part II:

Problem Definitions and Notations

2.1 View Update Translatability and Problem Scope

2.1.1 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 2.1. The update is correct if the diagram in Figure 2.1 commutes.

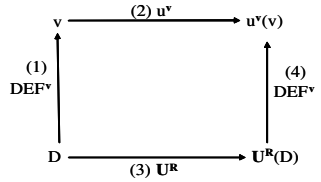


Figure 2.1: Correct View Updates Definition

2.1.2 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 [12] 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. We study both single view element and set of view elements deletions. For single view element deletion, we do not use a view update language, as how the view element is specified (by the view update language) is not significant.

There are two ways to specify the set of view elements which should get updated in a batch. The user can specifically point out every view element that needs to get updated. Those view elements can correspond to different schema nodes. To check the translatability of such updates, we can check the translatability of deleting every view element in the set. Therefore we will not discuss this situation. The second way is the user can specify the elements to be deleted by an XPath expression XP, starting from the root of the view. In the latter situation, any view element ve_i will get updated if the path from root of the view to ve_i qualifies XP. For short, we say these view elements qualify XP. We will use XPath as our view update language when we consider deleting a set of view elements in a batch. As XPath could become quite complicated, we will set constraints on using it, please refer to Chapter 4 for details.

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 because an XQuery that has 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:

2.1.3 Problem Definitions

For single view element deletion, we give the problem definition below:

Deleting Single View Element 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 $Schema(D)$?

For set of view element deletions, we give the problem definition below:

Deleting Set of View Elements Problem Statement: Let $Schema(D)$ be an XML schema and Q a view query over $Schema(D)$. Given an XPath XP, does there exist a translation for deleting the view elements, which qualify XP, 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 [9, 14]; data level analysis for the view update problem has been studied in [8].

2.2 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 2.1 [1]. 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 1.1 is shown in Figure 2.2 [2]. Consequently, every element in $Schema(D)$ has a corresponding schema node in ST_{Base}, denoted as SN_{Base}. For example, the element $professor$ in $Schema(D)$ in Figure 1.1 has a SN_{Base} in Figure 2.2, which is the node $professor$.

[1] 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.

[2] 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 Meaning	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*, including *source*	*Des(Source)*	The set of schema nodes that are the descendants of *Source*, including *Source*

Table 2.1: concepts and notations summary

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 1.2(b), Figure 1.3(b) and Figure 1.4(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 1.2(c), $SN_{View}(course_1)$ is the node $course$ in Figure 1.2(b).

Let us examine the view element $course_1$ in Figure 1.3(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, denoted 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 1.3(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 [3].

[3]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 not considering these ancestors does not affect translatability. Therefore, we do not include them in our candidates.

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 $Sources(ve_i)$. In the rest of the thesis, for an XPath expression, we will use the name of the corresponding base element for short as long as there does not exist any ambiguity.

For example, in Figure 2.3(c), let us consider the view element $name_1$. According to Figure 2.3(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, $Sources(name_1)$ = {$Document$ ("$base.xml$") $//department$, $\$dept//professor$, $\$prof/student$, $\$student/name$}. And we can also write it as $Sources(name_1)$ = {$department$, $professor$, $student$, $name_{student}$} for short.

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 2.3(c), $sources(name_1)$ is $\{department, professor, student_a, name_a\}$.

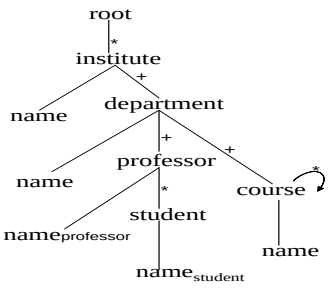


Figure 2.2: base schema of D

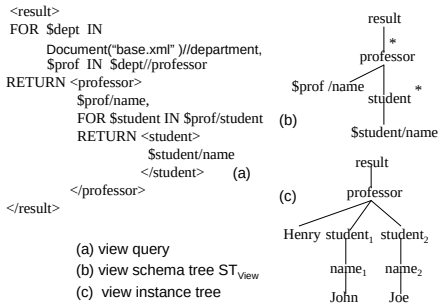


Figure 2.3: Query Q_4 and corresponding view

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 1.1 to delete $course_1$ in Figure 1.3(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 1.2(c). Therefore, such update will cause side-effects over view element $course_2$,as one of its sources get deleted. The second kind of side-effects is $source(ve_i)$ is also a source of another view element. For example, $course_b$ in Figure 1.1 is the source of $course_2$ in Figure 1.3(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, des($source(ve_i)$) $\cap$ $sources(ve_j)$ = $\emptyset$.

Chapter 3

Part III:

Solutions for Single View Element Update

3.1 Algorithm Analysis

3.1.1 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 $Henry$ and $student_1$ in Figure 2.3(c). Suppose we want to delete $student_1$. $Source(student_1)$ is $student$ in Figure 2.2. And $Des(Source(student_1))$ = {$student$, $name_{student}$}. On the other hand, $Sources(Henry)$ = {$department$,$professor$,$name_{professor}$}. So $Des(Source(student_1)) \cap Sources(Henry)$ = {$student$, $name_{student}$} $\cap$ {$department$, $professor$, $name_{professor}$} = $\emptyset$. This implies deleting the $source(student_1)$ will not affect any $source(Henry)$ and therefore $Henry$ will not be affected.

As $course$ in DTD in Figure 1.1 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. 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, such as $student_1$ and $student_2$ in Figure 2.3(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 1.1. 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, which will be illustrated respectively in the following sections.

Obviously, Lemma 1 forms an incomplete algorithm, as our analysis identified two kinds of view-elements that cannot be handled by the lemma. However, it provides a systematic way to study the problem. We will accordingly partition the view schema tree into three parts, as shown in Figure 3.1. 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-descendants of n. We 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 3.1.3. 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 3.1.4.

Note we have not considered invisible elements, since the lemma only acquires information from view schema tree, which is directly generated from XQuery statement. In the next section 3.3, how we are going to tackle invisible elements related side-effects will be presented in detail.

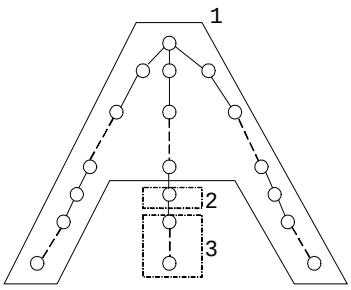


Figure 3.1: Schema Tree Structure

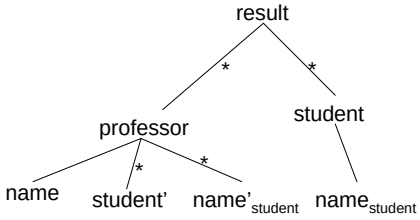


Figure 3.2: Schema tree of query in Figure 1.5 after appending schema nodes for invisible elements

After all the discussions in the following sections, our algorithm will cover all schema nodes without any overlap. Thus we can check all view elements for side-effects effectively, and a correct translation is returned if there exists one.

3.1.2 Making Invisible Elements Visible

As we have shown in Figure 1.5, there are some view elements that do not have corresponding view schema nodes. These view elements are called *invisible elements*. The rest of view elements are called *visible elements*. View elements become invisible when XQuery requires to return a base element be_i, which has other base elements as its descendants. As view schema tree is generated merely based on XQuery statement, only a view schema node SN_{View} corresponding to be_i will appear. However, in the view instance tree, all the descendants of be_i will appear. Thus these descendant elements become invisible in the view schema tree.

This makes Lemma 1 not useful to detect side-effects over invisible view elements. Because Lemma 1 traverses the whole view schema tree and examines $Sources$ of any view schema node are affected. Since invisible view elements do not have corresponding schema nodes in the view schema tree, Lemma 1 will not detect any side-effects over invisible view elements if there exists any.

Note as the user sees the view instance only, the view element required to be deleted could be

either visible or invisible. We will first assume the required view element is always visible. We will study how to detect side-effects over both visible and invisible elements in each group in Figure 3.1 in the following sections. Then we will study how to detect side-effects when the required view element is invisible in Section 3.3.

In order to make all the view elements visible in the view schema tree, we extend the view schema as follows. For every view schema node SN_{view}, let $SN_{visible}$ be the base schema node of the base element $be_{visible}$ that XQuery requires to return for SN_{view}. Get the descendants of $SN_{visible}$ and add them as children of SN_{view} in the view schema tree. Let us consider the query in Figure 1.5 again. Both $professor$ and $student$ in the original view schema tree have descendants that are invisible. After appending these descendants as children of $professor$ and $student$, the schema tree is shown in Figure 3.2. Now consider deleting a view element whose view schema node is $student$. Using Lemma 1, this will cause side effects, as there exists a schema node in Group 1, $student'$, that $Des(Source(student)) \cap Sources(student') \neq \emptyset$.

After extending the view schema tree as above stated, Lemma 1 can detect side-effects over both visible and invisible view elements.

3.1.3 Detecting Side-Effects in Group 2

In this section, we study how to detect side-effects over view elements in Group 2 when deleting a visible view element, ve_i. Note by definition all view elements in Group 2 share the same view schema node, which is visible. This is similar to the relational view update problem, and we can utilize the solutions from the relational scenario.

Updating Relational Views

In [14], 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 first 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 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. □

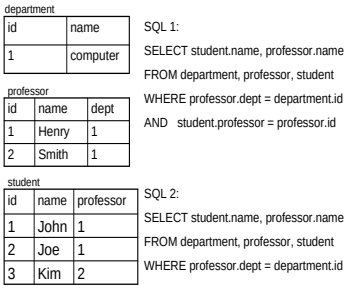


Figure 3.3: Queries over relational tables

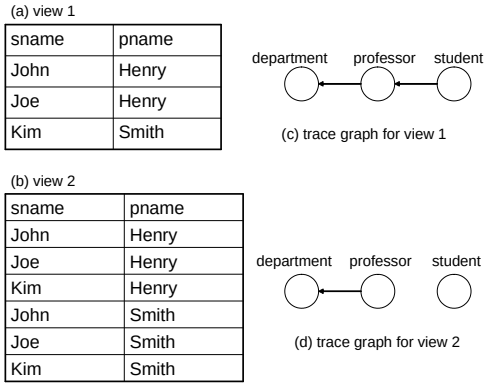


Figure 3.4: Views and their trace graphs

Let us consider two queries in Figure 3.3. According to Keller's algorithm, we have their trace graphs shown in Figure 3.4. In the trace graph of view 1, $student$ can reach all the nodes, which implies we can delete from $student$ to delete any single view element in view 1. On the other hand, there is no node that can reach all the nodes in the trace graph of view 2. Therefore, there is no correct translation of deleting any single view element in view 2.

In the above algorithm, an edge stands for a 1 to many join condition. Let us now examine what a path between two nodes in the graph signifies. This attribute is the base of our correctness proof. In the following proof, the name of the node is also the name of the relation this node represents.

Lemma 2 *For a certain node r_i in this directed graph, if there is a path from r_i to r_j, then each tuple in r_i will be joined with at most one tuple in r_j.*

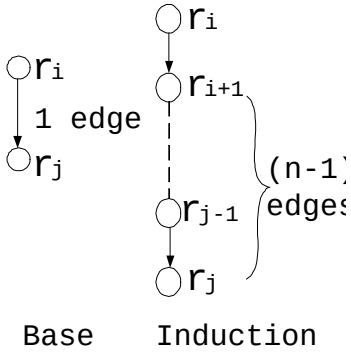


Figure 3.5: path from n_i to n_j

Proof 1 *This can be proved by induction on the length of path from r_i to r_j, as shown in Figure 3.5.*

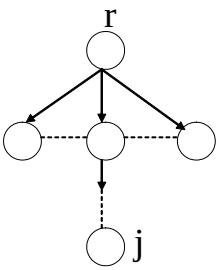
Base Step. *If the number of edges from r_i to r_j is 1, then tuples in r_i will join with at most one tuple in r_j.*

Induction Hypothesis. *Assume the claim holds when the number of edges from r_i to r_j is up to n-1.*

Induction Step. *We shall demonstrate the claim is true when the number of edges from r_i to r_j is n. Consider the path from r_{i+1} to r_j with (n-1) edges. A tuple in r_{i+1} will be joined with at most one tuple in r_j (from induction hypothesis). We also know that a tuple in r_i will join with at most one tuple in r_{i+1}. Therefore, a tuple in r_i will join with at most one tuple in r_j.* □

Proof 2 Correctness of Keller's Algorithm: *If the node which stands for relation r reaches all other nodes through edges, any tuple in relation r will join at most once with any tuple in any other relation, from Lemma 2. We can thus prove that tuples in the view have a 1-1 mapping with tuples in r by contradiction.*

For the sake of contradiction, let us assume it is possible to have two tuples in the view that map to the same tuple in the relation r, as shown in Figure 3.6. These two view tuples cannot map to the same tuple in all relations (from SQL query definition); suppose one of the relations where



t_n		t_{j1}	
t_n		t_{j2}	

(a) Relationship between nodes

(b) two tuples in view

Figure 3.6: view and query graph for contradiction

they map to different tuples is relation j and one of the view tuples maps to t_{j_1}, the other to t_{j_2}. On the other hand, node r can reach node j. But t_n should have joined with at most one tuple in relation j, which is a contradiction. So the assumption does not hold. So tuples in the view have a 1-1 mapping with tuples in the relation r.

Completeness of Keller's Algorithm: *Given a relational databases with Key constraints and an SPJ view over it, we can come up with a trace graph described in the algorithm. If there isn't any node r that can reach all other nodes, then any tuple t_r in r may join with more than one tuple in any other relation. This means t_r may contribute to more than one tuple in the view. Naturally, tuples in the view can not be guaranteed to have a 1-1 mapping with set of tuples in any relation.* □

With Keller's algorithm, we can find a set of base tuples to which the view elements have 1-1 mapping. This implies we can delete any view element by deleting its corresponding base element, which contribute to no other view elements.

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 2.3(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 $department$ [1]. Similarly, every $student$ is guaranteed to join with at most one $professor$. According to Keller's algorithm, we can draw the trace graph of $student$, shown in Figure 3.7. As $student$ can reach all the other nodes, we can delete view element $student_1$ by deleting base element $student_1$ in D, as analyzed before.

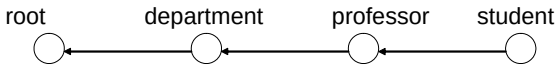


Figure 3.7: trace graph for $student$ in Figure 2.3(b)

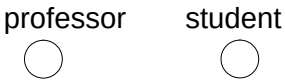


Figure 3.8: trace graph for $professor$-$student$ in Figure 1.4(a)

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 1.4(b). A view element of $prof$-$student$ has $Sources = \{professor, student\}$, without any edge between them in the trace graph, shown in Figure 3.8. However, as base schema in Figure 1.4(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 1.3(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 1.1. 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 1.3(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

[1] For now we assume all the XML elements are not recursive types. How recursive types cause side-effects will be discussed later in this section

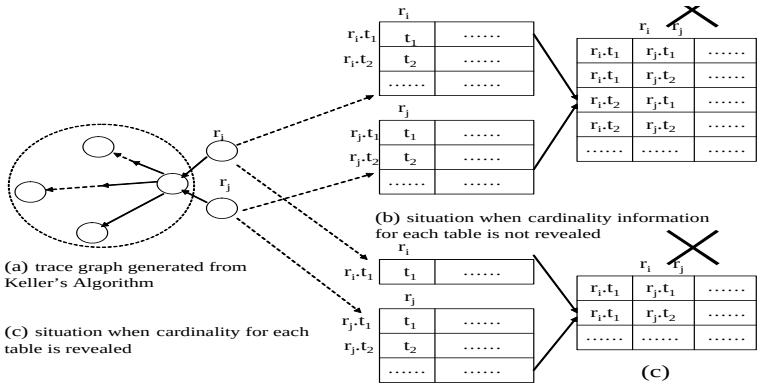


Figure 3.9: Keller's algorithm and cardinality constraints

constraints and recursive types in XML.

Handling Cardinality Constraints

How cardinality information impacts the translatability of view updates in relational scenario is illustrated in Figure 3.9, 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 3.9(b). However, if we know the cardinality information that there is only one tuple in r_i [2], then view tuples can be deleted from r_j, shown in Figure 3.9(c).

While such cardinality information cannot be specified easily in relational schema, it does exist in XML schema, as we mentioned in section 2.1.2. 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, 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 one of these axes: child, descendant, 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:*

[2] This is a quite strict requirement for an intuitive explanation, which will be relaxed in later discussions.

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 cardinality multiplication operation, please refer to Figure 3.10(a).*

2. *if n=1:*

   (a) *if* a_1 *is descendant,* $card(t, /a_1 :: t_1)$ = *.

   (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), $cardRE(t, re)$ = $cardRE(t_1, re_1)$+ $cardRE(t_1, re_2)$. *For the cardinality addition operation, please refer to Figure 3.10(b).*

      ii. *if* $re = (re_1 \mid re_2)$, $cardRE(t_1, re) = max\{ cardRE(t_1, re_1), cardRE(t_1, re_2)\}$. *For the cardinality max operation, please refer to Figure 3.10(c).*

      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.*

Consider Figure 2.2, cardinality between $root$ and $department$ can be computed as $card(root, /child :: institute/child :: department) = card(root, /child :: institute) \times card(institute, /child :: department) = * \times * = *$.

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,

x	1	0	*
1	1	0	*
0	0	0	0
*	*	0	*

(a) Multiplication

+	1	0	*
1	*	1	*
0	1	0	*
*	*	*	*

(b) Addition

max	1	0	*
1	1	1	*
0	1	0	*
*	*	*	*

(c) Max

Figure 3.10: Cardinality operations

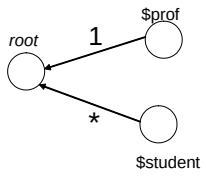


Figure 3.11: trace graph of $prof$-$student$ in Figure 1.4(b) with cardinalities

$card(SN_{ancestor}, r_i)$ 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 1.4 again. Figure 3.11 is the trace graph of $prof$-$student$ in Figure 1.4(b). With Definition 1, $card(root, professor) = 1$, $card(root, student)$ = *. Therefore, to delete the view element whose view schema node is $prof$-$student$, we can delete from Source $student$.

Handling Recursive Type

Recursive types may cause two kinds of side-effects as mentioned earlier. 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 1.2(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.*

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 3.13(a). Here both the be_l's satisfy P and have ancestor-descendant relationship. The $Source$, $student$, for a $student$ view element in Figure 2.3 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 1.2 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.

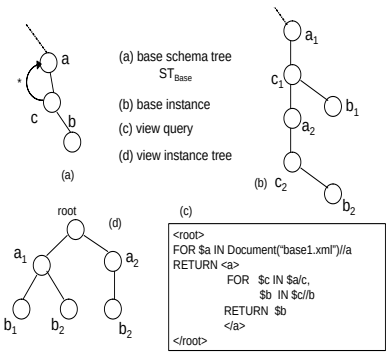


Figure 3.12: ST'_{Base}, Query Q_5 and corresponding view

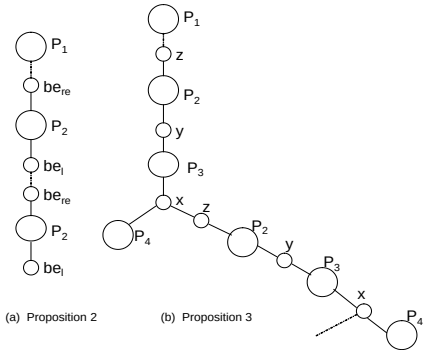


Figure 3.13: Illustrating Proposition 2 and Proposition 3

Now let us consider the second type of side-effects, where $source(ve_i)$ is also $source(ve_j)$. Consider the query in Figure 1.3(a). $course_c$ in Figure 1.1 contributes to two view elements, $course_2$ and $course_3$, in Figure 1.3(c). A more general example is shown in Figure 3.12. Figure 3.12(a) is the base schema and Figure 3.12(b) is one possible instance. Based on the query in Figure 3.12(c), we have the view instance tree shown in Figure 3.12(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, if 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 3.13(b) illustrates Proposition 3. Here, there are two view elements where $Source_2$ binds to the rightmost P_4, and $Source_1$ binds to the two different y's.

Actually this scenario implies a much stronger condition: there exists no correct translation for deleting view element ve_i that has such $Source$s. Let us examine this. Let $Source_i$,

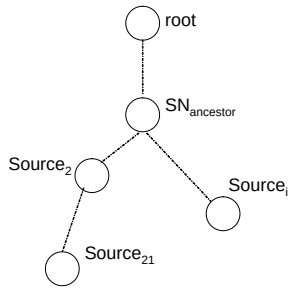


Figure 3.14: trace graph that qualifies Proposition 3

a $Source(ve_i)$, to be candidate we consider to delete ve_i. Since ve_i has $Source$s described in Proposition 3, $Source_i$ can either reach $Source_2$ or not. If $Source_i$ can reach $Source_2$, an instance of $Source_i$ can be the source of two different view elements. Therefore $Source_i$ cannot be a correct translation. Now let us consider the case in which $Source_i$ cannot reach $Source_2$, shown in Figure 3.14. Here we must consider cardinality constraints. Let $Source_{21}$ be a 0-indegree that reach $Source_2$ [3]. As the lowest common ancestor of all 0-indegree nodes, $SN_{ancestor}$, must be a node in the path from root to $Source_{21}$, $card(SN_{ancestor}, Source_{21})$ = *. According to Proposition 1, if $card(SN_{ancestor}, Source_i)$ = *, $Source_i$ cannot be a correct translation. If $card(SN_{ancestor}, Source_i)$ = 1, then the only possible correct translation is $Source_{21}$. However, as we discussed, $Source_{21}$ cannot be a correct translation. Therefore, $Source_i$ cannot be a correct translation if it cannot reach $Source_2$. 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 3.2 for how to integrate these propositions into our algorithm.

3.1.4 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, where n is the view schema node of a visible view element ve_i. Note view elements that are descendants of ve_i will get deleted with ve_i, according to the hierarchial structure

[3] Note S_{21} can be S_2

of XML view. Therefore, we focus on whether any view element, ve_j, that belongs to descendants of siblings of ve_i, gets affected when deleting $source(ve_i)$. Note ve_j could be either visible or invisible.

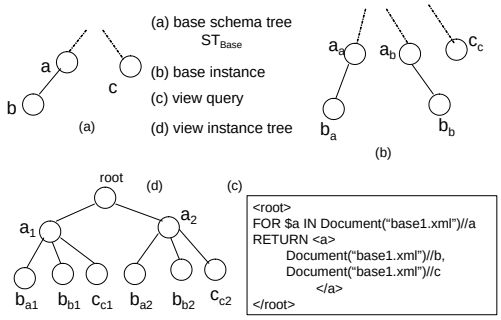


Figure 3.15: ST'_{Base}, $Query\ Q_6$

Figure 3.15 illustrates side-effects on Group 3. If we delete a_1 in Figure 3.15(d) by deleting a_a in Figure 3.15(b), then the view element b_{a2}, the descendant of a_2 in Figure 3.15(d) is deleted. This is a side-effect. On the other hand, there is no side-effects on view element c_{c2}.

Let us analyze why view element c_{c2} does not get affected. $Sources(a_1) = \{a\}$ and $Sources(c_{c2}) = \{a, c\}$. Intuitively, deleting a_1 by deleting the a element may cause side-effects over c_{c2}, because a is also a $Source(c_{c2})$. However, this is not true as c_{c2} is a descendant of a_2, which is a sibling of a_1. a_1 and a_2 share the same view schema and $Sources$, but they have different $sources$: $sources(a_1) = \{a_a\}$, $sources(a_2) = \{a_b\}$. As c_{c2} is the descendant of a_2, it also has a_b as one of its source, which is different from the $sources$ of a_1. Therefore, deleting from a will not change the part of $sources$ of c_{c2} which comes from a. In addition, a and its descendants in the base tree do not have ancestor-descendant relationship with c, which ensures deleting from a will not change the part of $sources$ of c_{c2} which comes from c. So, deleting from a will not change any $source$ of c_{c2}, therefore it is not affected.

Again, let us analyze why view element b_{a2} gets affected. $Sources(a_1) = \{a\}$ and $Sources(b_{a2}) = \{a, b\}$. Same with c_{c2}, deleting from a does not change the part of $sources$ of b_{a2} which comes from a. However deleting from a changes the part of $sources$ of b_{a2} which comes from b: deleting a_a in the base instance tree will also delete b_a, which is a $source(b_{a2})$. b is a descendant of a in the base schema tree, which means every base element of b will have an ancestor whose view schema

node is a. However in the view definition there is no condition to restrict that each base element of b can only join with one base element of a (e.g. b's ancestor). In the trace graph of b_{a2}, this is shown as b cannot reach a. Therefore the base element b_a also joins with a_b, and together become $sources(b_{a2})$. The above discussions can be generalized as the following lemma:

Lemma 3 *For every descendant schema node SN_d of $SN_{View}(ve_i)$, get its trace graph. Let ve_d be a view element such that $SN_{View}(ve_d)$ = SN_d. If $\exists\ r_i \in Sources(ve_d)$ such that $r_i \in Des((Source(ve_i)))$ and there is no path in the trace graph from r_i to $Source(ve_i))$, $Source\ (ve_i)$ cannot be the correct translation of deleting ve_i.*

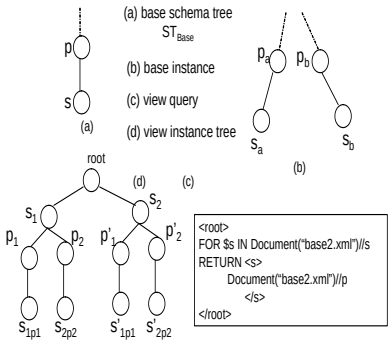


Figure 3.16: Q_7 and corresponding view

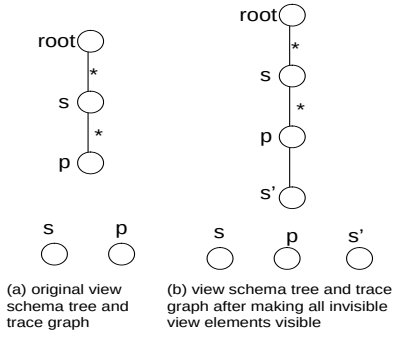


Figure 3.17: view schema trees and trace graphs for query in Figure 3.16(c) before and after making invisible view elements visible

Note the above lemma can also detect side-effects over invisible view elements in Group 3, assuming we have made all invisible view elements visible. For example, we want to delete s_1 in Figure 3.16(d). s_1 comes from a base element whose base schema node is s in Figure 3.16(a). Now we will try to use Lemma 3 to detect if deleting from s in Figure 3.16(a) will cause any side-effects over view elements in Group 3. According to the trace graph Figure 3.17(a) generated from the original view schema tree, we cannot identify any problem at all. After making all the invisible view elements visible, just like in Figure 3.17(b), we can observe that we add a new node in the trace graph, s'. $s' \in Sources(s'_{1p1})$ and $s' \in Des(Source(s_1))$. However there is no path in the trace graph from s' to s. Therefore deleting from s in Figure 3.16(a) will cause side-effects over invisible view elements in Group 3 and hence cannot be a correct translation for deleting s_1 in Figure 3.17(d).

3.2 Algorithm for Correctly Deleting Single Visible View Element in XML Scenario

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

3.2.1 Optimizations

As we discussed in the last section, in order to find a correct translation of deleting a visible view element, we partitioned the schema tree into three groups and proposed different proved lemma and propositions to detect the side-effects in each group respectively. As these three groups covers the whole schema tree, we can always correctly find a correct translation if exists. In this section, we will propose two kinds of optimizations to make the algorithm effective, which are all based on the following observations.

Observation 1 *Let ve_i be a descendant of ve_j in the view, $Sources(ve_j) \subseteq Sources(ve_i)$.*

In Lemma 1, in order to check if there exists side-effects in Group 1 when deleting ve_i from $Source(ve_i)$, we need to check if any $Source$ is affected for every view schema node in this Group. In fact we only need to check those view schema nodes that are leaves in Group 1. Given any non-leaf view element ve_j, it always has at least one leaf descendant, say ve_k. If $Source_(ve_j)$ is affected, as $Source_(ve_i) \in Sources(ve_k)$ according to Observation 1, ve_k will also get affected. Therefore only checking the leaf view elements suffice to detect side-effects in Group 1. This will greatly decrease the view elements we will check. For example, consider deleting $name_1$ in Figure 2.3(c). The view schema nodes in Group 1 are: { $professor$, $student$, $\$prof/name$ }. However we only need to check if there exists any side-effect over $\$prof/name$.

The second kind of optimizations enables us to decrease the candidates we consider to delete ve_i without affecting other view elements. Let ve_i be the view element we want to delete and ve_j the parent of ve_i. From Observation 1, we know that $Sources(ve_j) \subseteq Sources(ve_i)$. This

means if we try to delete ve_i from any $Source \in Sources(ve_j)$, ve_j will also get affected. Therefore, only $Source \in Sources(ve_i)_{prune}$ need to be considered as candidate of deleting ve_i, where $Sources(ve_i)_{prune}$ = $Sources(ve_i)$ - $Sources(ve_j)$. For example, consider deleting $student_1$ in Figure 2.3(c). We have $Sources(professor)$ = { $department$, $professor$ } and $Sources(student)$ = { $department$, $professor$, $student$ }. So $Sources(student)_{prune}$ = { $student$ }. This implies we only need to consider deleting from base schema node $student$ as the candidate of deleting view element $student_1$.

3.2.2 Algorithm

In this section, we will demonstrate the optimized algorithm Algorithm 3.2.2 that detects the correct translation of deleting a visible view element ve_i.

Algorithm 1 Algorithm 3.2.2 that correctly translating the deletion of a single visible view element

Step 0:

0. Append all the invisible view schema nodes as children of their nearest visible ancestor.
1. $Candidates$ = $Sources(ve_i)_{prune}$

Step 1:

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

Step 2:

5. Draw the trace graph of ve_i. Let $Sources_{Keller}$ be the set of n 0-indegree nodes in the trace graph.
6. 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.
7. Use Proposition 2 to check if $Source(ve_i)$ is recursive. If so $Candidates = Candidates - Source(ve_i)$.
8. 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.
9. $Candidates = Candidates \cap Sources_{Keller}$. If $Candidates = \emptyset$, the algorithm terminates; otherwise go to Step 3.

Step 3:

10. For every $Source \in Candidates$, if deleting $Source$ has side-effects on a descendant according to Lemma 3, $Candidates = Candidates - Source$.
11. The algorithm terminates. If $Candidates = \emptyset$, there is no correct translation of deleting ve_i; otherwise each $Source \in Candidates$ is a correct translation.

3.3 Detecting side-effects when deleting an invisible element

In the last two sections, we discussed how to detect side-effects of deleting a visible element. However, the problem of detecting the side-effects of deleting an invisible view element remains unsolved. Ideally, we want to use Algorithm 3.2.2 developed in previous sections. In this chapter, we will first introduce some concepts and explain why Algorithm 3.2.2 cannot be directly applied here, then we will give two propositions to efficiently check the side-effects when deleting an invisible view element. At last, we will propose an algorithm for efficiently checking side-effects when deleting an invisible element.

3.3.1 Compositions of $Sources(ve_{invisible})$ and $DataSource(ve_{invisible})$

In Section 3.1.2, we discussed how to make invisible view elements visible in the view schema tree. Let $ve_{invisible}$ be an invisible view element and $ve_{visible}$ its nearest visible view element. Their view schema nodes are $SN_{invisible}$ and $SN_{visible}$ respectively. The return statement of $SN_{visible}$ requires to display values of base elements of SN_{base}, shown in Figure 3.18. In order to make invisible descendants of SN_{base} visible, we append them as children of $SN_{visible}$. For example, SN_b in Figure 3.18(a) is appended as SN_{viewb} of $SN_{visible}$ in Figure 3.18(b).

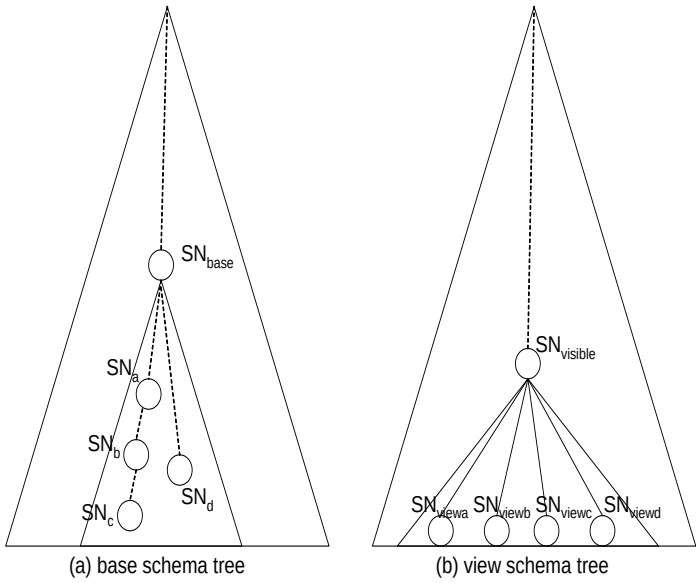


Figure 3.18: appending invisible view schema nodes

Let us consider what would happen if we apply Algorithm 3.2.2 directly upon invisible view elements. First we need to check side-effects in Group 1 of $SN_{invisible}$. Consider deleting an invisible view element $student_a$ in Figure 3.19(c). Obviously, deleting its $source$, the first $student$ element in Figure 1.1, is a correct translation. However, applying Lemma 1 over the view schema tree, shown as Figure 3.19(b), will show that this causes side-effects, because $Des(Source(student))$ $\cap\, Sources(name_{student}) \neq \emptyset$.

This happens because $name_{student}$ is appended directly under $professor$ as child and appears in Group 1 of $student$, whereas its instances are actually the descendants of view elements of $student$. Generally, this means there exist some invisible view elements that are descendants of instances of $SN_{invisible}$ but their view schema nodes are children of $SN_{visible}$ and appears in Group 1 in view schema tree.

Now we will introduce two new concepts. By definition SN_{base} is a $Source$ of $SN_{visible}$. As the values of view elements of $SN_{visible}$ come from base elements of SN_{base}, we call SN_{base} as the *DataSource* of $SN_{visible}$, denoted as $DataSource(SN_{visible})$. Similarly, $DataSource(SN_{invisible})$ is $SN_{invisibleBase}$, where $SN_{invisibleBase}$ is the corresponding base schema node of $SN_{invisible}$. For example, $DataSource(ve_b)$ is SN_b in Figure 3.18(a), where ve_b is a view element of SN_{viewb}. Correspondingly, the base element of $DataSource(ve)$ that is $source(ve)$ is denoted as $datasource(ve)$.

Let us consider how to compute $Sources(ve_{invisible})$. For example, what consists of $Sources(ve_b)$ in Figure 3.18(b), where ve_b is a view element of SN_{viewb}? According to Observation 1, $Sources(ve_{visible})$ $\subseteq Sources(ve_b)$. Also SN_b and any base schema node SN_i that is an ancestor of SN_b and descendant of SN_{base} is also $Source(ve_b)$. Because deleting a base element of SN_i will delete its descendants, including one whose base schema node is SN_b. For example, let $datasource(ve_a)$ of SN_a be the ancestor of $datasource(ve_b)$. Note SN_a is the descendant of SN_{base} and ancestor of SN_b. If we delete $datasource(ve_a)$ to delete ve_a, this will delete $datasource(ve_b)$, which will in turn delete ve_b.

In summary, $Sources(ve_{invisible})$ consists of three parts: $Sources(ve_{visible})$, $DataSource(ve_{invisible})$, and all base schema nodes that are ancestors of $DataSource(ve_{invisible})$ and also descendants of

SN_{base}.

3.3.2 Propositions that check side-effects when deleting invisible view elements

We only use it for checking side-effects over non-descendant elements of $SN_{visible}$, which is the blank area in Figure 3.20 and $SN_{visible}$ itself. However, among the children of $SN_{visible}$, which is the grey area in Figure 3.20, there may still exist some view schema nodes whose instances are also non-descendants of $SN_{invisible}$. In order to check the side-effects over these nodes, we need to examine the invisible view schema nodes more carefully.

Let us consider what are the candidates of correctly deleting $ve_{invisible}$. Using the optimizations in Section 3.2, we do not consider $Sources(ve_{visible})$ as candidates because deleting from them will have side-effects over view elements of $SN_{visible}$. Also, we do no consider any base schema node that is the ancestor of $DataSource(ve_{invisible})$ and descendant of $DataSource(ve_{visible})$. Because each of them is a $DataSource$ of an invisible view schema node, appearing as child of $SN_{visible}$ in the view schema tree. For example, SN_a in Figure 3.18 is the ancestor of SN_b and descendant of $SN_{visible}$. Suppose there is a view element ve_a of SN_a that has descendant ve_b of SN_b. If we delete from SN_a to delete ve_b, view element ve_a will get affected. Therefore, the only candidate we consider is $DataSource(ve_{invisible})$. Now we give the following proposition to check side-effects over view schema nodes in grey area of Figure 3.18 whose instances are non-descendants of $SN_{invisible}$.

Proposition 4 *If deleting $ve_{invisible}$ from a $DataSource(ve_{invisible})$ has no side-effect over $SN_{visible}$, then there is also no side-effect over children of $SN_{visible}$ whose instances are non-descendants of view elements of $SN_{invisible}$.*

proof: Let ve_p be a non-descendant view element of $ve_{invisible}$. For the sake of contradiction, let us assume deleting from $DataSource(ve_{invisible})$ has side-effects over ve_p. This implies $DataSource(ve_{invisible}) \in Sources(ve_p)$.

As we discussed above, $Source$s of an invisible view element ve_p consists of three parts: $Sources(ve_{visible})$, $DataSource(ve_p)$ and base schema nodes that are ancestors of $DataSource(ve_p)$ and descendants of $SN_{visible}$. However, as deleting from $DataSource(ve_{invisible})$ has no side-effects over view elements of $SN_{visible}$, $DataSource(ve_p) \notin Sources(ve_{visible})$. According to definition, view elements of SN_p are non-descendants of $ve_{invisible}$. Therefore, $DataSource(ve_p)$ is non-descendant of $DataSource(ve_{invisible})$. According to the hierarchial structure of XML Document, $DataSource(ve_{invisible})$ cannot be $DataSource(ve_p)$ or its ancestors. So $DataSource(ve_{invisible}) \notin Sources(ve_p)$, which contradicts the assumption.

Therefore deleting from $DataSource(ve_{invisible})$ has no side-effects over ve_p, Hence proved. □

For example, if deleting from SN_b in Figure 3.18 to delete ve_b has no side-effects over $SN_{visible}$, then there is no side-effects over SN_{viewa} and SN_{viewd}.

Now let us consider how to check side-effects over other view schema nodes in the grey area of Figure 3.20. These view schema nodes are of two kinds: nodes whose $DataSource$s are descendants of $DataSource(ve_{invisible})$, called as ***invisible descendants*** of $SN_{invisible}$ and $SN_{invisible}$ itself. For example, we need to check side-effects over SN_{viewb} and SN_{viewc} in Figure 3.18. We will first give the following proposition to check side-effects over $SN_{invisible}$:

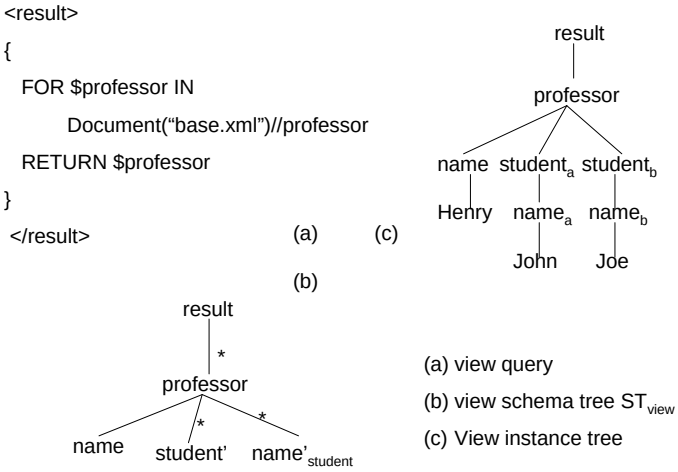


Figure 3.19: Applying Previous algorithm over deleting invisible view elements

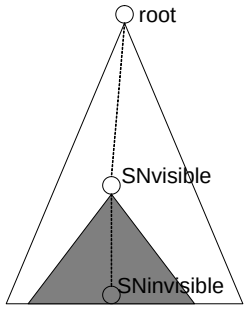


Figure 3.20: Schema Tree Partition for an invisible view schema node

Proposition 5 *Consider $ve_{invisible}$ and its nearest visible ancestor $ve_{visible}$. Let their view schema nodes be $SN_{invisible}$ and $SN_{visible}$ respectively. Deleting from $DataSource(ve_{invisible})$ will have side-effects over $SN_{invisible}$ if and only if deleting from $DataSource(ve_{visible})$ has side-effects over $SN_{visible}$.*

Note this does not mean $DataSource(ve_{visible})$ is required to be a correct translation of deleting $ve_{visible}$.

proof:

First we will show if deleting from $DataSource(ve_{visible})$ has side-effects over $SN_{visible}$, deleting from $DataSource(ve_{invisible})$ will have side-effects over $SN_{invisible}$. As there are two kinds of side-effects introduced in Section 2.2, we will discuss each kind separately.

Let us consider the case when the first kind of side-effect occurs. This implies there exist a view element $ve'_{visible}$, whose view schema node is also $SN_{visible}$. And $datasource(ve'_{visible})$ is the descendant of $datasource(ve_{visible})$. So when we delete $datasource(ve_{visible})$ to delete $ve_{visible}$, $datasource(ve'_{visible})$ will also get deleted. This in turn causes side-effects over $ve'_{visible}$. So base elements of SN_{base} have ancestor-descendant relationship. Therefore SN_{base} is a recursive node by definition.

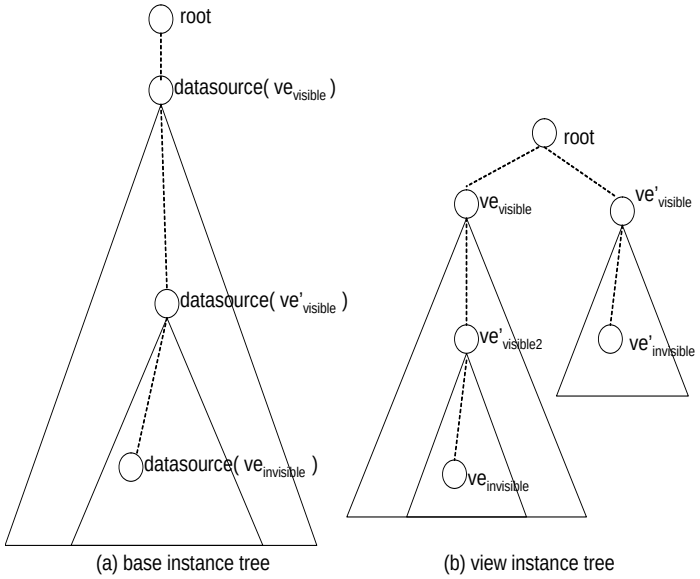


Figure 3.21: example that illustrates the case when the first condition holds

A possible view instance can be as displayed as Figure 3.21. $ve_{visible}$, and $ve'_{visible}$ share the same $DataSource$. And $datasource(ve_{visible})$ is the ancestor of $datasource(ve'_{visible})$ shown in Figure 3.21(a). Also, as the return statement of $SN_{visible}$ requires to display the contents of $ve_{visible}$, there must be a descendant of $ve_{visible}$, $ve'_{visible2}$, whose $datasource$ is $datasource(ve'_{visible})$. $ve_{invisible}$ may be a descendant of $ve'_{visible}$. This implies there exists another descendant of $ve'_{visible}$, $ve'_{invisible}$, that share the same $datasource$ with $ve_{invisible}$. Therefore, if we want to delete $datasource$ $(ve_{invisible})$ to delete $ve_{invisible}$, this will in turn deletes $ve'_{invisible}$, which causes side-effects.

Now let us consider the case when the second kind of side-effects occurs. This condition implies that there is another view element $ve'_{visible}$ that shares the same view schema node and the same $datasource$ with $ve_{visible}$. For $SN_{invisible}$, there will be two view elements, $ve_{invisible}$ and $ve'_{invisible}$, that are descendants of $ve_{visible}$ and $ve'_{visible}$ respectively and they share the same $datasource$. So deleting $datasource(ve_{invisible})$ to delete $ve_{invisible}$ will have side-effects over $ve'_{invisible}$.

Second, we will show if deleting from $DataSource(ve_{invisible})$ has side-effects over $SN_{invisible}$, then deleting from $DataSource(ve_{visible})$ will have side-effects over $SN_{visible}$. Let $ve_{invisible}$ and $ve'_{invisible}$ be descendants of $ve_{visible}$ and $ve'_{visible}$ respectively. If deleting $ve_{invisible}$ has side-effects over $ve'_{invisible}$, then $datasource(ve_{invisible})$ is either the ancestor of $datasource(ve'_{invisible})$ or $datasource(ve'_{invisible})$ itself. As $datasource(ve_{visible})$ is the ancestor of $datasource(ve_{invisible})$ and $datasource(ve'_{visible})$ is the ancestor of $datasource(ve'_{invisible})$, the relationship between $datasource$ $(ve_{visible})$ and $datasource$ $(ve'_{visible})$ will fail into one of the following three cases:

1. $datasource(ve_{visible})$ is $datasource(ve'_{visible})$. Then deleting $ve_{visible}$ or $ve'_{visible}$ will have side-effects over the other.

2. $datasource(ve_{visible})$ is an ancestor of $datasource(ve'_{visible})$. Then deleting $ve_{visible}$ will have side-effects over $ve'_{visible}$.

3. $datasource(ve_{visible})$ is an descendant $datasource(ve'_{visible})$. Then deleting $ve'_{visible}$ will have side-effects over $ve_{visible}$.

As we can see, no matter in which case, deleting from $DataSource(ve_{visible})$ will have side-effects over $SN_{visible}$.

In one word, the assumption leads to contradictions in all cases. So the assumption is not true. Hence proved. □

For example, consider deleting an invisible view element $ve_{student}$ whose view schema is $student$ in Figure 3.19(c). The nearest visible view schema node is $professor$. The $DataSource$ $(professor)$ is $professor$ in the base. Since there is no side-effect of deleting a view element of $professor$ in the view by deleting from $professor$ in the base, there is no side-effects over $student$ in Figure 3.19(c) if we delete from $student$ in the base to delete $ve_{student}$.

Lastly, we need to check the side-effects over invisible descendants of $SN_{invisible}$ when deleting from $DataSource(ve_{invisible})$ to delete $ve_{invisible}$. Let us review the example why b_{a2} in Figure 3.15 gets affected in Section 3.1.4. The reason is in the view definition there is no condition to restrict that every base element b of can only join with one base element of a (e.g. its ancestor).

However, this will not happen when the view element $ve_{invisible}$ is invisible. Because its descendants will only display the values of descendants of $datasource(ve_{invisible})$. In other words, $be_{descendant}$ will never join with any base element of $DataSource(ve_{invisible})$ except its own ancestor, where $be_{descendant}$ is a descendant of $datasource(ve_{invisible})$. Thus if deleting from $DataSource$ $(ve_{invisible})$ over invisible descendants of $SN_{invisible}$, it must because that $datasource(ve_{invisible})$ is $datasource(ve'_{invisible})$, where $ve'_{invisible}$ is another view element of $SN_{invisible}$. Therefore we have the following proposition:

Proposition 6 *deleting from $DataSource(ve_{invisible})$ causes side-effects over invisible descendants of $SN_{invisible}$ if and only it causes side-effects over $SN_{invisible}$.*

3.3.3 Algorithm for Correctly Deleting Single Invisible View Element in XML Scenario

With the above discussions, we propose the algorithm for finding correct translation of deleting an invisible view element $ve_{invisible}$, whose nearest visible ancestor is $ve_{visible}$.

Algorithm 2 Algorithm 3.3.3 that correctly translating the deletion of a single invisible view element

---

Step 0:

0. Append all the invisible view schema nodes as children of their nearest visible ancestor.

Step 1:

1. Let $Sources'$ be the union of $Source(ve_{visible})$ and $Sources$ of all non-descendant leaf view elements of $SN(ve_{visible})$.
2. If $Des(DataSource(ve_{invisible})) \cap Sources' \neq \emptyset$, the algorithm terminates. There is no correct translation. Else go to Step 2.

Step 2:

3. Use Step 2 in Algorithm 3.2.2, where $ve_i = ve_{visible}$ and $Candidates = DataSource(ve_{visible})$.
4. The algorithm terminates. If $Candidates \neq \emptyset$, then $DataSource(ve_{invisible})$ is a correct translation; otherwise there is no correct translation of deleting $ve_{invisible}$.

Chapter 4

Part IV:

Solutions for A Set of View Elements Update

4.1 Algorithm Analysis

As we discussed in Section 2.1, we will use XPath XP to specify the view elements that need to get deleted. However, XP can be quite complex. We will set the following constraints to XPath XP and try to relax them in future work: [1]

1. XP cannot have any axis except child axis (no "//" axis, for example).
2. XP does not have wildcards.
3. For every $element$ in XP, it can have predicates. Each predicate is of the form: $path$ = v, where v is a value of primitive type and $path$ is an XPath that starts from a child of $element$, having child axis only.

With the above constraints, the set of view elements that we can specify with XP always corresponds to view elements with the same view schema node. Therefore, we will again partition

[1] Note these restrictions do not apply to XPath expressions in view definition.

the view schema tree into three groups as in Figure 3.1 and check if there exists side-effects over any group for each candidate of translation.

4.1.1 Differences between Deleting a Set of View Elements and a Single View Element

If elements in XP have predicates such that only one view element is specified, then we will be dealing with how to correctly translate the deletion of one view element, which has been tackled in the previous chapter. Let us consider Figure 1.2 in Section 1.1. Suppose the user only wants to delete the view element that can be specified by the $XPath$: $result\ /course[1]$, e.g. $course_1$ in Figure 1.2(c). One way is to delete $course_a$ in Figure 1.1 but this will cause $course_2$ and $course_3$ in Figure 1.2(c) to get deleted, which are side-effects. In fact, there is no way to delete $course_1$ without any side-effect. Therefore deleting $course_1$ is untranslatable. From this, we can observe that deleting a single view element is a special case of deleting a set of view elements.

As shown in an example in Section 1.1, deleting a set of view elements may be translatable even when deleting a single view element in this set is untranslatable. For example, if the user wants to delete $result\ /course$ in Figure 1.2 in Section 1.1, e.g. $course_1$, $course_2$ and $course_3$ together, then deleting $course_a$ in Figure 1.1 is a correct translation, as the original side-effects over $course_2$ and $course_3$ become expected updates now. The side-effects of $course_1$ are *cancelled*.

From the above examples, we observe that the reason why deleting a set of view elements Set_{delete} may be translatable while deleting one of them, ve_{delete}, is untranslatable is that unexpected updates from deleting ve_{delete} over view, e.g. side-effects, become required from deleting other view elements in Set_{delete}. From this, we can directly get the following proposition:

Proposition 7 *Given a set of view elements Set_{delete} required to get deleted and SN_{delete}, which is their view schema node. Consider deleting only ve_{delete}, any of such view elements. If deleting from a $Source(ve_{delete})$ causes side-effects in Group 1 in Figure 3.1, deleting from $Source(ve_{delete})$ to delete Set_{delete} will also cause side-effects in Group 1.*

This is because by definition, view elements in Group 1 are non-descendant elements of all view elements of SN_{delete}, which includes Set_{delete}. Therefore any updates in Group 1 are always unexpected when deleting Set_{delete}.

Also, from the above discussion, we notice that when there is no predicate in $XPath$, we need to delete all the view elements of a view schema node and their descendants. Therefore we do not need worry side-effects over Group 2 and Group 3. So we have Proposition 8.

Proposition 8 *If there is no predicates in XP that specifies the view elements we want to delete, we do not need to check side-effects over Group 2 and Group 3.*

If there exists predicates in XPath, this means only a subset of view elements in Group 2 will get deleted. So we need to consider not only all the XML features, but also how having predicates makes impact over Algorithm 3.2.2 and Algorithm 3.3.3 in terms of checking side-effects over Group 2, which will be discussed in section 4.1.2. Note as having predicates is the only new feature here, we will first discuss how to handle predicates on the schema level without considering other XML features such as recursive types and cardinality constraints. Algorithms for these two features are the same as those for correctly translating the deletion of a single view element.

For Group 3, deleting only a part of view elements of the same schema node still need to use Proposition 3. Otherwise for a view element $ve_{descendant}$ of $SN_{descendant}$, a view schema node that is descendant of SN_{delete}, base elements whose base schema node belong to SN_{new} may join with more than one element of the candidate $Source$, where SN_{new} are the set of base schema nodes that exist in the trace graph of $ve_{descendant}$ but not in the trace graph of ve_{delete}. This in turn causes side-effects over descendants of ve_j, where ve_j is in Group 2 that needs to remain in the view.

4.1.2 Discussions on Predicates in XPath

In relational scenario, we have predicates in the form $t = v$, where t is an attribute of a table T and v is a value of a primitive type. Each such predicate specifies a set of tuples in T.

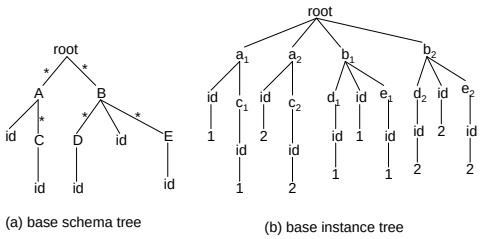


Figure 4.1: Base Instance

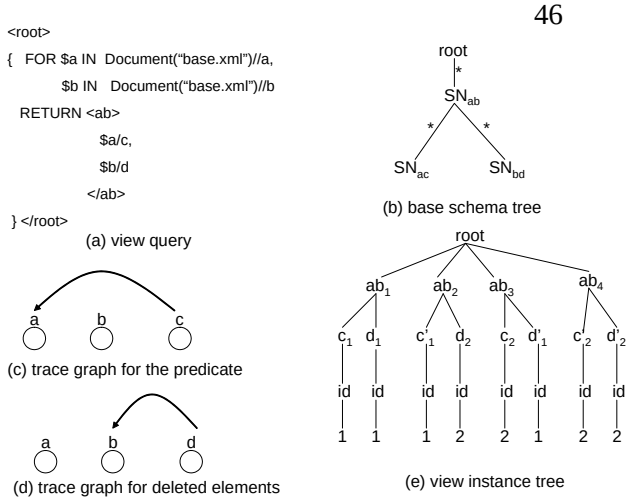


Figure 4.2: example for deleting a set of view elements specified by XP

In XML scenario, a predicate is associated with an element SN_i in XP. SN_i is either the ancestor of SN_{delete} or SN_{delete} itself, where SN_{delete} is the view schema node of the view elements the user wants to delete. Therefore each predicate associated with SN_i specifies a subset of view elements whose view schema node is SN_i. Take Figure 4.2 as an example. Suppose the base schema tree and base instance tree is as in Figure 4.1. Let XP be $root/SN_{ab}[SN_{ac}/id = 1]/SN_{bd}$. If there is no predicate, it means we want to delete all view elements of SN_{bd} in Figure 4.2(b) that are descendants of ab_1, ab_2, ab_3 and ab_4 in Figure 4.2(e). However, as the predicate specifies that view elements SN_{ab} need to have descendants of SN_{ac} whose id is 1, the user only wants to delete view elements of SN_{bd} that are descendants of ab_1 and ab_2, which is a subset of view elements of SN_{ab}.

Now let us consider if we can find a correct translation of deleting the elements specified by the above XP: $root/SN_{ab}[SN_{ac}/id = 1]/SN_{bd}$. Let one such element be ve_{bd}. $Sources(ve_{bd})$ = {A,B,D}. A and B are also $Source$s of view elements of SN_{ab}, so we can only consider D. However if we delete from base schema D, view elements of view schema SN_{bd} that are descendants of ab_3 and ab_4 will also get affected. Therefore there is no correct translation.

Then how to detect this on the schema level? Let us first examine predicates more carefully. As XP specifies restrictions over view elements of SN_{ac}, we draw the trace graph of SN_{ac} as in Figure 4.2(c). As we can see, the base schema node C can reach A, this implies every base

element of C will join with only one base element of A, which is its ancestor. More specifically, c_1 in Figure 4.1(b) will join with its ancestor a_1 only. Therefore we want to delete those view elements of SN_{bd} whose ancestors of SN_{ab} have a_1 as their $source$. According to Observation 1, we want to delete those view elements of SN_{bd} that has a_1 as their $source$. Then let us take a look at the trace graph of view schema SN_{bd} shown in Figure 4.2(d), which is the view schema node of view elements we want to delete. As the base schema D cannot reach A, this implies that every base element of D may join more than one element of A. Specifically, d_1 in Figure 4.1(b) joins with both a_1 and a_2.

Now it is clear how side-effects happens. We want to delete view elements of SN_{bd} that has a_1 as $source$. And we can only delete from D. However, as there is no join condition between A and D, every element of D joins with more than one element of A. Hence there will always occur second kind of side-effects. To prevent this, we need to make sure that every base element of D joins with at most one base element of A. This can be ensured if D can reach A in the trace graph of SN_{bd}.

To generalize the above observation, we can have the following proposition:

Proposition 9 *In XP, for every predicate $SN_1/SN_2/.../SN_n$ = v, get the trace graph of SN_n. Suppose there are a set of $Sources$, $Sources_{set}$, that can be reached by $DataSource(ve_n)$, where ve_n is a view element of SN_n. Let $Sources_{reach}$ = $Sources_{set} \cap Sources(ve_{delete})$, where ve_{delete} is one of the view elements to delete. Draw the trace graph of ve_{delete}, if a $Source(ve_{delete})$ can reach all the $Source \in Sources_{reach}$, then deleting from $Source(ve_{delete})$ will not cause second kind of side-effects.*

Note even though a $Source \in Sources_{reach}$ cannot be reached by the candidate $Source(ve_n)$, we can still check its relative cardinality using Proposition 1, which is the same to correctly translating the deletion of a single view element.

4.2 Algorithm for Correctly Deleting A Set of View Elements specified by XPath in XML Scenario

Similar to the case in deleting a single view element, we will give two algorithms respectively for deleting a set of visible view elements and invisible view elements.

4.2.1 Algorithm for Correctly Deleting A Set of Visible View Elements specified by XPath in XML Scenario

Let ve_i be one of the view elements to delete. With our above analysis and propositions, we have the following algorithm:

Algorithm 3 Algorithm 4.2.1 that correctly translating the deletion of a set of visible view elements

Step 0:

0. do Step 0 in Algorithm 3.2.2

Step 1:

1. do Step 1 in Algorithm 3.2.2.

Step 2:

2. Let $Sources_{all}$ be the union of $Sources_{set}$s of all predicates, each $Sources_{set}$ computed as in Proposition 9.
3. Draw the trace graph of ve_i. Let $Sources_{Keller}$ be the set of n 0-indegree nodes in the trace graph.
4. Let $Sources_{reach}$ = $Sources_{all} \cap Sources(ve_i)$.
5. For every element $Source_{keller} \in Sources_{Keller}$, check if it can reach all the elements in $Sources_{reach}$.
6. If not, then check if $card(Source_{notReach}, Source_{keller})$ = 1 for every $Source_{notReach} \in Sources_{notReach}$, where $Sources_{notReach}$ are all $Source$s that $Source_{keller}$ fail to reach. If not, $Sources_{Keller}$ = $Sources_{Keller}$ - $Source_{keller}$.
7. Do Step 2.7 to Step 2.9 in Algorithm 3.2.2.

Step 3:

8. do Step 3 in Algorithm 3.2.2.

4.2.2 Algorithm for Correctly Deleting A Set of Invisible View Elements specified by XPath in XML Scenario

The following algorithm correctly translate the deletions of a set of invisible view elements specified by XP. The idea is the same as deleting a single invisible view element. The only difference is, in order to consider predicates, we need to check side-effects over $SN_{invisible}$ using Step 2 in Algorithm 4.2.1. Let $ve_{invisible}$ be one of the view elements to delete and $ve_{visible}$ be one of its ancestors.

Algorithm 4 Algorithm 4.2.2 that correctly translating the deletion of a set of invisible view elements

---

Step 0:

0. Append all the invisible view schema nodes as children of their nearest visible ancestor.

Step 1:

1. Let $Sources'$ be the union of $Source(ve_{visible})$ and $Sources$ of all non-descendant leaf view elements of $SN(ve_{visible})$.
2. If $Des(DataSource(ve_{invisible})) \cap Sources' \neq \emptyset$, the algorithm terminates. There is no correct translation. Else go to Step 2.

Step 2:

3. Use Step 2 in Algorithm 4.2.1, where ve_i = $ve_{visible}$.
4. The algorithm terminates. If $Candidates \neq \emptyset$, then $DataSource(ve_{invisible})$ is a correct translation; otherwise there is no correct translation of deleting $ve_{invisible}$.

Chapter 5

Part V:

Conclusion and Future Work

5.1 Conclusion

In this paper we presented algorithms for correctly translating the deletion of a visible or invisible XML view element as deleting an element in the underlying XML base. We also presented algorithms for correctly translating the deletion of a set of visible or invisible view elements. Our algorithms use 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 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. Further, we need to consider more semantics and features both in XML Schema and XQuery statements.

Bibliography

[1] http://www.w3.org/xml/query/. 2006.

[2] S. Abiteboul. On views and XML. In *PODS*, pages 1–9, 1999.

[3] A. Bohannon, B. Pierce, and J. Vaughan. Relational Lenses: A Language for Updatable Views. In *PODS*, pages 338–347, 2006.

[4] V. Braganholo, S. Davidson, and C. Heuser. The updatability of xml views over relational databases. In *WEBDB*, pages 31–36, 2003.

[5] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, 2004.

[6] Y. B. Chen, T. W. Ling, and M.-L. Lee. Designing Valid XML Views. In *ER*, pages 463–478, 2002.

[7] B. Choi, G. Cong, W. Fan, and S. Viglas. Updating recursive xml views of relations. In *ICDE*, pages 766–775, 2007.

[8] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *The VLDB Journal*, pages 471–480, 2001.

[9] 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.

[10] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems (TODS)*, pages 557–575, 1981.

[11] M. Fernndez, W. Tan, and D. Suciu. Silkroute: Trading between relations and xml. In *International World Wide Web Conference*, 2000.

[12] International Organization for Standardization (ISO) & American National Standards Institute (ANSI).

[13] M. Jiang, L. Wang, M. Mani, and E. Rundensteiner. Updating views over recursive xml. In *EROW*, 2007.

[14] 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.

[15] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates Through Views: A New Hope. In *VLDB*, 2006.

[16] H. Kozankiewicz, J. Leszczyowski, and K. Subieta. Updatable xml views. *Advances in Databases and Information Systems*, pages 381–399, September 2003.

[17] P. Lehti and P. Fankhauser. Towards type safe updates in xquery. *http://www.ipsi.fhg.de/ lehti/Typing*

[18] A. Y. Levy and Y. Sagiv. Queries independent of updates. In 19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings, *pages 171–181, 1993.*

[19] Oracle Technologies Network. Using XML in Oracle Database Applications. http://technet.oracle.com/tech/xml/htdocs/about_oracle_xml_products.htm, November 1999.

[20] 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.*

[21] *I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In* Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, *pages 413–424, May 2001*.

[22] *R. Vercammen. Updating xml views.* VLDB PhD Workshop, *pages 6–10, 2005*.

[23] *L. Wang, E. A. Rundensteiner, and M. Mani. UFilter: A Lightweight XML View Update Checker. In* ICDE, poster paper, *2006.*