

An Optimized Two-Step Solution for Updating XML Views

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

Worcester Polytechnic Institute, Worcester, MA 01609, USA

Abstract. View updating is a long standing difficult problem. Given a view defined over base data sources and a view update, there are several different updates over the base data sources, called *translations*, that perform the update. A translation is said to be *correct* if it performs the update and at the same time does not update any portion of the view not specified in the update (no *view side-effects*). The view update problem finds a correct translation for a given view update if one exists. In the relational scenario, previous research has attempted to study the view update problem either by utilizing only the schema knowledge, or by directly examining the base data. While utilizing only the schema knowledge is very efficient, we are not guaranteed to find a correct translation even if one exists. On the other hand, examining the base data is guaranteed to find a correct translation if one exists, but is very time-consuming. The view update problem is even more complex in the XML context due to the nested hierarchical structure of XML and the restructuring capabilities of the XQUERY view specification. In this paper we propose a schema-centric framework, named HUX, for efficiently updating XML views specified over relational databases. HUX is complete (always finds a correct translation if one exists) and is efficient. The efficiency of HUX is achieved as follows. Given a view update, HUX first exploits the schema to determine whether there will never be a correct translation, or there will always be a correct translation. Only if the update cannot be classified using the schema, HUX will examine the base data to determine if there is a correct translation. This data-level checking is further optimized in HUX, by exploiting the schema knowledge extracted in the first step to significantly prune the space of potential translations that is explored. Experiments illustrate the performance benefits of HUX over previous solutions.

1 Introduction

Both XML-relational systems [6, 20] and native XML systems [14] support creating XML wrapper views and querying against them. However, update operations against such virtual XML views are not yet supported in most cases. While several research projects [4, 22] began to explore this XML view updating problem, they typically pick one of the translations, even if it is not correct (in other words, they might pick a translation that causes view side-effects). Allowing side-effects to occur is not practical in most applications. The user would like the updated view to reflect changes as expected, instead of bearing additional changes in the view.

One approach to solve this problem is to determine the view side-effects caused by each translation. Here, for each translation, compare the view before the update and the view after the update as in [21]. If a side-effect is detected, then the translation is rejected. However, this could be very expensive, as the space of possible translations could be very

large, and also computing the before and after image for each translation could also be very time consuming.

Some other approaches such as [16] allow the user to determine if the side-effects caused by a translation are acceptable. In other words, the users participate in setting rules for pruning down the candidate translations. However, in most cases users do not have such knowledge, and simply want the view updated exactly as expected. This is the same scenario that we assume in this paper. Therefore we employ the strategy that a translation will be directly rejected if any view side-effects are detected.

To identify the space of possible translations, the concepts of provenance [5] and lineage [11] are used, where lineage refers to descriptions of the origins of each piece of data in a view. In [11], the authors exhaustively examine every possible translation obtained from the lineage to determine if the translation causes any view side-effects. This is done by examining the actual instance data. This data-centric approach can be quite time consuming as shown in [5].

In stead of a data-centric approach, the schema of the base and the view could be used to determine a correct translation for a view update, if one exists. Such schema centric approaches are proposed for Select-Project-Join views in the relational context in [13, 15]. In [4], the authors map an XML view to a set of relational views and therefore transform XML view update problems back to relational view update problems. Such a transformation, however, is not sufficient to detect all the possible view side-effects, as we will discuss in the following two sections. A pure schema-based approach is efficient, but rather restrictive. The disadvantage of a pure schema-based approach is that, for many view update cases, it is not possible to determine the translatability (whether there exists a correct translation) by only examining the schema.

Now, with the following three examples, let us examine how the above data-based or schema-based approach can be used in XML view updates scenario. In these examples, a view update can be classified as **translatable**, which indicates there exist a side-effect free translation, or otherwise **untranslatable** using either schema or data knowledge. Also we assume that when we delete an element in the XML view, we will delete all its descendant elements as well.

1.1 Motivating Examples

Fig. 1(a) shows a running example of a relational database for a course registration system. An XML view in Fig. 1(c) is defined by the *view query* in Fig. 1(b). The following examples illustrate cases of classifying updates as translatable or untranslatable. XML update language from [22] or update primitives from [4] can be used to define update operations. For simplicity, in the examples below we only use a delete primitive with the format (*delete nodeID*), where *nodeID* is the abbreviated identifier of the element to be deleted¹. For example, C1 is the first *Course* element, P1.TA1 is the first *TA* element of the first *Professor* element. We use *Professor.t₁* to indicate the first tuple of base relation *Professor*.

Example 1. Update $u_1 = \{\text{delete P1.TA1}\}$ over the XML view in Fig. 1(c) deletes the student “Chun Zhang”. We can delete *StudentTA.t₂* to achieve this without causing any view side-effect. We can determine that any TA view element can be deleted by deleting

¹ Note that the view here is still *virtual*. In reality, this *nodeID* is achieved by specifying conditions in the update query.

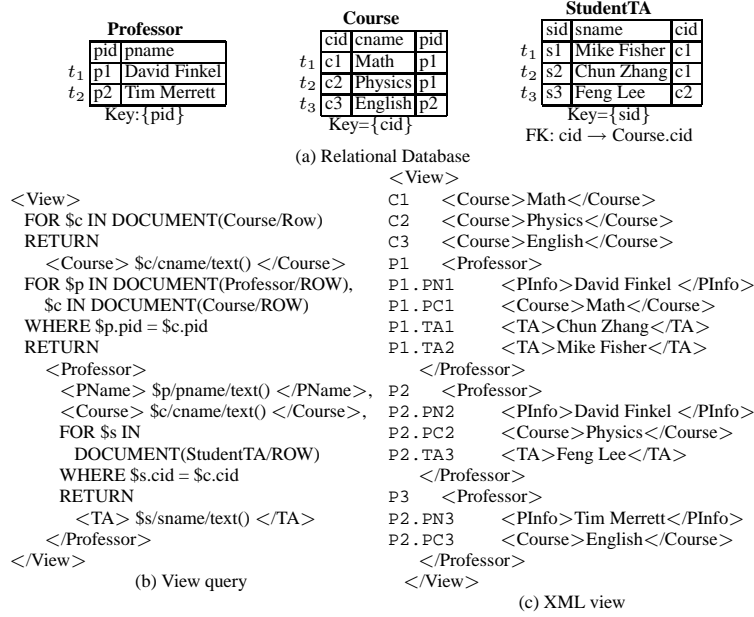


Fig. 1. Running Example of the Course Registration System

the corresponding tuple in the studentTA relation, by utilizing only the schema knowledge (as we will study in Section 4). **The schema knowledge is sufficient to determine that this update is translatable.**

Example 2. Consider the update $u_2 = \{\text{delete } P2.PC2\}$. The appearance of the view element $P2.PC2$ is determined by two tuples: $Professor.t_1$ and $Course.t_2$. There are three translations for this view update: $T_1 = \{\text{delete } Professor.t_1\}$, $T_2 = \{\text{delete } Course.t_2\}$ and $T_3 = \{\text{delete } Professor.t_1, \text{delete } Course.t_2\}$. All three translations are incorrect and cause view side-effects - translation T1 will delete the parent Professor view element (P2), translation T2 will delete one of the Course view elements (C2), translation T3 will delete both P2 and C2. We can determine that there is no correct translation for deleting any Course view element by utilizing only the schema knowledge (studied in Section 4). **The schema knowledge is sufficient to determine that this update is untranslatable.**

Example 3. Consider the update $u_3 = \{\text{delete } P3\}$. This can be achieved by deleting $Professor.t_2$. Let us consider the update $u_4 = \{\text{delete } P2\}$. The appearance of the view element P2 is determined by two tuples: $Professor.t_1$ and $Course.t_2$. However, we cannot find any correct translation for this view update: if we delete $Course.t_2$, view element C2 will also get deleted; if we delete $Professor.t_1$, view element P1 will also get deleted. The difference between u_3 and u_4 indicates that sometimes the schema knowledge itself is not sufficient for deciding translatability. **The translatability of these updates depends on the actual base data.**

1.2 HUX: Handling Updates in XML

Our Approach Our XML view updating system is called **HUX**(Handling Updates in XML). The main idea behind HUX is illustrated by Figure 2. As a user feeds an update over an XML view element into the system, HUX first utilizes the schema knowledge of the underlying source to identify whether there exists a correct translation. If we can conclude from the schema knowledge that the update is untranslatable, it will be immediately rejected. Similarly, if we can find the correct translation of the view update, we will directly pass it to the SQL engine for update execution. Only when we find that the translatability of the update depends on the actual data, we will come to the second step, where data checking is performed. By integrating schema checking and data checking together, we can guarantee to find correct translations efficiently if one exists.

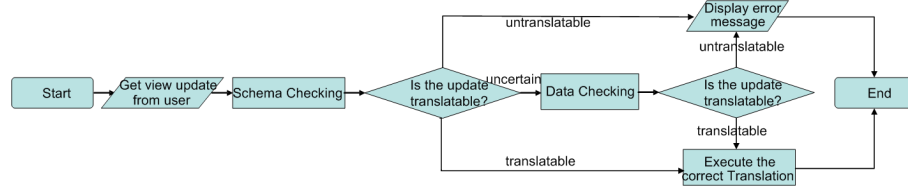


Fig. 2. Flowchart of HUX

Let us illustrate HUX for the motivating examples. During the schema level check, update u_1 is classified as translatable. We translate this update by deleting the corresponding tuple in the *StudentTA* relation. Update u_2 will be found to be untranslatable by the schema-level check and is directly rejected. Updates u_3 and u_4 cannot be classified as translatable or untranslatable by the schema-level check. Therefore we proceed to the data-level check, where we find that u_3 is translatable and u_4 is not. As the correct translation of u_3 has also been identified, it will be updated and u_4 will be rejected.

In addition, we can further optimize data-checking step by utilizing the extracted schema knowledge to prune down the search space of finding correct translations. For example, consider Example 3. When updating a Professor view element, there are three possible ways: deleting the corresponding tuple in Professor table, deleting the corresponding tuple in Course table or deleting both of them. However, as we analyzed in the schema-checking step, each Course tuple also contributes to the existence of a Course view element. Therefore we cannot delete a Professor view element by deleting its Course tuple as a Course view element will get deleted as view side-effects. Therefore, on the data level, we only need to check the Professor base tuple. Details of how to maximally utilize schema knowledge in the data-checking are presented in Section 5.

XML-Specific Challenges Note the above integration of two approaches can be applied to all data models. In this paper, we examine the scenario where the view update is against an XML view built over relational databases. For studying this problem, we will try to extend relational view update solutions to handle this scenario for the following two reasons:

1. There has been lot of work on relational view update problems, including both schema approach [1, 10, 17, 15, 13] and data approach [12].

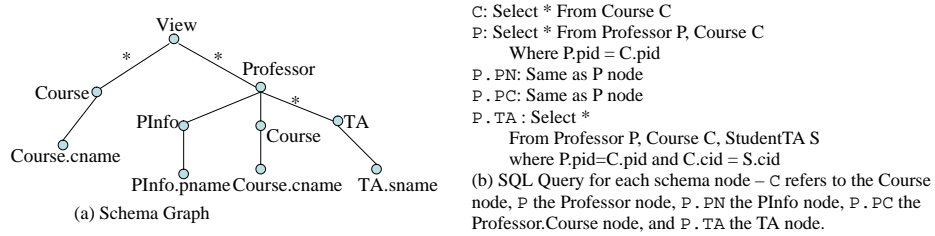


Fig. 3. Schema graph of the XML view in Figure 1

2. We can treat XML views as a "composition" of a set of relational views [20, 4]. Here, each node in the schema graph of the view (Fig. 3) can be considered as generated by a relational view, with an associated SQL query. The set of instances of a schema node is therefore given by this SQL query.

Intuitively, an update over a certain XML view schema node can be treated as an update over its relational mapping view. This in turn can be handled as relational view update problem. However, a simple transformation of XML view update problem into relational view update problem will not suffice. The relational view update problem considers side-effects only on elements with the same view query as the element to be updated. However, an XML view has many different view schema nodes, with different mapping SQL queries, and we need to examine the side-effects on all these elements. In the following sections, we will highlight these differences and explain how they are handled in our HUX approach.

Contributions. We make the following contributions in this paper. (1) We propose the first pure data-driven strategy for XML view updating, which guarantees that all updates are correctly classified. (2) We also propose a schema-driven update translatability reasoning strategy, which uses schema knowledge including now both keys and foreign keys to efficiently filter out untranslatable updates and identify translatable updates when possible. (3) We design an interleaved strategy that optimally combines both schema and data knowledge into one update algorithm, which performs a complete classification in polynomial time for the core subset of XQuery views ² (4) We have implemented the algorithms, along with respective optimization techniques in a working system called HUX. We report experiments assessing its performance and usefulness.

Outline. Section 2 formally defines the view updating problem and reviews the clean source theory. The pure data-driven side-effect checking strategy is described in Section 3, while the schema-driven one is described in Section 4. We combine the power of both schema and data checking into the schema-centric update algorithm of HUX (Section 5). Section 6 provides our evaluation, Section 7 reviews the related work and Section 8 concludes our work.

2 Preliminary

An XML view V is specified by a **view definition** DEF_V over a given relational database D . In our case, DEF_V is an XQuery expression [24] called a *view query*. Let \mathcal{U} be the

² The complexity analysis is omitted for space considerations.

domain of update operations over the view. Let $u \in \mathcal{U}$ be an update on the view V . An *insertion* adds while a *deletion* removes an element from the XML view. A *replacement* replaces an existing view element with a new one.

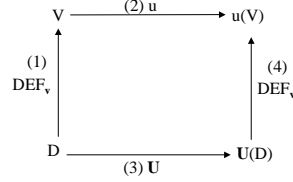


Fig. 4. Correct Translation of View Updates

Definition 1. A relational update sequence U on a relational database D is a **correct translation** of an update u on the view V iff $u(DEF_V(D)) = DEF_V(U(D))$.

A correct translation is shown in Fig. 4 holds. Intuitively, a correct translation exactly performs the view update and nothing else, namely, without view side-effects.

Clean Extended Source Theory The *clean source theory* [13], has been widely used as theoretical foundation for the relational view update problem [12, 5]. Let R_1, R_2, \dots, R_n be the set of relations referenced by the SQL query Q of a view schema node v . Informally a view element e 's **generator** $g(e)$ is $\{R_1^*, R_2^*, \dots, R_n^*\}$, where $R_i^* \in R_i (i = 1..n)$ contains exactly the tuple in R_i used to derive e . For example, the generator of the *Professor* view $P1$ in Fig. 1(c) is $g(P3) = \{Professor.t_2, Course.t_3\}$. The definition of the generator follows [13], also called *Data lineage* [12] and *Why Provenance* [5].

Further, each R_i^* is a **source** of the view element, denoted by s . For example, there are two possible sources of $P3$, namely, $s_1 = \{Professor.t_2\}$, $s_2 = \{Course.t_3\}$. A **clean source** is a source of an element used only by this particular element and no other one. For instance, s_1 is a clean source of $P3$, but s_2 is not since s_2 is also part of the generator of $C3$.

Given an element e and its source s . Let E be the set of tuples t_j in database D , which directly or indirectly refer to a tuple in the source s through foreign key constraint(s). In other words, E is the additional set of tuples that will get deleted when we delete s . Now, $extend(s) = s \cup E$ is called the **extended source** of s . For example, $extend(Professor.t_1) = \{Professor.t_1, StudentTA.t_1, StudentTA.t_2\}$.

As concluded in [13, 25], an update translation is correct if and only if it deletes or inserts a clean extended source of the view tuple. Intuitively, it means that the update operation only affects the “private space” of the given view element and will not cause any view side-effect.

However, the update translatability checking based on the clean source theory above must examine the actual base data. Also, the number of potential translations of a given update can be large [5]. Therefore we propose instead to use the schema knowledge to filter out the problematic updates whenever possible. This prunes the search space in terms of candidates we must consider. We thus introduce a set of corresponding schema-level concepts as below.

e	A view element
$g(e)$	The generator of e
s	The source of e
$extend(s)$	The extended source of s

(a) Main Concepts at the Data-level

v	A view schema node
$G(v)$	The schema-level Generator of v
S	The schema-level Source of v
$Extend(S)$	The schema-level Extended Source of S

(b) Main Concepts at the Schema-level

Fig. 5. Main Concepts Used for View Updates

Given a view element e and its schema node v . *Schema-level generator* $G(v)$ indicates the set of relations from which the generator $g(e)$ is extracted. Similarly, S denotes the set of relations the source s derived from, named *schema-level source*. For example, $G(\mathcal{P}) = \{Professor, Course\}$. Schema level sources include $S_1 = \{Professor\}$, $S_2 = \{Course\}$. Note that $S \subseteq Extend(S) \subseteq D$.

XML View Elements Classification Given an XML view update on element e , a correct translation for this update can update any descendant element of e in addition to updating e . However, this translation should *not* affect any of the non-descendant elements of e . We classify these non-descendant elements into three groups as shown in Figure 6. **Group-NonDesc** includes view elements whose schema nodes are non-descendant ones of v . **Group-Self** includes those whose schema node is v . **Group-Desc** includes those whose schema nodes are descendants of v .

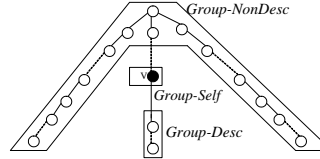


Fig. 6. Schema Tree Structure

For example, let the view element e be $\mathcal{P}1$ from Fig 1. Then Group-NonDesc includes $\mathcal{C}1, \mathcal{C}2, \mathcal{C}3$. Group-Self includes $\mathcal{P}2, \mathcal{P}3$. Group-Desc includes $\mathcal{P}2.PN2, \mathcal{P}2.PC2, \mathcal{P}2.TA3, \mathcal{P}3.PN3, \mathcal{P}3.PC3$. For updating a view element e , if there is a translation that performs the update without affecting any element of any of the three groups, then this is a correct translation of the given update.

3 Data-driven View Updating

Using clean source theory, most commercial relational data base systems [21, 2, 8] and some research prototypes [5, 12] directly issue SQL queries over the base data to identify view side-effects. If any clean source (exclusive data lineage [12]) is found to exist, then this source can be a correct translation. Below we extend this approach to find a clean extended source for updating elements in an XML view.

Given the generator $g(e) = \{R_1^*, R_2^*, \dots, R_n^*\}$ of a view element e of a schema node v . Intuitively, deleting any R_i^* from the generator will certainly delete the element e . However, when R_i^* get deleted, $extend(R_i^*)$ (all the base tuples that directly or indirectly refer to it) will also get deleted. Thus if any $t' \in extend(R_i^*)$ also belongs to $g(e')$, where e'

is another view element other than e or its descendants, e' will also get deleted. This is a view side-effect and implies R_i^* cannot be a correct translation. Therefore, if R_i^* is a correct translation, all the view elements should remain unchanged after deleting $extend(R_i^*)$, except e and its descendants.

Let D and D' be relational database instances before and after deleting R_i^* ³. Also, Q^v is the SQL query for a schema node v in view schema graph. Then $Q^v(D)$ and $Q^v(D')$ stand for sets of view elements that correspond to v before and after deleting $extend(R_i^*)$ respectively. Thus, $Q^v(D) - Q^v(D')$, **denoted as** $\lambda(v)$, are view elements that are deleted by deleting R_i^* . The following three rules detect side effects in each of the three groups of nodes (Fig. 6). The rules are self-explanatory and we skip examples for the first two rules for space considerations.

Rule 1 Consider Group-NonDesc node v' . Deleting R_i^* from $g(e)$ will delete the element e without causing side-effect on any element e' of v' if $\lambda(v') = \emptyset$.

Rule 2 Consider Group-Self node v' . Deleting R_i^* from $g(e)$ will delete the element e without causing side-effect on any element e' of v' if $\lambda(v') = e$.

For a schema node v' in Group-Desc, e' of v' will get deleted if e' is a descendant of e . For example, consider $c1$ in Figure 7(c). Its generator is $\{Course.t_1\}$. Let R_i^* be $Course.t_1$. When R_i^* gets deleted, $extend(Course.t_1) = \{Course.t_1, StudentTA.t_1, StudentTA.t_2\}$ gets deleted as well. This in turn deletes $s1'$, $s1''$, $s2'$ and $s2''$ in Figure 7(c), which causes view side-effects.

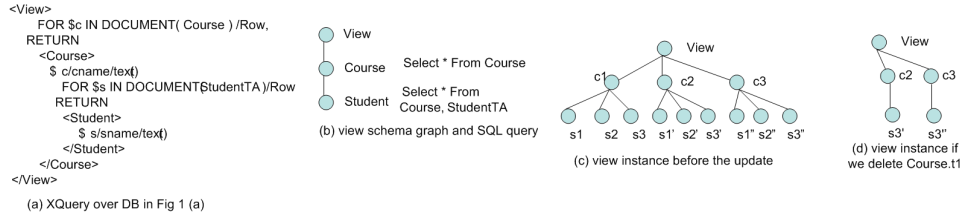


Fig. 7. Examples for Group-Desc

Let us make it more general. Let the SQL query for node v be $Q^v(R_1, R_2, \dots, R_n)$. Let v' be a node in Group-Desc, whose SQL query is $Q^{v'}(R_1, R_2, \dots, R_n, S_1, S_2, \dots, S_m)$ (note that $Q^{v'}$ will include all relations specified in Q^v). For an element e of node v , let $g(e) = \{R_1^*, R_2^*, \dots, R_n^*\}$. The elements of v' that are descendants of e are given by $Q^{v'}(R_1^*, R_2^*, \dots, R_n^*, S_1, S_2, \dots, S_m)$. To determine if deleting R_i^* causes side effects on elements of v' , we need to check whether the elements of v' that are deleted are those that are descendants of e , as stated below.

Rule 3 Consider Group-Desc node v' . Deleting R_i^* will delete e without causing side-effects on elements of node v' if $\lambda(v') = Q^{v'}(R_1^*, R_2^*, \dots, R_n^*, S_1, S_2, \dots, S_m)$.

³ As $extend(R_i^*)$ gets deleted when deleting R_i^* , terms "deleting R_i^* " and "deleting $extend(R_i^*)$ " are used interchangeably.

Theorem 1. Consider a source R_i^* of e . For $\forall v$, where v is a schema node in the view, if there are no side effects caused by deleting R_i^* according to Rules 1 - 3, then R_i^* is a correct translation for deleting e .

4 Schema-driven View Updating

In the previous section, we described the approach of identifying side-effects by *examining the actual base data*. This approach is *correct* and *complete*, meaning we can always reject all untranslatable updates and identify all translatable updates. However, this data examination step could be quite expensive. We now propose a more effective solution based on schema knowledge. Using schema knowledge only, we will classify an update as *untranslatable* (Example 2), *translatable* (Example 1) and *uncertain* (Example 3).

On the schema level, when we classify an update as "translatable", it implies for all the database instances, there exists a correct translation for this update. Just like in Section 3, we will set three rules to check if the given user view update can achieve this in Section 4.1. Similarly, when we classify an update as "untranslatable", it implies for all the database instances, any translation of this update will always cause side-effects. We will discuss this in Section 4.2. For those updates that we cannot guarantee it is translatable or untranslatable, we will examine the actual base data to classify it.

4.1 Schema-level Translatable Updates

To classify an update as always translatable (Example 1), we have to check whether a clean source always exists for any update on the schema node. The following rules are used to identify whether it is possible to delete a source without ever causing any side-effects on Group-NonDesc, Group-Self and Group-Desc nodes. Note as we need to guarantee this update is translatable for all database instances, the following rules tend to be conservative, which means they may be too strict for some specific instances. For a specific database instance, we will not reject an view update if none of its translations satisfy all these rules, we only claim that we haven't found a correct translation.

Similar to the idea in data-checking in Section 3, we will consider every Source $S \in G(v)$ as a translation, as deleting the generator $R_i^* \in S$ will delete e . For each S , we use the following rules to check if deleting tuples from R_i will cause side-effects on nodes in different groups. We again describe these rules without examples as they are self-explanatory and for space considerations.

Rule 4 Given a view schema node v . Deleting a source $S \in G(v)$ will not cause any side-effect on view element of node v' in Group-NonDesc if $Extend(S) \cap G(v') = \emptyset$.

For elements in Group-Self, the approach is similar to that for the relational view update problem [15, 13]. Here, we first construct a Computation Dependency Graph as below.

Definition 2. Computation Dependency Graph \mathcal{G}_C .

1. Given a view schema node v computed by SQL query Q^v . Let R_1, R_2, \dots, R_n be relations referenced by Q^v . Each R_i , $1 \leq i \leq n$ forms a node in \mathcal{G}_C .
2. Let R_i, R_j be two nodes ($R_i \neq R_j$). There is an edge $R_i \rightarrow R_j$ if Q has a join condition of the form $R_i.a = R_j.b$ and $R_j.b$ is UNIQUE in R_j .

3. If Q has a join condition $R_i.a = R_j.b$ where $R_i.a$ is *UNIQUE* for R_i and $R_j.b$ is *UNIQUE* for R_j , then there are two edges $R_i \rightarrow R_j$ and also $R_j \rightarrow R_i$.

Fig. 8 shows the computation dependency graph for P and $P.TA$ nodes in Fig 3.

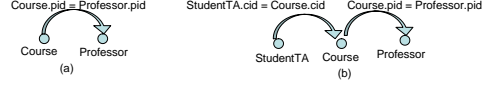


Fig. 8. (a) \mathcal{G}_C of P-node and (b) \mathcal{G}_C of P.TA-node

Rule 5 Given a view schema node v and its computation dependency graph \mathcal{G}_C . Deleting a source S of v will not cause side-effect in any view element of Group-Self if the corresponding node of S in \mathcal{G}_C can reach all other nodes.

For a node v' in Group-Desc, we check that the view query for v' captures all the key/foreign key constraints between any relation in $G(v') - G(v)$ and the Source S of v . We say that a query captures a key/foreign key constraint, fk references k, if the query includes a predicate of the form $k=fk$. Intuitively, this ensures that when a tuple of S is deleted, the only view elements that are affected are the descendants of e , the view element to be deleted.

Rule 6 Deleting a source S of v will not cause any view side-effect in any view element of node v' in Group-Desc if for any $R_j \in G(v') - G(v)$, key/foreign key constraints between R_j and S are captured in the query for v' .

Theorem 2. For a given view update u , if a translation U does not cause side-effects on Group-NonDesc nodes (by Rule 4, on Group-Self nodes (by Rule 5) and on Group-Desc nodes (by Rule 6), then U is a correct translation.

4.2 Schema-level Untranslatable Updates

Schema-level untranslatable updates are determined by the rule below. If the generator of a view schema node v is the same as that of its parent node, then it is guaranteed that deletion of any source of an element of v will cause side-effects on its parent element. See Example 2 in Section 1.1.

Rule 7 Given a view schema node v , and its parent schema node v_p , a source S of v will cause side-effects on v_p if $S \in G(v_p)$.

Theorem 3. Given a view schema node v , and its parent schema node v_p , if $G(v) = G(v_p)$, then there is no correct translation for updating any element of v .

4.3 Schema-level Uncertain Updates

If for a view schema node v , we cannot determine at the schema-level whether an update of an element of v is translatable or untranslatable, we say that update of this node is schema-level uncertain.

5 Schema-centric XML View Updating Algorithm

Given a user view update, after applying rules defined in Section 4, we may not still be able to classify it as translatable or not. In such cases, we can use rules defined in Section 3 to determine which translations are correct. With those rules, we can always classify an update as translatable or untranslatable. Further, the observations made during the schema-checking can be used to optimize the data-checking.

For example, consider the updates u_3 and u_4 in Example 3 in Section 1. Here, v is P in the schema graph shown in Figure 3. There are two relations in $G(v)$: *Professor* and *Course*. After applying the schema checking rules on these two translations separately, we cannot find a correct translation, and these updates are classified as “uncertain”. Now we perform the data-checking. However, the schema-checking reveals that *Course* is not a correct translation, and hence this need not be checked. Further, while checking *Professor*, the schema-checking reveals that there will be no side-effects on $P.TA$, and hence side-effects on $P.TA$ need not be examined during data-checking.

In summary, when doing data-checking, schema-checking knowledge can help us in two ways:

1. Prune down translations. Those translations identified as incorrect on the schema-level need not be considered at the data-level.
2. Prune down view schema nodes. Given a translation R_i and a schema node v' in Group-NonDesc, Group-Self or Group-Desc, if R_i will never cause any side-effects on v' , there is no need to check side-effects on v' on the data-level.

In order to pass the observations from the schema-level checking to the data-level checking, we introduce a data structure $SS(v)$ (Search Space), that keeps track of the observations made with regards to the update of an element of v . The column names are all the relations in $G(v)$ (they are the different possible translations). The row names are nodes in the schema graph of the view (the nodes for which we need to determine whether there will be side-effects). Consider the view defined in Figure 1 in Section 1. The initial Search Space (before any rules are applied) consists of view schema nodes $P, P.PN, P.PC, P.TA, C$ are shown in Figure 9.

view schema nodes \ translations	Course
P	
P.PN	
P.PC	
P.TA	
C	

view schema nodes \ translations	Professor	Course
P		
P.PN		
P.PC		
P.TA		
C		

view schema nodes \ translations	Professor	Course	StudentTA
P			
P.PN			
P.PC			
P.TA			
C			

Fig. 9. Initial Search Space for view schema nodes in Figure 3

During the schema-level checking, if we determine that a relation R_i in $G(v)$ does not cause side-effects on a view schema node v' , we denote it as “ \checkmark ” in the cell (R_i, v') . On the other hand, if we determine that R_i will cause side-effects on v' , we denote it as “ \times ” in the cell (R_i, v') . If the schema-level check cannot provide any guarantees as to whether there will be side-effects on v' or not, then the cell is left blank. Let us illustrate how the schema level checking proceeds for our example in Fig. 1.

First Step: Schema-Checking After applying Rule 7 on each relation in $G(v)$, $SS(P)$ and $SS(C)$ remain unchanged. For the rest of view schema nodes, their search spaces are updated as in Figure 10. This also implies we should not consider *Professor* for deleting view elements of $P.PN$, $P.PC$ and $P.TA$ nodes. Now, we can classify any update on $P.PC$ and $P.PN$ as untranslatable, as every one of their Sources will cause side-effects.

translations view schema nodes	Professor	Course
P	X	X
P.PN		
P.PC		
P.TA		
C		

translations view schema nodes	Professor	Course	StudentTA
P	X	X	
P.PN			
P.PC			
P.TA			
C			

Fig. 10. Search Spaces after applying Rule 7

Now, we utilize rules in Section 4.1. Currently, we need to check translation *Course* for C , *Professor* for P and *StudentTA* for $P.PA$. After applying the three rules, updated search spaces are shown in Figure 11. We classify any update on $P.TA$ as translatable, and the correct translation is to delete from *StudentTA*.

translations view schema nodes	Professor	Course
P		
P.PN	v	
P.PC	v	
P.TA	v	
C	v	X

translations view schema nodes	Course
P	
P.PN	
P.PC	
P.TA	
C	v

translations view schema nodes	Professor	Course	StudentTA
P	X		v
P.PN		X	v
P.PC		X	v
P.TA			v
C			v

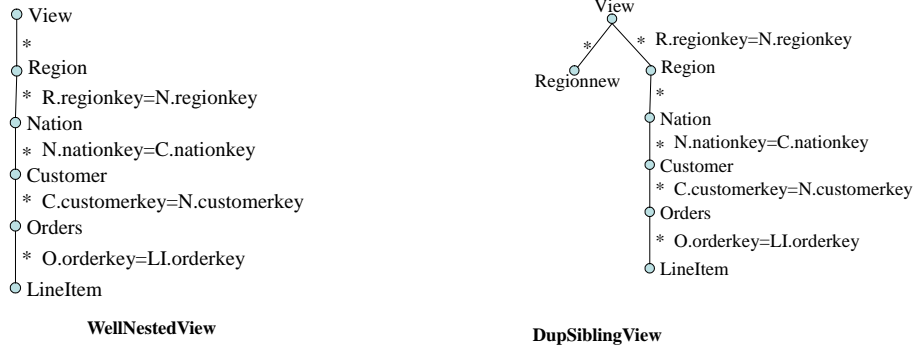
Fig. 11. Search Spaces after applying rules in Section 4.1

Second Step: Data-Checking Updates on P and C nodes are classified as uncertain by the schema-level checking, therefore given a user update, we need to do a data-level check. However, for update of an element of node P , we need to check whether the source from *Professor* table will cause any side-effect on elements of node P , which implies we need to use Rule 2 only. Now, for updates u_3 and u_4 in Example 3, we find that u_3 is translatable, and the correct translation is delete $\{Professor.t_2\}$; u_4 is untranslatable.

6 Evaluation

We conducted experiments to address the performance impact of our system. The test system used is a dual Intel(R) PentiumIII 1GHz processor, 1G memory, running SuSe Linux and Oracle 10g. The relational database is built using TPC-H benchmark [23]. Two views are used in our experiments, with their view schemas shown below.

The cost of the schema-level marking and checking as the view query size increases is shown in Fig. 12. The view L uses only the lineitem table; the view LO uses the lineitem and orders table, and so on, and the views are WellNestedViews. The schema-level marking is done at compile time, and the time for this marking increases with the view query



size. However, given an update, the check involves only checking the mark for that node and hence it is constant, and negligible.

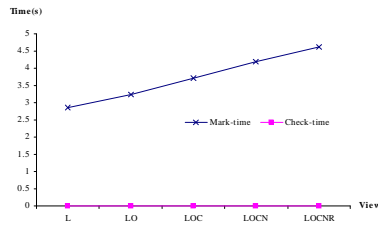


Fig. 12. Schema-Level Checking Performance

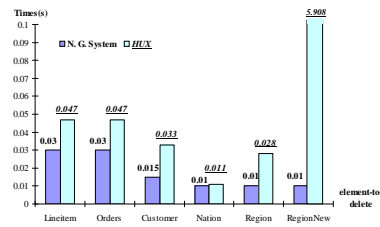


Fig. 13. HUX vs. Non-Guaranteed System

Run-time Overhead of HUX Let us compare the run-time overhead of HUX. The run-time costs of HUX include: check the compile-time mark, do a run-time check if needed, find the correct translation and perform it (if translatable). We compared HUX against a Non-Guaranteed (NG) System that arbitrarily chooses a translation. The comparison using the DupSiblingView is shown in Fig. 13. Updates on Lineitem, Orders, Customer and Region are found to be translatable at the schema-level. For effective comparison, we assume that the NG system magically finds the correct translation. Note the small overhead for HUX. Nation is determined to be schema-level untranslatable. Updates on RegionKey require a data-level check and hence is expensive.

HUX vs. Data-based View Update System. If schema-level check can classify an update as translatable or untranslatable, then HUX is very efficient as compared to data-based systems. Fig 14 shows that the time taken at run-time for HUX stays constant. However, the data level check is much more expensive for different view query sizes. The best case for data level check is when the first translation checked is a correct translation; the worst case is when all the translations except the last are incorrect translations.

We also compared HUX against a pure data-based XML updating system as in Section 3. Note that HUX is very efficient except when data-level checks also need to be performed, as for updates against elements of Regionnew. See Fig. 15.

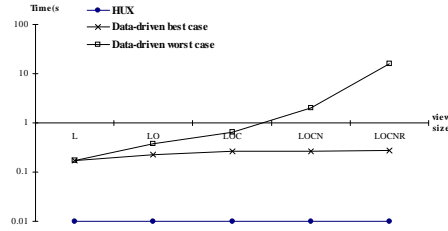


Fig. 14. HUX vs. relational view update system

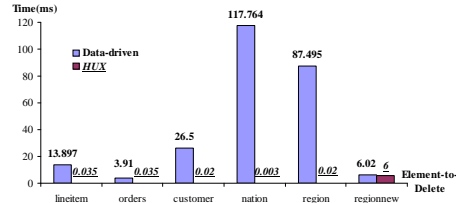


Fig. 15. HUX vs. Pure data-based view update system

7 Related Work

[17, 15] study view update translation mechanism for SPJ queries on relations that are in BCNF. [3] further extends it to object-based views. [22] studies execution cost of updating XML views using triggers versus indices. Recent works [4] study the update over *well-nested* XML views. However, as authors map XML view into relational view updating problem, some of the constraints in XML views cannot be captured. [19, 7] 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. Our work in this paper is *orthogonal* to these works by addressing new challenges related to the decision of translation existence instead of assuming a view to be well-nested or valid.

We have studied schema-based approaches for checking the existence of a correct translation [25]. This work combines the schema-based approaches with data-based approaches efficiently to perform more updates. The concept of generator that we used is borrowed from [13], and is similar to the concepts of *data provenance* [5] or *lineage* [12]. In a recent work [18], the authors propose an approach to store auxiliary data for relational view updates; in their framework all updates are translatable. Another recent work [9] studies the problem of updating XML views materialized as relations. However, they do not consider performing the updates against the original relations from which the XML view was published as studied in this work.

8 Conclusion

In this paper, we have proposed an efficient solution for the XML view update problem. A progressive translatability checking approach is used to guarantee that only translatable updates are fed into the actual translation system to obtain the corresponding SQL statements. Our solution is *efficient* since we perform schema-level (thus very inexpensive) checks first, while utilizing the data-level checking only at the last step. Even during this last step, we utilize the schema knowledge to effectively reduce the search space for finding the translation. Our experiments illustrate the benefits of our approach. Our approach can be applied by any existing view update system for analyzing the translatability of a given update before its translation is attempted.

References

1. F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, pages 557–575, Dec 1981.

2. S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - The XML Enabled Data Management System. In *ICDE*, pages 561–568, 2000.
3. T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *SIGMOD*, pages 248–257, 1991.
4. 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.
5. P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.
6. M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.
7. Y. B. Chen, T. W. Ling, and M.-L. Lee. Designing Valid XML Views. In *ER*, pages 463–478, 2002.
8. J. M. Cheng and J. Xu. XML and DB2. In *ICDE*, pages 569–573, 2000.
9. B. Choi, G. Cong, W. Fan, and S. Viglas. Updating Recursive XML Views of Relations. In *ICDE*, 2007.
10. S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. *Journal of the Association for Computing Machinery*, pages 742–760, Oct 1984.
11. Y. Cui and J. Widom. Run-time translation of view tuple deletions using data lineage. In *Technique Report, Stanford University*, June 2001.
12. Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. In *ACM Transactions on Database Systems*, volume 25(2), pages 179–227, June 2000.
13. 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.
14. H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native xml database. In *VLDB*, 2002.
15. 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.
16. A. M. Keller. Choosing a view update translator by dialog at view definition time. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 467–474. Morgan Kaufmann, 1986.
17. A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.
18. Y. Kotidis, D. Srivastava, and Y. Velegarakis. Updates Through Views: A New Hope. In *VLDB*, 2006.
19. T. W. Ling and M.-L. Lee. A Theory for Entity-Relationship View Updates. In *ER*, pages 262–279, 1992.
20. M. Fernandez et al. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.
21. 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.
22. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, May 2001.
23. TPCH. TPC Benchmark H (TPC-H). <http://www.tpc.org/information/benchmarks.asp>.
24. W3C. XQuery 1.0 Formal Semantics. <http://www.w3.org/TR/query-semantics/>, June 2003.
25. L. Wang, E. A. Rundensteiner, and M. Mani. Updating XML Views Published Over Relational Databases: Towards the Existence of a Correct Update Mapping. In *DKE Journal*, 2006.