WPI-CS-TR-05-11 July 2005

U-Filter: A Lightweight XML View Update Checker

by

Ling Wang
Elke A. Rundesnteiner and Murali Mani

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

U-Filter: A Lightweight XML View Update Checker

Ling Wang, Elke A. Rundensteiner and Murali Mani
Worcester Polytechnic Institute, Worcester, MA 01609, USA
{lingw|rundenst|mmani}@cs.wpi.edu

October 12, 2005

Abstract

We study in this paper the problem of whether a correct relational update translation can be found for a given update over an XML view. For this, we propose a lightweight update checking framework named *U-Filter*. It first performs two steps of schema-level (and thus very inexpensive) checks based on a view definition analysis. Only when necessary, a third checking step, requiring base data access and thus more expensive, is employed. For the latter, we design an *internal* strategy as well as an *external* strategy (with respect to the DBMS). This three-step checking process is guaranteed to filter out all XML updates that cannot be translated. Finally, the remaining updates are fed to the update translation engine, which generates the corresponding SQL update statements. Our experiments illustrate the usefulness of *U-Filter* and the performance impact achievable by the proposed algorithm.

1 Introduction

Both XML-relational systems such as [13, 25] and native XML systems such as [20] support creating XML wrapper views and querying against them. However, update operations against such virtual XML views in most cases are not supported yet.

Two problems concerning updating XML views need to be tackled. First, *update translatability* concerns whether some updates on the base data storage, which typically may be a relational database or a native XML document, can be made to effect the given update to the view without causing any view-side-effect [3, 16, 18]. Second, we need to devise an appropriate *translation strategy*. That is, assuming the view update is indeed translatable, how to map the updates on the XML view into the equivalent tuple-based SQL updates or XML document updates on the base data.

The second issue, the translation strategy, has been studied in recent works [4, 7, 8, 15, 28, 29]. Under the assumption that the given update is translatable, [7, 8] propose an approach to convert the XML view update problem into relational view update problem. [29] studies the execution performance of translated updates. Commercial database systems such as SQL-Server2000 [28], Oracle [4] and DB2 [15] also provide system-specific solutions for restricted update types, again under the assumption of the given updates always being translatable.

Based on the idea of data provenance (lineage) – the description of the origins of each piece of data in a view, recent works [10, 11, 17] indicate a loose connection between the concept of provenance and the view update problem. The distinction between "why provenance" (the source data that had some influence on the existence of the data) and "where provenance" (the location(s) in the source databases from which the data was extracted) is used to indicate the potential correct translation. However, these works do not answer the questions important to update translatability such as (i) whether the why or where provenance is the correct translation and (ii) if they are not, whether there *exists* at least another correct translation?

This update translatability issue is important in terms of both correctness and performance. Without translatability checking, blindly translating a given view update into relational updates can be dangerous. Such blind translation may result in *view side effects*. To identify this, the view before the update and after the update would have to be compared as done in [28]. To adjust for such an error, the view update would have to be rejected and the database would have to be recovered for example by rolling back. This would be rather time consuming, depending on the size of the database. However, by performing an update translatability analysis, such ill-behaved updates could instead be identified early on and rejected, and it would be less costly.

In this paper, we propose a general methodology to assess the translatability of an update over an *arbitrary* XML view of a relational database, when various schema level conflicts and data level conflicts potentially exist.

XML view update translatability problem is more complex than that of pure relational view update translatability [3, 16, 18]. Not only do all the problems in the relational context still exist in XML semantics, but we also have to address the new update issues introduced by the XML hierarchical data model and its flexible update language. Especially, when duplications and inconsistent constraints between the view and the database schema exist, the problem is further complicated as shown by examples below.

publisher

	pubid	pubname
t_1	A01	McGraw-Hill Inc.
t_2	A02	Simon & Schuster Inc.
t_3	B01	Prentice-Hall Inc.

Legend: ▨ Primary Key □ Non Key

book

	bookid	title	pubid	price	year
t_1	98001	TCP/IP Illustrated	A01	37.00	1997
t_2	98002	Programming in Unix	A02	45.00	1985
t_3	98003	Data on the Web	A01	48.00	2004

review

	bookid	reviewid	comment	reviewer
t_1	98001	001	A good book on network.	William
t_2	98001	002	Useful for advanced user.	John

```
CREATE TABLE publisher(
   pubid VARCHAR2(10),
   pubname VARCHAR2(100) UNIQUE NOT NULL,
   CONSTRAINTS PubPK
         PRIMARYKEY (pubid))

CREATE TABLE book(
   bookid VARCHAR2(20),
   title VARCHAR2(100) NOT NULL,
   pubid VARCHAR2(10),
   price DOUBLE CHECK (price > 0.00),
   year  DATE,
   CONSTRAINTS BookPK
         PRIMARYKEY (bookid),
   FOREIGNKEY (pubid)
         REFERENCES publisher (pubid))

CREATE TABLE review(
   bookid VARCHAR2(20),
   reviewid VARCHAR2(3),
   comment VARCHAR2(100),
   reviewer VARCHAR2(10),
   CONSTRAINTS BookPK
         PRIMARYKEY (bookid,reviewid),
   FOREIGNKEY (bookid)
         REFERENCES book (bookid))
```

Figure 1: Relational Database of Running Example

```
<DB>
   <publisher>
      <row>
         <pubid>A01</pubid>
         <pubname> McGraw-Hill Inc. </pubname>
      </row> ...
   </publisher>
   <book>
      <row>
         <bookid>98001</bookid>
         <title>TCP/IP Illustrated</title>
         <pubid>A01</pubid>
         <price>37.00</price>
         <year>1997</year>
      </row> ...
   </book>
   <review>
      <row>
         <bookid>98001</bookid>
         <reviewid>001</reviewid>
         <comment>A good book on network.</comment>
         <reviewer>William</reviewer>
      </row> ...
   </review>
<DB>
```

Figure 2: Default XML View of Database in Fig. 1

```
<BookView>
FOR $book IN document("default.xml")/book/row,
      $publisher IN document("default.xml")/publisher/row
WHERE ($book/pubid = $publisher/pubid)
AND ($book/price<50.00) AND ($book/year > 1990)
RETURN {
   <book>
      $book/bookid, $book/title, $book/price,
      <publisher>
         $publisher/pubid, $publisher/pubname
      </publisher>,
      FOR $review IN document("default.xml")/review/row
      WHERE ($book/bookid = $review/bookid)
      RETURN{
       <review>
            $review/reviewid, $review/comment
       </review>}
   </book>},
FOR $publisher IN document("default.xml")/publisher/row
RETURN{
      <publisher>
         $publisher/pubid, $publisher/pubname
      </publisher>}
</BookView>
```

(a)

```
<BookView>

   <book>
      <bookid>98001</bookid>
      <title>TCP/IP Illustrated</title>
      <price>37.00</price>
      <publisher>
         <pubid>A01</pubid>
         <pubname> McGraw-Hill Inc. </pubname>
      </publisher>
      <review>
         <reviewid> 001 </reviewid>
         <comment>
            A good book on network.
         </comment>
      </review >
      <review>
         <reviewid> 002 </reviewid>
         <comment>
            Useful for advanced user.
         </comment>
      </review >
   </book>
<book>
   <bookid>98003</bookid>
   <title>Data on the Web</title>
   <price>48.00</price>
   <publisher>
      <pubid>A01</pubid>
      <pubname> McGraw-Hill Inc. </pubname>
   </publisher>
</book>

<publisher>
   <pubid>A01</pubid>
   <pubname> McGraw-Hill Inc. </pubname>
</publisher>
<publisher>
   <pubid>A02</pubid>
   <pubname> Simon & Schuster Inc </pubname>
</publisher>
<publisher>
   <pubid>B01</pubid>
   <pubname> Simon & Schuster Inc </pubname>
</publisher>

<BookView>
```

(b)

Figure 3: XML Views (b) defined by the View XQuery (a) over Relational Database in Fig. 1

1.1 Motivation Example

Fig. 1 shows a running example of a relational schema and sample data of a book database. Recent XML systems (XPERANTO [13], SilkRoute [19]) use a basic XML view, called *default XML view*, to define the one-to-one relational-to-XML mapping (Fig. 2). On top of this default XML view, a *virtual view* can be introduced to define user-specific XML wrapper views. Such a virtual view (Fig. 3b) can be specified by an XML query expression called a *view query* (Fig. 3a). Updates are specified in our work by adopting the syntax from [29]. Fig. 4 shows several examples of view updates.

Example 1 *In Fig. 4, u_1 inserts a new book element into BookView. We notice that the title of the new book is empty and the price is "0.00". However, the underlying relational schema has the constraints that the title of book tuples is NOT NULL, while the price of the book tuple should be a positive number. Thus, u_1 is not translatable since it directly conflicts with the check constraints from the relational schema.*

Example 2 *u_2 in Fig. 4 deletes the publisher of the first book. In the underlying relational database, there is a foreign key from book relation to publisher relation. So, when the publisher is deleted, the corresponding book tuple has to be either also deleted, or the pubid of the book needs to be replaced with NULL, depending on the deletion policy defined by the foreign key constraints. However, neither of these two are correct because they both would cause the side-effect of the corresponding book to no longer appear in the view. We thus say that u_2 is not translatable since it causes a view side effect.*

u₁

```
FOR $root IN document(“BookView.xml")
UPDATE $root {
   INSERT
      <book>
         <bookid>"98004"<bookid>
         <title> </title>
         <price> 0.00 </price>
         <publisher> … </publisher>
      </book> }
```

u₂

```
FOR $root IN document(“BookView.xml"),
      $book IN $root/book
WHERE $book/bookid/text() = " 98001"
UPDATE $root {
         DELETE $book/publisher}
```

u₃

```
FOR $book IN
document(“BookView.xml")/book
WHERE
$book/title/text() = “DB2 Universal Database"
UPDATE $book {
   INSERT
      <review>
            <reviewid>001</reviewid>
            <comment>
               Easy read and useful.
            </comment>
      </review>}
```

u₄

```
FOR $root IN document(“BookView.xml")
UPDATE $root {
   INSERT
      <book>
         <bookid>"98001"<bookid>
         <title>" Operating Systems "</title>
         <price> 20.00 </price>
         <publisher>
            <pubid>A01</pubid>
            <pubname>
               McGraw-Hill Inc.
            </pubname>
         </publisher>
      </book> }
```

Figure 4: Updates over View in Fig. 3

Example 3 *The update u_3 in Fig. 4 inserts a review for the book “DB2 Universal Database”, while this book is not in the view. And u_4 inserts a new book which conflicts with an existing book ($book.t_1$), since they both have “bookid=98001”. Both u_3 and u_4 are said to be not translatable.*

1.2 U-Filter: Our Approach for View Update Checking

We note from above examples that the potential conflicts in both schema or data level can affect the translatability of a given view update. To address these factors we propose a lightweight view update checking framework called *U-Filter*. It first performs two steps of schema-level (and thus very inexpensive) checking. Only when necessary, more expensive checking requiring the base data to be accessed is employed.

The first step, called *update validation*, identifies whether the given view update is valid according to the *view schema*. This can be pre-defined [6] or be inferred from the view definition query and the base relational schema knowledge. Given that lots of work has already been done in the literature on schema validation [6, 26], here we focus only on questions closely related with the view update scenario, such as which constraints should the validation procedure consider. The problem in Example 1 is identified by this step.

In the second step, called *schema-driven translatability reasoning*, any update determined to be valid by Step 1 is further examined. Here the potential view side effects are checked, which can be caused by different reasons such as (i) foreign key constraints conflicting with the view structure or (ii) base data duplication in the view. This compile-time check only utilizes the view query and the relational schema. Example 2 is identified to be not translatable in this step.

Updates passed the previous two steps could potentially still conflict with the base data (Example 3). In our third step, the run-time *data-driven translatability checking*, such conflicts will be identified. This check can only be resolved by examining actual base data. This is typically rather expensive. Hence it is practical to employ this only at the end, when the prior check steps have already been considered and the update has successfully passed these filters.

Fig. 5 shows the framework of U-Filter. We present algorithms and optimizations for each step in U-Filter. This represents a practical approach that could be applied by any existing view update system for analyzing the translatability of a given view update before translation of it is attempted.

Contributions. Our contributions in this paper include: (1) We propose a lightweight three-step framework called *U-Filter* as a practical solution for XML view update translatability problem. (2) We identify the constraints existing in either the view or relational base that need to be considered for the *update validation* and model them using *Annotated Schema Graph*. (3) We propose a *Schema-driven TrAnslatability Reasoning* algorithm (STAR) to classify an XML view update into different translatability categories. (4) We design several *Data-driven Translatability Checking* approaches to identify the untranslatable updates caused by data level conflicts. (5) We conduct a variety of experiments to assess the performance and usefulness of our *U-Filter* approach.

Outline. Section 2 formally defines the problem we are tackling in this paper. Section 3 describe the internal query representation in U-Filter named Annotated Schema Graph. Sections 4, 5 and 6 describe each of the three checking steps respectively. Section 7 provides an evaluation of our solution. Section 8 reviews the related work while Section 9 concludes our work.

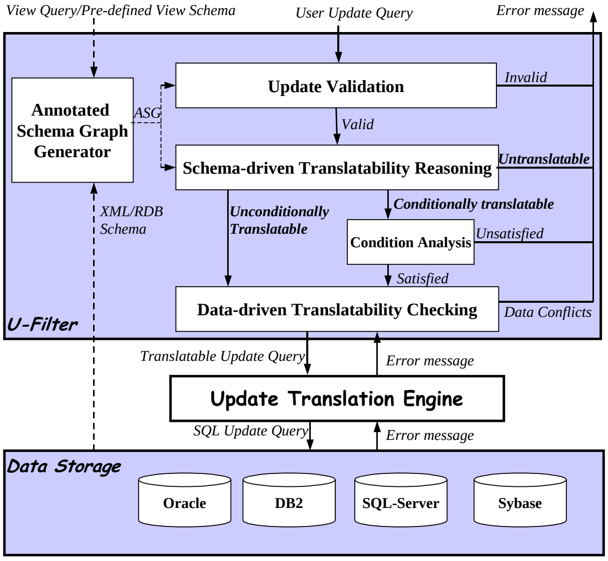

View Query/Pre-defined View Schema
User Update Query
Error message
Update Validation
Invalid
Annotated Schema Graph Generator
ASG
Valid
Schema-driven Translatability Reasoning
Untranslatable
XML/RDB Schema
Unconditionally Translatable
Conditionally translatable
Condition Analysis
Unsatisfied
Satisfied
Data-driven Translatability Checking
Data Conflicts
U-Filter
Translatable Update Query
Error message
Update Translation Engine
SQL Update Query
Error message
Data Storage
Oracle
DB2
SQL-Server
Sybase

Figure 5: Framework of U-Filter

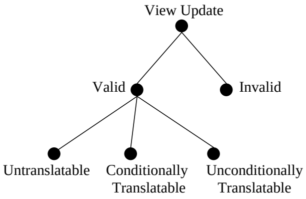


Figure 6: The partition of view update domain $\mho$

$$\begin{tikzcd}
V \arrow[r, "(2)\ u"] & u(V) \\
D \arrow[u, "(1)\ DEF_v"] \arrow[r, "(3)\ \mathbf{U}"'] & \mathbf{U}(D) \arrow[u, "(4)\ DEF_v"']
\end{tikzcd}$$

Figure 7: Correct translation of view update to relational update

2 Problem Definition

The relational data model can be described as below. For a relation R, let $\mathcal{A} = \{a_1, a_2, ..., a_m\}$ be its attribute set. A relational database is represented as D, and its schema as $\{(R_1, R_2, \ldots, R_n), \mathcal{F}\}$, where $R_1, R_2, \ldots, R_n$ are the relations in D, and $\mathcal{F}$ is the set of constraints. The **XML view** V is defined by a **view definition** DEF_V over a given relational database D. In our case, DEF_V is an XQuery expression [30] called a *view query*.

Let $\mho$ be the domain of the update operation over the view. Let $u \in \mho$ 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 shows several examples of view updates expressed in the "XQuery" like language [29].

A taxonomy of the view update domain $\mho$ is shown in Fig. 6. First of all, a **valid view update** is an insert, delete or replace operation that satisfies all constraints in the view schema. Updates u_2, u_3 and u_4 in Fig. 4 are valid updates since they agree with all constraints of the view schema. Update u_1, however, is an invalid update as shown by Example 1.

Definition 1 *A relational update sequence U on a relational database D is a* **correct translation** *of a valid update u on the view V iff (i) $u(DEF_V(D)) = DEF_V(U(D))$ and (ii) if $u(DEF_V(D)) = DEF_V(D) \Rightarrow U(D) = D$.*

A correct translation means the "rectangle" rule shown in Fig. 7 holds. Intuitively, this implies that the translated relational updates exactly perform the view update, namely, without view side effects. In addition, if an update operation does not affect the view, then it should not affect the relational base either. This guarantees that any modification of the relational base is indeed done for the sake of the view. For instance, the update u_3 is not translatable since it tries to insert a review into a book that is not in the view. The second criterion is guaranteed if the translation is done by query composition, and hence generally can be achieved [1].

For a valid update, if a correct translation does not exist, u is **untranslatable**. If additional conditions are required to be satisfied for a correct translation to exist, then u is said to be **conditionally translatable**. As we will show in later sections, this additional condition includes for example translated update minimization and duplication consistency checking. Otherwise, u is called **unconditionally translatable** (Fig. 6)

[1] Note that some commercial databases (DB2) provides the view creator an option to update the data not visible through the view. Our solution can be easily adjusted to this scenario by relaxing this restriction

We now can define the problem of **XML view update translatability** as the problem of first classifying an update as either *valid* or *invalid*, then classifying a *valid* update as either *unconditionally, conditionally translatable* or *untranslatable*.

3 Annotated Schema Graph

U-Filter uses Annotated Schema Graph (ASG) to model the constraints from both the view query and the relational schema. ASG is then extensively used by schema level checking steps.

3.1 Classification of Constraints

The constraints, extracted from the relational schema or inferred from the view query [2] will help us to decide whether a given update is valid, and possibly even translatable. They are compiled once and reused thereafter for any future update checking specified over this same view.

Let us first examine a predicate p of the form $a\ \theta\ b$, where $\theta \in \{=, \neq, <, \leq, >, \geq\}$. We say that p is a *non-correlation predicate* if b is a literal (e.g., \$*book/year = 1990*). Otherwise, p is a *correlation predicate* (e.g., \$*book/pubid* = \$*publisher/pubid*).

We divide the constraints as *local* or *global*. Intuitively, constraints that affect only one tuple of a base relation or one view element are called *local constraints*. Otherwise, they are called *global constraints*. The *non-correlation predicates* in the view query form *local constraints*, while the other constraints in the view query, such as *correlation predicates*, *cardinality constraints* and *hierarchical structure* form *global constraints*.

In the relational schema, the *local constraints* include all the constraints specified over one relation, such as domain constraints, NOT NULL constraints and Check constraints for the domain. The *global constraints* include constraints specified over multiple relations in the relational schema, such as foreign key constraints.

3.2 Annotated Schema Graphs

For each view, two ASGs are generated to represent the constraints described in Section 3.1. The *base ASG* includes only the global constraints from the relational schema, while all the rest constraints are captured in the *view ASG*.

Fig. 8 depicts the view ASG for BookView in Fig. 3. The **view ASG**, denoted by $\mathcal{G}_V$, represents the hierarchical structure of the *XML view* [3]. Let $N_{\mathcal{G}_V}$ and $E_{\mathcal{G}_V}$ respectively denote the nodes and edges of $\mathcal{G}_V$.

$N_{\mathcal{G}_V}$ includes four kinds of nodes: root, internal, tag and leaf nodes. Node v_R in Fig. 8 is the *root node*. Without loss of generality, we assume there is always a root tag to enclose the FLWR expression in the view query. Otherwise, we would simply add a "dummy" root node. A *leaf node* v_L represents an atomic type. The parent of a leaf node, named *tag node* v_S, identifies a simple view element or an attribute. All the remaining nodes are *internal nodes* (v_C), each of which identifying a complex view element. In the rest of the paper, we also use v to indicate a node in $\mathcal{G}_V$ without specifying its type.

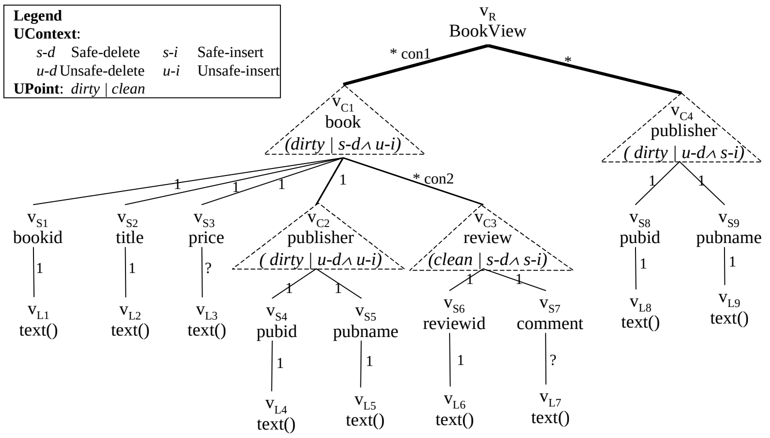
Each node is associated with its annotation set as shown in the *Node Annotation Table* in Fig. 8. The annotation of a leaf node includes $\{$*name, type, property, check*$\}$. A leaf node v_L has its corresponding relational attribute name $R.a$ as its *name* annotation. The *type* of v_L represents its domain constraints. The *property* of v_L captures $\{$*Not Null*$\}$ constraints, depending on the constraints on its relational attribute $R.a$. For instance, the property of node v_{L8} in Fig. 8 is marked as *Not Null*, since *publisher.pubid* is the key of the *publisher* relation. The *check* annotation of v_L represents the relational check constraints.

The annotation of v_R, v_C or v_S node includes the *name* field which corresponds to the tag name of the element it is modeling. Moreover, the v_R and v_C nodes include their **Update Context Binding** and **Update Point Binding** in their annotation. The *UCBinding(v_C)* extracts all the relations that have some influence on the existence of v_C in DEF_V. The *UPBinding(v_C)* includes all the relations referred in constructing v_C. For example, *UCBinding(v_{C1})* = $\{$*book,publisher*$\}$ since both relations *book* and *publisher* are used to decide the existence of *book* element (v_{C1}). *UPBinding(v_{C1})* = $\{$*book,publisher,review*$\}$ since (i) both *book* and *publisher* relations are used to construct the *book* element and (ii) *review* relation is used to construct *review* elements inside. Readers familiar with SilkRoute [25] can consider the UCBinding as all relations in the FROM clauses of the SQL statements generated for a node v, while the UPBinding as all relations in the FROM clauses of all its descendent nodes.

Given two nodes $v_1, v_2 \in N_{\mathcal{G}_V}$, the edge $e(v_1, v_2) \in E_{\mathcal{G}_V}$ represents that v_1 is a parent of v_2 in the view hierarchy. Each edge is labeled by its end node pair (v_1, v_2). Its annotation includes its cardinality type (inferred from the view query) and its condition (if any), extracted from the correlation predicate in the view query. The cardinality types are from the enumeration domain $\{1, ?, +, *\}$, representing $1 : 1$, $1 : \{0, 1\}$ (at most one), $1 : n+$ (at least one) and $1 : n$ (any) respectively. Note that the incoming edge of v_L is either 1 or ?, depending on whether it can be Null. We also define a function rel to extract the relations in D referenced by the view query DEF_V. For example, in Fig. 3(a), *rel(DEF_V)* = $\{$*publisher,book,review*$\}$. Note that *rel(DEF_V)=UPBinding(v_R)*.

[2] We do not propagate relational constraints (such as Key) to the XML as done by [9], we merely use them to filter out ill-behaved updates. However, work in [9] can be easily included to favor the checking procedure in general.

[3] Computing $\mathcal{G}_V$ is done similarly as in SilkRoute [25], for details please refer to [33].



Node Annotation Table

v_R: name = BookView
UCBinding = {}
UPBinding = {book,publisher,review}
	v_{L1}: name = book.bookid
	type = string
	property = {Not Null}

v_{C1}: name = book
UCBinding = {book,publisher}
UPBinding = {book,publisher,review}
	v_{L2}: name = book.title
	type = string
	property = {Not Null}

v_{C2}: name = publisher
UCBinding = {book,publisher}
UPBinding = {publisher}
	v_{L3}: name = book.price
	type = string
	property = {}
	check = {0.00<value<50.00}

v_{C3}: name = review
UCBinding = {book,publisher,review}
UPBinding = {review}
	v_{L4}: name = publisher.pubid
	type = string
	property = {Not Null}

v_{C4}: name = publisher
UCBinding = {publisher}
UPBinding = {publisher}
	v_{L5}: name = publisher.pubname
	type = string
	property = {Not Null}

v_{S1}: name = bookid
v_{S2}: name = title
v_{S3}: name = price
	v_{L6}: name = review.reviewid
	type = string
	property = {Not Null}

v_{S4}: name = pubid v_{S5}: name = pubname	v_{L7}: name = review.comment type = string

v_{S6}: name = reviewid
v_{S7}: name = comment
	v_{L8}: name = publisher.pubid
	type = string
	property = {Not Null}

v_{S8}: name = pubid
v_{S9}: name = pubname
	v_{L9}: name = publisher.pubname
	type = string
	property = {Not Null}

Edge Annotation Table

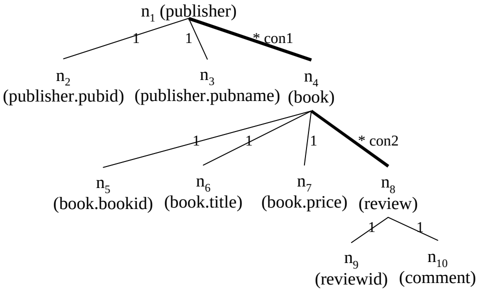
(v_R, v_{C1}): type = * , condition = {book.pubid = publisher.pubid}
(v_R, v_{C4}): type = *
(v_{C1}, v_{C3}): type = * , condition = {book.bookid = review.bookid}
(v_{C1}, v_{C2}): type = 1

Figure 8: View ASG of *BookView* in Fig. 3

The **Base ASG** $\mathcal{G}_D$ is a DAG that captures the hierarchical and cardinality constraints inferred from the key and foreign key constraints of the relational database. Let $N_{\mathcal{G}_D}$ denote the nodes and $E_{\mathcal{G}_D}$ denote the edges. $\mathcal{G}_D$ is computed as follows. For each leaf node in the view ASG, there exists a corresponding relational attribute. The union of all these relational attributes forms the leaf nodes of $\mathcal{G}_D$. Each leaf node is annotated by $\{$*name,property*$\}$. The *name* is given by its corresponding attribute name. The *property* (if any) captures $\{$*key*$\}$ constraints. For a leaf node n_l with name $R.a$, we introduce a node n corresponding to R and an edge (n, n_l). For any two nodes n_1, n_2 that correspond to relations R, S respectively, we introduce an edge (n_1, n_2), if there is a foreign key from S to R. The base ASG of BookView is shown in Fig. 9.

4 Update Validation

The *update validation* step identifies whether the given view update is *valid* according to the local constraints captured in the view ASG. Since the view schema has been extracted and represented as the view ASG in Section 3, we now only focus on the



Node Annotation Table

n_1: name = publisher	**n_6:** name = book.title
n_2: name = publisher.pubid property = {Key}	**n_7:** name = book.price
	n_8: name = review
n_3: name = publisher.pubname	**n_9:** name = review.reviewid property = {Key}
n_4: name = book	
n_5: name = book.bookid property = {Key}	**n_{10}:** name = review.comment

Edge Annotation Table

(**n_1**, **n_4**): type = *, condition = {book.pubid = publisher.pubid}
(**n_4**, **n_8**): type = *, condition = {book.bookid = review.bookid}

Figure 9: Base ASG of *BookView* in Fig. 3

u5
```
FOR $book IN  document("BookView.xml")/book
WHERE $book/price/text() > 50.00
UPDATE $book {
      DELETE $book/review }
```

u6
```
FOR $book IN document("BookView.xml")/book
UPDATE $book {
      DELETE $book/bookid/text() }
```

u7
```
FOR $root IN document("BookView.xml")
UPDATE $root {
    INSERT
        <book>
            <bookid>"98004"</bookid>
            <title>" Operating Systems "</title>
            <price> 20.00 </price>
        </book> }
```

u8
```
FOR $book IN  document("BookView.xml")/book
WHERE $book/price < 40.00
UPDATE $book {
      DELETE $book/review }
```

u9
```
FOR $root IN document("BookView.xml"),
      $book =$root/book
WHERE $book/price > 40.00
UPDATE $root {
      DELETE $book }
```

u10
```
FOR $book IN  document("BookView.xml")/book
WHERE $book/price > 40.00
UPDATE $book {
      DELETE $book/publisher }
```

u11
```
FOR $book IN document("BookView.xml")/book
WHERE $book/title/text() = "Programming in Unix"
UPDATE $book {
    DELETE $book/review}
```

u12
```
FOR $book IN document("BookView.xml")/book
WHERE $book/title/text() = "Data on the Web"
UPDATE $book {
    DELETE $book/review}
```

u13
```
FOR $book IN document("BookView.xml")/book
WHERE book/title/text() = "Data on the Web"
UPDATE $book {
  INSERT
    <review>
        <reviewid>001</reviewid>
        <comment>Easy read and useful.</comment>
    </review>}
```

Figure 10: View Update Example

question: what kind of validation must be considered for different update types (delete, insert) respectively [4]?

Delete. Two checks must be considered in the delete case. (i) Does the element to be deleted appear in the view? If an update operation does not affect the view, then it should not affect the relational base either (Definition 1). This means that the non-correlation predicate specified in the user update must "overlap" with the check constraints captured in the view ASG leaf node. For this, we examine the *check* annotation of the leaf node. For example, u_5 in Fig. 10 tries to delete all the reviews from the book that costs more than $50. However, the BookView in Fig. 3 only includes those books that cost less than $50. Thus u_5 is said to be invalid.

(ii) For deletes over a node in the view ASG, is that node deletable? Since a relational delete always removes one or more tuples, while an XML delete is more flexible, and could delete either just a single value or even a complete subtree. In general, deletion of a node with the incoming edge as "1" is invalid. For example, u_6 in Fig. 10 is invalid since leaf node v_{L1} is required to be *Not Null* in Fig. 8.

Insert. Assume that the insert happens on the schema node v in the view ASG. We first examine whether the node to be inserted conforms to the hierarchy specified in the view ASG, by examining the name annotation of the node and the type annotation of the edges. For example, u_7 in Fig. 10 is invalid since the type annotation of the edge (v_{C1}, v_{C2}) in Fig. 8 is "1", that is, each book must have exactly one publisher.

Second, we consider whether the values inside the element to be inserted conform to the constraints captured in the view ASG. (i) The leaf node value must be in the domain defined by its *type* annotation. (ii) The leaf node value must satisfy the *check* annotation. (iii) The leaf node value cannot be empty if the *property* annotation includes *Not Null*. For instance, u_1 in Fig. 4 is invalid since the node annotation table in Fig. 8 indicates that (i) the $title$ cannot be *NULL* according to the property of the node v_{L2} and (ii) the $price$ should be a positive number based on the check annotation of the node v_{L3}.

After Step 1, the view updates which directly conflict with any of the local constraints are rejected instantly. The rest are passed to the next step for view side effect checking.

5 Schema-driven Translatability Reasoning

Using the global constraints captured in both view ASG and base ASG, this step classifies a valid view update as *untranslatable*, *conditionally translatable* or *unconditionally translatable*.

For example, u_8 in Fig. 10 is unconditionally translatable, and a correct translation is to delete $review.t_1$ and $review.t_2$ from $review$ table. The update u_9 in Fig. 10 is said to be conditionally translatable. The direct translation *U* = $\{$*(delete from book where rowid* = t_3*),(delete from publisher where rowid* = t_1*)*$\}$ is not a correct translation, since $publisher.t_1$ is still referenced by the first book element in the view. Deleting it will cause a view side effect. Thus, in order for this update to be translatable, an additional condition is associated with this update, namely "Apply update minimization in translation". Such kind of minimization is studied in [2, 21, 22]. The update u_{10} in Fig. 10 is untranslatable. The direct translation is *U* = $\{$*delete from publisher where rowid* = $t_1\}$. Note that there is a foreign key from the *book* relation to the *publisher* relation. Therefore this deletion will cause the *book* to disappear also.

Our earlier work [32, 34] proposes a *clean extended source theory* as criteria for determining whether a given translation mapping is correct. A clean extended source represents a correct translation for the given view update, which achieves the desired delete operation without causing any view side effect. An update translation is correct if and only if it deletes or inserts a clean extended source of the view element. Based on this theoretical foundation, we now propose a concrete **schema-driven update**

[4] Currently we consider replace as a deletion followed by an insertion.

translatability reasoning (STAR) algorithm. It utilizes a static *STAR marking* procedure and a dynamic *STAR checking* procedure to decide the translatability of a valid update from the update validation step. The correctness proof of our STAR algorithm can be found in [33].

In this paper, we consider only the internal nodes v_C in our schema level update translatability checking. These nodes are marked by a dashed line in Fig. 8. The treatment of other nodes is either trivial or similar to that of v_C nodes. For instance, deleting the root node v_R is always translatable. Similarly any valid update of a v_L node will be translatable. Updates of v_S nodes are handled similar to an update over a v_C node.

5.1 STAR Marking Procedure

We use a STAR marking procedure to encode each node in $\mathcal{G}_V$ at compile time by its *update point type* and *update context type*, labeled as *(UPoint|UContext)*. This mark is then used to determine the translatability of updates specified on the nodes.

5.1.1 Update Context Type

The **update context type** (UContext) of a node in $\mathcal{G}_V$ determines whether a view side effect might arise when deleting or inserting an instance of this node. A node is said to be *safe-delete* if for the operation of deleting any of its instance, there exists a translation which is guaranteed to not cause any view side effect. Otherwise it is said to be *unsafe-delete*. Similarly, a node is said to be *safe-insert* if for the operation of inserting a new instance, there exists a translation which will not cause any view side effect. Otherwise, it is said to be *unsafe-insert*.

The intuition for update context type is that duplication in the view is the major cause of view side effects. Duplication appears in different forms: (i) two instances of the same view ASG node might be duplicated, and thus map to the same relational data and (ii) two instances of different view ASG nodes could also include duplicate sub-elements.

The following rules are used to determine the UContext of a node using the view ASG $\mathcal{G}_V$. Rule 1 identifies the unsafe internal nodes caused by duplication (i). Rules 2 and 3 identify the unsafe v_C nodes caused by duplication (ii).

Recall the *UCBinding* defined in Section 3. Given an internal node $v_C \in N_{\mathcal{G}_V}$ and its parent node $v \in N_{\mathcal{G}_V}$. We define the *Current Relations* of v_C as *CR(v_C) = UCBinding(v_C) $-$ UCBinding(v)*. We say a Join condition $R_i.a = R_j.b$ on an edge $e = (v_{C1}, v_{C2})$ is a **proper Join** if (i) $R_j \in CR(v_{C2})$ and (ii) $R_i.a$ is a unique identifier of $R_i \in CR(v_{C1})$. A proper Join ensures no duplicates are introduced for v_{C2} by this Join.

Rule 1: *Let $e = (v_{C1}, v_{C2})$ be an edge in $\mathcal{G}_V$ with type "*". $UContext$ of any node in the subtree rooted at v_{C2} is unsafe-delete and unsafe-insert if e is not associated with a proper Join condition (as described above).*

Rule 1 can be used to identify "missing" Join conditions. For example, assume that we removed the second WHERE clause in BookView in Fig. 3. That is, the edge (v_{C1}, v_{C3}) in Fig. 8 were not annotated with any condition. All the nodes in the subtree of v_{C3} are unsafe now since the whole review table is now nested inside of each individual book, even if unrelated.

On the other hand, this rule can also identify "improper" Join conditions (a Join condition causing duplicates). As an example, assume the second WHERE clause in BookView in Fig. 3 is replaced by a correlated predicate "$\$book/title = \$review/comment$". Then the edge (v_{C1}, v_{C3}) in Fig. 8 is annotated with a Join condition $book.title = review.comment$. Since neither $book.title$ nor $review.comment$ is UNIQUE, we will then mark all the nodes in the subtree of v_{C3} as *unsafe-delete* and *unsafe-insert*.

Now assume all * edges between the internal nodes of $\mathcal{G}_V$ are annotated with a proper Join condition. Is it possible for duplication to exist? The answer is yes.

Rule 2 below is used to identify unsafe internal nodes for the delete operation, which could cause other nodes, which are not its descendants, to disappear. As an example, again consider v_{C2} in Fig. 8. Rule 2 below will mark v_{C2} node as *unsafe-delete*, because it affects the appearance of the whole *book* node. Given a relation R, we define $\mathbf{extend(R)} \subseteq rel(DEF_V)$ as a set of relations that refer to R through foreign key constraint(s).

Rule 2: $UContext(v_C)$ *= unsafe-delete if* $\neg\exists R \in CR(v_C)$ *such that* $\forall v'_C \in N_{\mathcal{G}_V}$ *being a non-descendant node of* v_C*,* $extend(R) \cap UCBinding(v'_C) = \emptyset$.

The *UCBinding* difference between the node to be deleted and its parent, denoted by $CR(v_C)$ in Rule 2, indicates the smallest search space for a *clean extended source* [32]. If none of the relations in $CR(v_C)$ is a clean extended source, then deleting an instance of node v_C will cause a view side effect.

As an example, consider v_{C4} in Fig. 8. We have *UCBinding(v_{C4})=*{*publisher*} and *UCBinding(v_R)=*{}. Thus $CR(v_{C4})$=*{publisher}*. For *R = publisher*, $extend(R)$ = {*publisher,book,review*}. Note that *UCBinding(v_{C1})* = {*book,publisher*} and *extend(R)* $\cap$

UCBinding(v_{C1}) $\neq \emptyset$. Deleting an instance of v_{C_4} will potentially cause a view side effect on v_{C1} – the book might also disappear. Thus $UContext(v_{C4})$ = *unsafe-delete*, as marked in Fig. 8.

Similarly, Rule 3 below is used to identify unsafe internal nodes for an insert operation, which could cause the appearance of other nodes, which are not its descendants, as view side effect. For example, Rule 3 below will mark v_{C2} node in Fig. 8 as *unsafe-insert*, because it will cause the appearance of v_{C4} as view side effect. The *UPBinding()* defined in Section 3 is utilized to identify this case.

Rule 3: $UContext(v_C)$=*unsafe-insert if* $\exists\ v'_C \in \mathcal{G}_V$ *that is a non-descendant node of* v_C *such that (i)* $UPBinding(v_C) \cap CR(v'_C) \neq \emptyset$ *and (ii)* $UContext(v'_C)$=*unsafe-delete hold.*

Intuitively, if (i) does not hold in Rule 3, then inserting an instance of v_C will insert to those relations only referred to by v_C itself. It will not cause view side effects on any other schema nodes. However, if (i) holds, then inserting v_C indicates the potential appearance of an instance of v'_C as side effect since v'_C "shares" some common relation with v_C. But if (ii) does not hold, that is the update context type of v'_C is safe-delete, then at least we can alway eliminate the potential side effect by deleting the clean source of v'_C. Thus only (i) and (ii) hold together denote the appearance of an unsafe-insert situation.

Consider v_{C1} in Fig. 8. We have *UPBinding(v_{C1}) =* $\{$*book, publisher, review*$\}$ and $CR(v_{C_4})$ = $\{$*publisher*$\}$. Thus (i) holds. Since *UContext(v_{C4}) = unsafe-delete*, thus we say that *UContext(v_{C1}) = unsafe-insert*. Inserting a book might cause the insertion of an instance of v_{C4} as a view side effect, if the publisher does not exist in the *publisher* relation before.

The following lemmas propose the correctness of above rules. For proofs please refer to [33].

Lemma 1 *Let* $v \in \mathcal{G}_V$ *and* $e \in I(v)$, *where* $I(v)$ *denotes the set of instances in* V *of* v. *There exists a correct translation for deleting* e *that does not cause any view side-effects if* $UContext(v)$ *= safe-delete.*

Lemma 2 *Let* $v \in \mathcal{G}_V$ *and* e *be a new instance of* v. *There exists a correct translation for inserting* e, *that does not cause any view side-effects if* $UContext(v)$ *= safe-insert.*

5.1.2 Update Point Type

The *update context type* of a schema node $v \in \mathcal{G}_V$ introduced above determines whether there exists at least one *clean extended source* [32]. If the answer is yes ($UContext(v)$ *= safe-delete or safe-insert*), then the next question is how to find it? As suggested by [10, 11], the "where-provenance" (refers to the location(s) in the source databases from which the data was extracted) is a good candidate to start the search for a clean extended source in the relational context. Below, we enhance the notion of the where-provenance by the *mapping closure* concept to also consider the effect of constraints from the relational database and XQuery's nested query syntax. We use the *update point type* (UPoint) to indicate whether the mapping closure is a clean extended source of v.

Closure. We use the concept of closure in $\mathcal{G}_V$ and $\mathcal{G}_D$ to indicate the effect of an update on the view and on the relational database respectively. The **closure** of a node v in $\mathcal{G}_V$, denoted by v^+, is defined as follows. (1) $v_L^+ = \{v_L\}$. (2) Otherwise, v^+ is the union of its children's closures grouped by their hierarchical relationship and marked by their cardinality. For simplicity, in the closure the cardinality of $^+$ and * are both represented as *, and the cardinality of 1 and ? are omitted. For example, in Fig. 8, $v_{L1}^+ = \{v_{L1}\}$ while $v_{C2}^+ = \{v_{L4}, v_{L5}\}$ and v_{C1}^+=$\{v_{L1}, v_{L2}, v_{L3}, v_{L4}, v_{L5}, (v_{L6}, v_{L7})^{*con_2}\}$.

The **closure** of an internal node in $\mathcal{G}_D$ is defined as the union of its children leaf nodes and the closure of its non-leaf direct children nodes. For example, n_1^+ = $\{n_2, n_3, (n_4^+)^{*con1}\}$=$\{n_2, n_3, (n_5, n_6, n_7, (n_8^+)^{*con2})^{*con1}\}$=$\{n_2, n_3, (n_5, n_6, n_7, (n_9, n_{10})^{*con2})^{*con1}\}$. The closure of a leaf node is the same as the closure of its parent node. For instance, in Fig. 9, $(n_9)^+$ = $(n_8)^+$ = $\{n_9, n_{10}\}$. Note that this closure definition in $\mathcal{G}_D$ is based on the pre-selected update policy: *same type* and *delete cascade*. When a different update policy is used, the definition has to be adjusted accordingly. However, the policy used affects only the closure definitions of the base ASG, while the remaining steps for translatability checking remain the same [33].

Given two closures C_1 and C_2, we define $C_1 \subseteq C_2$, if C_1 appears in C_2. In Fig. 9, n_8^+ = $\{n_9$,$n_{10}\}$ and n_4^+ = $\{n_5$,n_6,n_7,$(n_9, n_{10})^{*con_2}\}$, thus $n_8^+ \subseteq n_4^+$. Two closures C_1 and C_2 are **equal**, denoted by $C_1 \equiv C_2$, if $C_1 \subseteq C_2$ and $C_1 \supseteq C_2$. In Fig. 9, we have $n_5^+ \equiv n_6^+$.

Moreover, we define the closure of a set of nodes $N \in N_{\mathcal{G}_D}$, denoted by N^+, as N^+ = $\bigsqcup_{(n_i \in N)} n_i^+$, where $\bigsqcup$ is a "Union-like" operation that combines the nodes but eliminates duplicates. That is, $\forall n_k, n_j \in N$, if $n_k^+ \subseteq n_j^+$, N^+ = $\bigsqcup_{(n_i \in N, i \neq k)} n_i^+$. For instance, in Fig. 9, $(n_4, n_8)^+$ = $(n_4)^+ \bigsqcup (n_8)^+$ = $(n_4)^+$ = $\{n_5$,n_6,n_7,$(n_9, n_{10})^{*con_2}\}$.

Mapping Closure. Intuitively, the relationship between the closure of an internal node v_C in $\mathcal{G}_V$, denoted by C_V, and its mapping closure C_D defined below, answers the following question. If an instance of v_C is deleted or inserted, what will be affected in the relational database?

The *mapping closure* of v_C is defined as follows. First we compute $C_V = v_C^+$ in $\mathcal{G}_V$. Let T*=Distinct(getNodes(*C_V*))*, where getNodes() is a function to extract all the nodes from a given closure, while Distinct() removes duplicates by node identification. For each node $v_{Li} \in T$, we define its *mapping leaf node* n_i in $\mathcal{G}_D$ to be the one with the same *name* in its annotation. Let N denote the set of mapping nodes from T. Let $C_D = N^+$ in $\mathcal{G}_D$. We call C_D the *mapping closure* of v_C. For example, v_{C2}^+ = $\{v_{L4}$,$v_{L5}\}$ and T = $\{v_{L4}$,$v_{L5}\}$. Then the mapping nodes in $\mathcal{G}_V$ is N=$\{n_2$,$n_3\}$ and N^+ = $\{n_2$,n_3,$(n_5$,n_6,n_7,$(n_9, n_{10})^{*con2})^{*con1}\}$. This is the mapping closure C_D we are looking for.

Definition 2 $UPoint(v_C)$ = **clean** *if* $C_V \equiv C_D$. *Otherwise,* $UPoint(v_C)$ = **dirty**.

It is only necessary to consider the update point type when the update context type is safe. For example, in Fig. 8, v_{C1} is marked as *(dirty*$|$*safe-delete,unsafe-insert)*. In this case, inserting a book might cause a view side effect as we explained before. Deleting a book is safe, but since the UPoint mark is dirty, we have to examine whether its publisher is referenced by other books.

Algorithm 1 Mark $\mathcal{G}_V$ with (UPoint$|$UContext) pair

```
/*Mark (UPoint|UContext) for $\mathcal{G}_V$ */
PROCEDURE markViewASG ($\mathcal{G}_V$, $\mathcal{G}_D$)
computeClosure($\mathcal{G}_V$, $\mathcal{G}_D$)
markUContext($\mathcal{G}_V$, $\mathcal{G}_D$)
markUPoint($\mathcal{G}_V$, $\mathcal{G}_D$)

/* Mark $UContext(v_C)$ as safe or unsafe for deletion and insertion*/
PROCEDURE markUContext ($\mathcal{G}_V$, $\mathcal{G}_D$)
/* First mark unsafe nodes*/
Initiate rules set S for update context checking
Add rules 1 to 3 into S in order
while S has more rules to be evaluated do
  Get the next rule r from S
  evaluateRule(r,$\mathcal{G}_V$, $\mathcal{G}_D$)
end while
/* Mark the rest of them as safe */
while $N_{\mathcal{G}_V}$ has more unmarked $v_C$ nodes do
  Get the next node $n \in N_{\mathcal{G}_V}$
  $UContext(v_C)$ = safe-delete$\wedge$safe-insert
end while

/* Mark $UPoint(v_C)$ as clean or dirty*/
PROCEDURE markUPoint($\mathcal{G}_V$, $\mathcal{G}_D$)
while $N_{\mathcal{G}_V}$ has more unmarked nodes do
  Get the next node $v_C \in N_{\mathcal{G}_V}$
  $C_V$ = getClosure( $v_C$, $\mathcal{G}_V$)
  $C_D$ = getClosure( Distinct(getNodes($C_V$)), $\mathcal{G}_D$)
  if $C_V \equiv C_D$ then
    $UPoint(v_C) = clean$
  else
    $UPoint(v_C) = dirty$
  end if
end while
```

5.2 STAR Checking Procedure

Once the ASGs are analyzed and marked by *(UPoint*$|$*UContext)* labels using Algorithm 1, the *STAR checking procedure* is used to decide the update translatability and additional conditions required (if any). Observations 1 and 2 serve this purpose.

Observation 1 *A deletion on an* **unsafe-delete** *node is un-translatable. A deletion on a* **(clean** $|$ **safe-delete)** *node is unconditionally translatable. A deletion on a* **(dirty** $|$ **safe-delete)** *node is conditionally translatable. The condition required is translation minimization. This refers back to the source-side-effect-minimization defined in [12], and studied in [2, 21, 22]. This condition guarantees that the translated update sequence avoids the view side effect from duplication.*

Observation 2 *A insertion on an* **unsafe-insert** *node is un-translatable. A insertion on a* **(clean** $|$ **safe-insert)** *node is unconditionally translatable. A insertion on a* **(dirty** $|$ **safe-insert)** *node is conditionally translatable. The required condition is duplication consistency. That is, the duplicate parts inside the element to be inserted should have consistent values.*

For example, u_8 in Fig. 10 is unconditionally translatable since v_{C3} in Fig. 8 is a *(clean* $|$ *safe-delete)* node. u_{10} is untranslatable since $UContext(v_{C2})$ = *unsafe-delete* in Fig. 8. The update u_9 is conditionally translatable by Observation 1. While deleting the book, we will not delete its corresponding publisher, if another book references this publisher.

6 Data-driven Translatability Checking

As motivated by Example 3 in Section 1, any update operation which passes through all schema level checks (Steps 1 and 2) might still be untranslatable. This can only be detected by examining the actual data, as described below.

6.1 Data-driven Update Context Check

The update context check focuses on the *context* of the update operation. It aims to answer the question whether the view element that the user update is inserting into or deleting from exists in the view content. For instance, in Example 3 (u_3 in Fig. 4), the *book* into which the *review* is to be inserted is not in the view. Thus u_3 is not translatable. Similarly, u_{11} in Fig. 10 will be rejected since it is trying to delete the reviews of the book "Programming in Unix", which does not appear in the view.

We can address this by composing the view query with the user update query into a *probe query* as done by most XML data management systems which support queries over views [13, 25]. This *probe query* is then evaluated over the relational engine. For u_3, the probe query PQ_1 will be:

PQ_1: SELECT bookid FROM publisher,book,review
WHERE book.title ="Programming in Unix" AND book.price $<$ 50.00
AND book.year$>$1990 AND book.pubid $=$ publisher.pubid

The result of this query can be used for two different purposes. First, if the result set is empty, this means the qualified *book* does not exist in the view. Thus the given insert operation is not translatable, and hence rejected. For example, the probe query used by u_{11}, which is the same as PQ_1, returns empty result. u_{11} will also be rejected.

Second, in some cases the results of the probe query are required by the translated SQL update statements. For example, u_{13} in Fig. 10 is similar to u_3 in Fig. 4, except that the *review* is now inserted into the book named "Data on the Web". The result of the probe query PQ_2 now includes one qualified book. Further, the *bookid* from PQ_2 will be used in the translated SQL statement U_1.

PQ_2: SELECT bookid FROM publisher,book,review
WHERE book.title ="Data on the Web" AND book.price $<$ 50.00
AND book.year$>$1990 AND book.pubid $=$ publisher.pubid

U_1 = {INSERT INTO review VALUES "98003", "001", "easy read and useful"}

As we will see later, the results of the probe queries can also be materialized and re-used to generate the full insert tuple or to eliminate redundant joins.

6.2 Data-driven Update Point Check

The update u_4 in Fig. 4 is determined to be not translatable. The reason is that a *book* with the key *(bookid,pubid) =(98001,A01)* already exists in the *book* relation. A data conflict thus exists. However, this data conflict is different from that described in Section 6.1. That is, the data conflict exists in the updated data itself (update point) instead of its context. Several approaches can be used to solve this problem.

6.2.1 Internal Approach

As proposed by [7, 8], the XML view can be mapped into a set of relational views. The update over the XML view can then be mapped into an update over this set of relational views. We thus would convert the XML view update problem into a relational view update problem. For example, Fig. 11 shows the mapping relational view of the *BookView* in Fig. 3. The update u_{13} on BookView will now be translated into U_V on *RelationalBookView*.

```
CREATE VIEW RelationalBookView AS
   SELECT p.pubid, p.pubname, b.bookid, b.title, b.price, r.reviewid, r.comment
   FROM ( Publisher AS p LEFT JOIN ( Book AS b LEFT JOIN Review AS r
               ON b.bookid = r.bookid ) ON p.pubid = b.pubid );
```

RelationalBookView

pubid	pubname	bookid	title	price	reviewid	comment
A01	McGraw-Hill Inc.	98001	TCP/IP Illustrated	37.00	001	A good book on network.
A01	McGraw-Hill Inc.	98001	TCP/IP Illustrated	37.00	002	Useful for advanced user.
A01	McGraw-Hill Inc.	98003	Data on the Web	48.00	null	null

Figure 11: The Mapping Relational View of BookView in Fig. 3

However, this approach has several shortcomings. First of all, this approach is rather limited since many current commercial relational database systems such as [24] support update operations over SelectProject-views, but are limited on supporting updates over Join-views.

Second, for performance reasons, this approach would cause unnecessarily expensive data queries. The update u_{13} only specified *(title, reviewid, comment)*. Thus we need to find the *bookid*. However, in this inside approach, the translated relational view update U_V above also has to find *(pubid,pubname,price)*. The latter is not really needed. Our experimental studies in Section 7 also illustrate this inefficiency.

6.2.2 External Approach

To avoid the shortcomings of the inside approach, we now introduce other more practical approaches to handle the *update decomposition* outside the relational engine. Here each resulting SQL update statement from the update translation engine will be specified over only a single table. Two alternative strategies can be useful, namely *hybrid* strategy or *outside* strategy.

U_V = {INSERT INTO RelationalBookView
(pubid,pubname,bookid,title,price,reviewid,comment)
VALUES (“A01”, “McGraw-Hill Inc”, “98003”,
“Data on the Web”, 48.00, “001”,“easy read and useful”)}

In the **hybrid strategy**, checking data conflicts is done by the relational engine, namely, the view update is decomposed and translated into a sequence of SQL updates without any data conflict checking. This update sequence is then fed into the relational engine, and we wait for its error or success response.

As an example, let us consider both insert and delete cases. In the insert case, the update u_4 in Fig. 4 maps into U_2 below. The relational engine executes U_2 and generates an error message since this insert conflicts with the Key constraint. In the delete case, u_{12} in Fig. 10 is translated to U_3 below. Note that U_3 accesses the table TAB_book, which is the materialized view from PQ_2 in Section 6.1. The relational engine executes U_3 and generates a warning message that zero tuples are deleted.

U_2 = {INSERT INTO book VALUES “98001”,“Operating Systems”,“A01”,20.00,1994}
U_3 = {DELETE FROM review
WHERE review.bookid IN SELECT bookid FROM TAB_book}

Alternatively, the **outside strategy** issues a probe query to check whether a data conflict exists for each of the relations we will insert into or delete from. For example, u_4 in Fig. 4 can be checked using the probe query PQ_3. Since its result is not empty, we conclude that there is a data conflict. u_4 is not translatable. Update u_{12} in Fig. 10 can be checked using the probe query PQ_4. Since the result set is empty, we conclude that the tuple to be deleted does not exist.

PQ_3 = { SELECT bookid FROM book
WHERE book.bookid = “98001” AND book.pubid =“A01”}
PQ_4 = {SELECT ROWID FROM review
WHERE review.bookid IN SELECT bookid FROM TAB_book}

We notice that the probe query used in the outside approach is very similar with the update query used in the hybrid approach. A natural question is if it is worthwhile to probe before update translation? The answer to this question depends on several factors, such as the shape of the view and the indices of the relational databases, as we illustrate in our experimental studies.

7 Evaluation of U-Filter

7.1 Views Handled by U-Filter

The view ASG used in our solution has the same limitations as the view forest from SilkRoute [19]. ASG also does not express if/then/else expressions; order functions, user-defined and aggregate functions, such as max(), count(), etc. We conduct an evaluation on the expressiveness of our view ASG model for W3C use cases. The evaluation result is shown in Figure 12.

In [12], the authors study the complexity of the update translatability problem in the case of deletion over relational SPJU views. They show that this problem is poly-time solvable with respect to the size of the database for SPU and SJ views, whereas it is NP-hard for PJ and JU views. Note that Project here implicitly eliminates the duplicates. since we restrict the view query handled by our ASGs, our views are actually a combination of SPJ views (in XML format), where we do not consider distinct operations in Project. Our STAR marking procedure is a schema-level check that runs in poly-time in the size of the view query. The STAR checking procedure takes only a hash operation time. Step 3 uses SQL engine, and runs in poly-time over the database size.

7.2 Performance Evaluation

Below we evaluate the performance of Step2 and Step3. The test system used is a dual Intel(R) PentiumIII(TM) 1GHz processor, 1G memory, running SuSe Linux and Oracle 10g. The relational database is built using TPC-H benchmark [1].

Performance of STAR Algorithm. Consider an XML view $V_{success}$ where the five relations (REGION, NATION, CUSTOMER, ORDER, LINEITEM) are nested following the key and foreign key constraints. Updates over any internal node of this view are unconditionally translatable. As shown by Fig. 13, even for a tiny database (1M), the STARChecking time is almost negligible in a successful execution.

Now consider another view V_{fail}, where the five relations are first joined linearly as above, then the relation to be updated (e.g.,REGION) is published again under the root tag. According to our STAR algorithm, deleting a region element from the view is not translatable, thus should be rejected. If the STAR checking procedure is not utilized, the system would submit the update. After the side effect has been identified, the transaction has to rollback to undo all the changes. Fig. 14 shows that this undo procedure is very expensive. On the other hand, our STAR checking algorithm can identify it early and thus remains very efficient, even if the DBsize increases.

For both views, our experiments also show that the STAR marking procedure stays cheap. The time for $V_{success}$ is 0.12s, while marking V_{fail} takes 0.15s.

View Query	Included	Reason
XMP-{Q1-Q3, Q5, Q7-Q9, Q11, Q12}	√	
XMP-{Q4,Q10}	×	Distinct()
XMP-Q6	×	Count()
TREE-Q1	√	
TREE-Q2	√	
TREE-{Q3,Q4,Q5,Q6}	×	Count()
R-{Q1,Q3, Q4,Q16,Q17}	√	

R-{Q2,Q5,Q6-Q15}	×	max(),avg()
		count()

R-Q18	×	Distinct()

Figure 12: Evaluation of W3C User Case

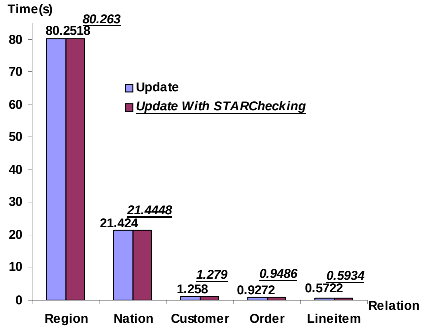


Figure 13: Performance of a translatable view update (DBsize=1Mb)

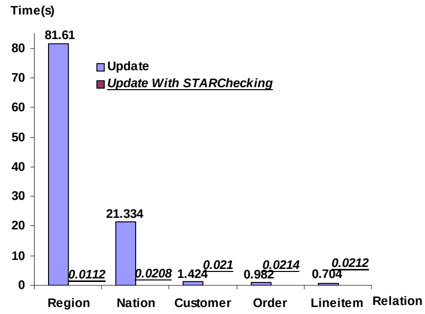


Figure 14: Performance of a untranslatable view update (DBsize=1Mb)

About Data-driven Translatability Checking. We consider two views, namely, V_{linear} where the five relations are joined linearly or V_{bush} where they are joined "evenly". First, let's assume that the relational database supports updates over relational Join-views. We compare the performance of the internal approach and the external approach using V_{linear}. Fig. 15 shows that the internal approach is more expensive than the external approach (hybrid strategy) for inserting a new *lineitem* into the view. This is because the internal approach issues a probe query to retrieve "all" attributes from "all" other four relations in order to construct a complete relational view tuple, which is inserted into the corresponding relational view (as explained in Section 6.2.1). While the External approach only retrieves the necessary information to form a *lineitem* tuple, that is the $L\_ORDERKEY$.

The performance comparison between two strategies of the external approach is shown in Figures 16 and 17. Fig. 16 shows that the hybrid strategy performs better in the V_{bush} case. The reason is that both strategies include similar amount of join operations, but the hybrid strategy generates simplified updates, which does not materialize the intermediate result. Further, the relatively "big" gap between the hybrid and outside strategies is due to the indices defined. Oracle builds indices over the primary keys and foreign keys, which is used by the Join condition in the hybrid strategy. The outside strategy, however, performs joins over the materialized view, where indices do not exist.

Fig. 17 shows that the outside strategy performs better in different failed cases of V_{linear}. *Fail2* means no qualified tuples in LINEITEM exist, but tuples in CUSTOMER and ORDER are deleted. While in *Fail1* there are no deletes over all three tables. As we can see, for *Fail2*, in the hybrid strategy, the delete queries over LINEITEM will still be executed and they return a warning message "zero tuples deleted". In the outside strategy, the probe query will identify it and the delete statement is not issued on LINEITEM. This is another advantage of the outside strategy since the failed case is detected early.

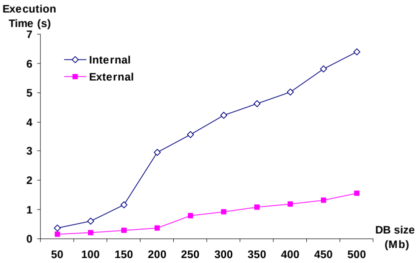


Figure 15: Internal vs. External for u over V_{linear}

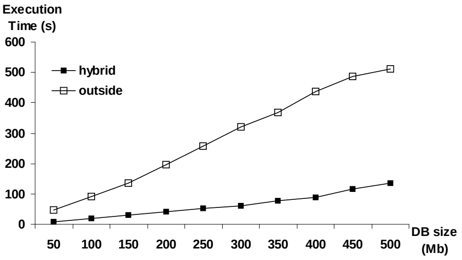


Figure 16: Outside vs. Hybrid for u over V_{bush}

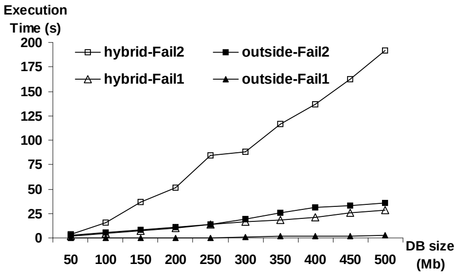


Figure 17: Outside vs. Hybrid for u over V_{linear} in failed cases

7.3 Practical Application of U-Filter Approach

To examine the practicality of our approach, we studied the Protein Sequence Database (PSD) from [27]. From typical user studies over this domain [27] gained by discussion with biologist (Ryder, Elizabeth F) at WPI, we observed the following: (i) The *well-nested view* assumed by [7, 8], where the nesting "follows" the key and foreign key constraints, is not often the case in this domain. (ii) The delete SET NULL policy is typically used in this domain as opposed to the delete cascade policy. U-Filter approach hence provides a practical solution to this domain, because it supports even non-well-nested views as well as flexible update policies.

8 Related Work

[2, 21, 22] study the view update translation mechanism for SPJ queries on relations that are in BCNF. These works have been further extended for object-based views in [5]. Commercial database systems, such as Oracle [4], DB2 [15] and SQL-Server [28], also provide XML support. [29] presents an XQuery update grammar. Assuming that the update is indeed translatable and has in fact already been translated into updates over a relational database. [29] also studies the performance of executing the translated updates by using relational techniques, such as triggers or indices. Our work addresses a different aspect of the view update problem, namely, *view update translatability* instead of *update translation strategy*.

An abstract formulation of the update translatability problem is given by the *view complementary theory* in [3, 16]. It uses the invariance of the complement of a view, namely *database side-effect free*, to decide the translatability of a given update. However, by requiring the *database side-effect free* property, the complementary theory is too restrictive to be practical. In [18], the authors relax the criteria for a correct translation as only requiring *view side-effect free*. Based on the notion of a *clean source*, it presents an approach in the relational context for determining the existence of update translations by performing a syntax analysis of the view definition.

Our earlier work [31] studies the update translatability of XML views over the relational database in the "round-trip" case, which is characterized by a pair of reversible lossless mappings for (i) loading the XML documents into the relational database, and (ii) extracting an XML view identical to the original XML document back out of it. We prove that the view updates in this case are always translatable according to the *view complementary theory* [3, 16]. Recent works [7, 8] study the update over *well-nested* XML views. They assume joins are through keys and foreign keys, and nesting is controlled to agree with the integrity constraints and to avoid duplication. As our work on updates shows, an update over such a view is thus always translatable. [23] develops a theory within the framework of the ER approach to characterize the conditions under which mappings exist. It is further extended in [14] to guide the design of valid XML views. Valid views based on this design approach are a proper subset of general XML views studied in this paper. [14] avoids the duplication from both joins as well as 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 when no particular restrictions have been placed on the defined view for the update translatability. That is, in general, conflicts are possible and a view cannot always be guaranteed to be revert-able [31], well-nested [7, 8] or valid [14] (as assumed by these prior works).

In [32] we first extend [18] into a *clean-extended source theory* for XML views to serve as a criteria of determining whether a given translation is correct. [34] focuses on identifying the factors deciding the translatability of deletions over XML views (part of Step 2). Our work in this paper now provides a general framework consisting of three steps based on the factors identified previously. Further, as part of our data-level check we are able to analyze the performance of existing work [8]; we are able to suggest alternative approaches that can work with existing DBMS without imposing additional requirements, and that yield better performance.

Recent works [10, 11, 17] indicate a loose connection between *data provenance* [10, 11] or *lineage* [17] and the view update problem. The distinction between "why provenance" and "where provenance" is used to guide the view update process to find an appropriate update translation. Their work has several similarities with ours, e.g., try to find the data trace (provenance) at the query syntax level. However, we utilize this data trace or provenance for a different purpose. The question that [10, 11] tries to answer is: given two equivalent queries that are rewritings of each other, when are the provenances guaranteed to be identical? Instead, we use the provenance to find a correct translation, if one exists, for a given update query.

9 Conclusions

In this paper, we have proposed a lightweight framework *U-Filter* to address the XML view update translatability problem. A three-step translatability checking process is used to guarantee that only translatable updates are fed into the actual translation system to obtain the corresponding SQL statements.

Our solution is *practical* since it does not require any additional update capability from the relational database. It can be applied by any existing view update system for analyzing the translatability of a given update before its translation is attempted. Our solution is also *efficient* since we perform schema-level (thus very inexpensive) checks first, while utilizing the data-level checking only at the last step. Even when data has to be accessed, we issue probe queries whose results can be reused for later update translation.

In the future, we would like to study how our solution can be adapted to XML views published over native XML documents, in particular XML-specific issues, such as order handling.

References

[1] TPC Benchmark H (TPC-H). http://www.tpc.org/information/benchmarks.asp.

[2] A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.

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

[4] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - The XML Enabled Data Management System. In *ICDE*, pages 561–568, 2000.

[5] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *SIGMOD*, pages 248–257, 1991.

[6] M. Benedikt, C. Y. Chan, W. Fan, and R. Rastogi. DTD-Directed Publishing with Attribute Translation Grammars. In *VLDB*, pages 838–849, 2002.

[7] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. On the Updatability of XML Views over Relational Databases. In *WEBDB*, pages 31–36, 2003.

[8] 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.

[9] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Reasoning about keys for xml. In *Information System*, 2003.

[10] P. Buneman, S. Khanna, and W.-C. Tan. Data provenance: Some basic issues. In *Foundations of Software Technology and Theoretical Computer Science*, 2000.

[11] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[12] P. Buneman, S. Khanna, and W. C. Tan. On Propagation of Deletions and Annotations Through Views. In *PODS*, pages 150–158, 2002.

[13] 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.

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

[15] J. M. Cheng and J. Xu. XML and DB2. In *ICDE*, pages 569–573, 2000.

[16] S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. *Journal of the Association for Computing Machinery*, pages 742–760, Oct 1984.

[17] Y. Cui, J. Widom, and J. L. Wienner. Tracing the lineage of view data in a warehousing environment. In *ACM Transactions on Database Systems*, volume 25(2), pages 179–227, June 2000.

[18] 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.

[19] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[20] 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.

[21] 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.

[22] A. M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB*, pages 467–474, 1986.

[23] T. W. Ling and M.-L. Lee. A Theory for Entity-Relationship View Updates. In *ER*, pages 262–279, 1992.

[24] K. Loney. *Oracle Database 10g : The Complete Reference*. McGraw-Hill, 2004.

[25] M. Fernandez et al. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[26] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. In *ACM TOIT*, 2005.

[27] P. I. Resource. Protein Sequence Database. http://pir.georgetown.edu/.

[28] 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.

[29] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, May 2001.

[30] W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.

[31] L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *XML Database Symposium*, pages 223–237, 2003.

[32] L. Wang and E. A. Rundensteiner. On the Updatability of XQuery Views Publised over Relational Data. In *ER*, pages 795–809, 2004.

[33] L. Wang, E. A. Rundensteiner, and M. Mani. U-Filter: A Full-fledged XML-to-Relational Update Translatability Checking Framework. Technical Report WPI-CS-TR-05-11, Computer Science Department, WPI, 2005.

[34] 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*, 2005 to appear.