On the Updatability of XML Views Published over Relational Data

Ling Wang and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
{lingw, rundenst}@cs.wpi.edu

Abstract. Updates over virtual XML views that wrap the relational data have not been well supported by current XML data management systems. This paper studies the problem of the existence of a correct relational update translation for a given view update. First, we propose a *clean extended-source theory* to decide whether a translation mapping is correct. Then to answer the question of the existence of a correct mapping, we classify a view update as either *un-translatable*, *conditionally* or *unconditionally translatable* under a given *update translation policy*. We design a graph-based algorithm to classify a given update into one of the three update categories based on schema knowledge extracted from the XML view and the relational base. This now represents a practical approach that could be applied by any existing view update system in industry and in academic for analyzing the translatability of a given update statement before translation of it is attempted.

1 Introduction

Typical XML management systems [5, 9, 14] support the creation of XML wrapping views and the querying against these virtual views to bridge the gap between relational databases and XML applications. Update operations against such wrapper views, however, are not well supported yet.

The problem of updating XML views published over relational data comes with new challenges beyond those of updating relational [1, 7] or even object-oriented [3] views. The first is the *updatability*. That is, the mismatch between the hierarchical XML view model and the flat relational base model raises the question whether the given view update is even mappable into SQL updates. The second is the *translation strategy*. That is, assuming the view update is indeed translatable, how to translate the XQuery updates statements on the XML view into the equivalent tuple-based SQL updates expressed on the relational base.

Translation strategies have been explored to some degree in recent work. [11] presents an XQuery update grammar and studies the execution performance of translated updates. However, the assumption made in this work is that the given update is indeed translatable and that in fact it has already been translated into SQL updates over a relational database, which is assumed to be created by a fixed inline loading strategy [8]. Commercial database systems such as SQL-Server2000 [10], Oracle [2] and DB2 [6] also provide system-specific solutions for restricted update types, again under the assumption of given updates always being translatable.

Our earlier work [12] studies the XML view updatability for the "round-trip" case, which is characterized by a pair of invertable lossless mappings for (1) loading the XML documents into the relational bases, and (2) extracting an XML view identical to the original XML document back out of it. We prove that such XML views are always updatable by any update operation valid on the XML view. However, to the best of our knowledge, no result in the literature focuses on a general method to assess the updatability of an *arbitrary* XML view published over an *existing* relational database.

This view updatability issue has been a long standing difficult problem even in the relational context. Using the concept of "clean source", Dayal and Bernstein [7] characterize the *schema conditions* under which a relational view over a single table is updatable. Beyond this result, our current work now analyzes the key factors affecting the view updatability in the XML context. That is, given an *update translation policy*, we classify updates over an XML view as *un-translatable*, *conditionally* or *unconditionally translatable*. As we will show, this classification depends on several features of the XML view and the update statements, including: (a) granularity of the update at the view side, (b) properties of the view *construction*, and (c) types of *duplication* appearing in the view. By extending the concept of a "clean source" for relational databases [7] into "clean extended-source" for XML, we now propose a theory for determining the existence of a correct relational update translation for a given XML view update.

We also provide a graph-based algorithm to identify the conditions under which an XML view over a relational database is updatable. The algorithm depends only on the view and database schema knowledge instead of on the actual database content. It rejects un-translatable updates, requests additional conditions for conditionally translatable updates, and passes unconditionally translatable updates to the later update translation step. The proof of correctness of our algorithm can be found in our technical report [13]. It utilizes our *clean extended-source theory*.

Section 2 analyzes the factors deciding the XML view updatability, which is then formalized in Section 3. In Section 4 we propose the "clean extended-source" theory as theoretical foundation of our proposed solution. Section 5 describes our graph-based algorithm for detecting update translatability. Section 6 provides conclusions.

2 Factors for XML View Updatability

Using examples, we now illustrate what factors affect the view updatability in general, and which features of XML specifically cause new view update translation issues. Recent XML systems [5, 9, 14] use a *default XML view* to define the

book

bookid	title
98001	TCP/IP Illustrated
98002	Programming in Unix
98003	Data on the Web

Legend: Primary Key / Non Key

price

bookid	amount	website
98001	63.70	www.amazon.com
98003	56.00	www.amazon.com
98003	45.60	www.bookpool.com

```
CREATE TABLE book(
  bookid VARCHAR2(20),
  title VARCHAR2(100),
  CONSTRAINTS BookPK
     PRIMARYKEY (bookid))

CREATE TABLE price(
  bookid VARCHAR2(20),
  amount DOUBLE,
  website VARCHAR2(100),
  CONSTRAINTS PricePK
     PRIMARYKEY (bookid, website),
  FOREIGNKEY (bookid)
     REFERENCES book (bookid))
```

Fig. 1. Relational database

```
<DB>
  <book>
    <row>
      <bookid>98001</bookid>
      <title>TCP/IP Illustrated</title>
    </row> ...
  </book>
  <price>
    <row>
      <bookid>98001</bookid>
      <amount>63.70</amount>
      <website>www.amazon.com</website>
    </row> ...
  </price>
<DB>
```

Fig. 2. Default XML view of database shown in Figure 1

Q1

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
  <book_info>
    $book/bookid,
    $book/title,
    FOR $price
      IN document("default.xml")/price/row
    WHERE
        $book/bookid = $price/bookid
    RETURN {
      <price_info>
        $price/amount,
        $price/website
      </price_info>}
  </book_info>
</bib> }
```

V1

```
<bib>
  <book_info>
    <bookid>98001</bookid>
    <title> TCP/IP Illustrated </title>
    <price_info>
      <amount>63.70</amount>
      <website> www.amazon.com</website>
    </price_info>
  </book_info>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <amount>56.00</amount>
      <website> www.amazon.com</website>
    </price_info>
    <price_info>
      <amount>45.60</amount>
      <website>
        www.bookpool.com
      </website>
    </price_info>
  </book_info>
</bib>
```

(a) View V1 defined by Q1

Q2

```
<bib>
FOR $book IN document("default.xml")/book/row,
      $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
  <price_info>
      $price/amount,
      $price/website,
      <book_info>
        $book/bookid,
        $book/title
      </book_info>
  </price_info>
</bib> }
```

V2

```
<bib>
  <price_info>
    <amount>63.70</amount>
    <website> www.amazon.com </website>
    <book_info>
      <bookid>98001</bookid>
      <title>TCP/IP Illustrated</title>
    </book_info>
  </price_info>
  <price_info>
    <amount>56.00</amount>
    <website> www.amazon.com </website>
    <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
    </book_info>
  </price_info>
  <price_info>
    <amount>45.60</amount>
    <website> www.bookpool.com </website>
    <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
    </book_info>
  </price_info>
</bib>
```

(b) View V2 defined by Q2

Q3

```
<bib>
FOR $book IN document("default.xml")/book/row,
      $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
  <book_info>
      $book/bookid,
      $book/title,
      <price_info>
        $price/amount,
        $price/website
      </price_info>
  </book_info>
</bib>}
```

V3

```
<bib>
  <book_info>
    <bookid>98001</bookid>
    <title> TCP/IP Illustrated </title>
    <price_info>
      <amount>63.70</amount>
      <website>  www.amazon.com  </website>
    </price_info>
  </book_info>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <amount>56.00</amount>
      <website>  www.amazon.com </website>
    </price_info>
  </book_info>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
    </price_info>
  </book_info>
</bib>
```

(c) View V3 defined by Q3

Q4

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
  <book_info>
    $book/bookid,
    $book/title,
    FOR $price
      IN document("default.xml")/price/row
    WHERE $book/bookid = $price/bookid
    RETURN {
      <price_info>
        $book/bookid,
        $price/amount,
        $price/website
      </price_info>}
  </book_info>
</bib> }
```

V4

```
<bib>
  <book_info>
    <bookid>98001</bookid>
    <title> TCP/IP Illustrated </title>
    <price_info>
      <bookid>98001</bookid>
      <amount>63.70</amount>
      <website>  www.amazon.com  </website>
    </price_info>
  </book_info>
  <book_info>
    </bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <bookid>98003</bookid>
      <amount>56.00</amount>
      <website>  www.amazon.com </website>
    </price_info>
    <price_info>
      <bookid>98003</bookid>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
    </price_info>
  </book_info>
</bib>
```

(d) View V4 defined by Q4

Fig. 3. View V1 to V4 defined by XQuery Q1 to Q4 respectively

one-to-one XML-to-relational mapping (Fig. 2). A *view query* (Fig. 3) is defined over it to express user-specific XML wrapper views. User updates over the virtual XML views are expressed in XQuery update syntax [11] (Fig. 4). Also, we only consider insertion/deletion in our discussion. A replacement is treated as a deletion followed by an insertion without specifically discussion.

2.1 Update Translation Policy

Clearly, the *update translation policy* chosen for the system is essential for the decision of view updatability. An update may be translatable under one policy, while not under another one. We now enumerate common policies observed in the literature [3, 11] and in practice [14].

Policies for update type selection. (1) Same type. The translated update always must have the same update type as the given view update. (2) Mixed type. Translated updates with a different type are allowed.

Policies for maintaining referential integrity of the relational database under deletion. (1) Cascade. The directly translated relational updates cascade to update the referenced relations as well. (2) Restrict. The relational update is restricted to the case when there are no referenced relations. Otherwise, reject the view update. (3) Set Null. The relational update is performed as required, while the foreign key is set to be NULL in each dangling tuple.

The translatability of a *valid view update* under a given policy can be classified as *unconditionally translatable*, *conditionally translatable* and *un-translatable*. A view update is called *un-translatable* if it cannot be mapped into relational updates without violating some consistency. A view update is *unconditionally translatable* if such a translation always exists under the given policy. Otherwise, we call it *conditionally translatable*. That is, under the current update policy, the given update is not translatable unless additional conditions, such as assumptions or user communication, are introduced to make it translatable.

When not stated otherwise, throughout the paper we pick the most commonly used policy, that is, *same update type* and *delete cascade*. If a different translation policy is used, then the discussion can be easily adjusted accordingly. Also, we do not indicate the order of the translated relational updates. For a given execution strategy, the correct order can be easily decided [1, 11, 12].

2.2 New Challenges Arising from XML Data Model

Example 1. **: View construction consistency.**
Assume two view updates u_1^V and u_2^V (Fig. 4) delete a "book_info" element from $V1$ and $V2$ in Fig. 3 respectively.

(i) Fig. 5 shows u_1^V is *unconditionally translatable*. The translated relational update sequence U^R in Fig. 5(b) will delete the first book from the "book" relation by u_1^R, and its prices from the "price" relation through u_2^R. By reapplying the view query $Q1$ on the updated database D' in Fig. 5(c), the updated XML view in Fig. 5(d) equals the user *expected* updated view $V1'$ in Fig. 5(a).

u^V_1
```
FOR $root IN document("V1.xml"),
    $book IN $root/book_info
WHERE $book/title/text() = " TCP/IP Illustrated"
UPDATE $root {
    DELETE $book }
```

u^V_2
```
FOR $root IN document("V2.xml"),
    $book IN $root/price_info/book_info
WHERE $book/title/text() = " TCP/IP Illustrated"
UPDATE $root {
    DELETE $book }
```

u^V_3
```
FOR $root IN document("V3.xml"),
    $book IN $root/book_info
WHERE $book/title/text() = " Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
    DELETE $book }
```

u^V_4
```
FOR $root IN document("V4.xml"),
    $book IN $root/book_info
WHERE $book/title/text() = "Data on the Web"
AND $book/price_info/website = " www.amazon.com "
UPDATE $root {
    DELETE $book/price_info}
```

u^V_5
```
FOR $root IN document("V2.xml"),
    $price IN $root/price_info
WHERE $price/book_info/title/text() = "TCP/IP Illustrated"
UPDATE $root {
    DELETE $price }
```

u^V_6
```
FOR $root IN document("V3.xml"),
    $book IN $root/book_info
WHERE $book/title/text() = " Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
    DELETE $book/price_info }
```

u^V_7
```
FOR $root IN document("V4.xml"),
    $book IN $root/book_info
WHERE $book/title/text() = "Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
    DELETE $book }
```

u^V_8
```
FOR $root IN document("V3.xml")
UPDATE $root {
  INSERT
    <book_info>
      <bookid>"98003"<bookid>
      <title>" Data on the Web "</title>
      <price_info>
        <amount>56.00</amount>
        <website>www.ebay.com</website>
      </price_info>
    </book_info> }
```

Fig. 4. Update operations on XML views defined in Fig.3

```
<bib>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <amount>56.00</amount>
      <website>www.amazon.com</website>
    </price_info>
    <price_info>
      <amount>45.60</amount>
      <website>www.bookpool.com</website>
    </price_info>
  </book_info>
</bib>
```

(a) $V1'$

(d) $Q1(D')$. Same with (a).

```
u1R: DELETE FROM book
     WHERE book.ROWID IN (
       SELECT DISTINCT book.ROWID FROM book
       WHERE (book.title = 'TCP/IP Illustrated') )

u2R: DELETE FROM price
     WHERE price.ROWID IN (
       SELECT DISTINCT price.ROWID FROM book,price
       WHERE (book.title = 'TCP/IP Illustrated') AND
             (book.bookid = price.bookid) )
```

(b) U^R

book

bookid	title
98002	Programming in Unix
98003	Data on the Web

price

bookid	amount	website
98003	56.00	www.amazon.com
98003	45.60	www.bookpool.com

(c) D'

Fig. 5. Translate u^V_1 (a) $V1'$: The user expected updated view, (b) U^R: The translated update, (c) D': The updated relational database, (d) $Q1(D')$: The regenerated view.

(ii) Fig. 6 shows u^V_2 is *un-translatable*. First, the relational update u^R_1 in Fig. 6(b) is generated to delete the book (bookid=98001) from the "book" relation. Note the foreign key from the "price" relation to the "book" relation (Fig. 1). The second update operation u^R_2 will be generated by the update translator to keep the relational database consistent. The regenerated view in Fig. 6(d) is different than the user expected updated view $V2'$ in Fig. 6(a). No other translation is available which could preserve consistency either.

The existence of a correct translation is affected by the *view construction consistency* property, namely, whether the XML view hierarchy agrees with the hierarchical structure implied by the base relational schema.

Example 2. **: Content duplication.**
Next we compare the two virtual XQuery views $V1$ and $V3$ in Fig. 3. The book (bookid=98003) with two prices is exposed twice in $V3$, while only once in $V1$. The update u^V_3 in Fig. 4 will delete the "book_info" element from amazon, while keeping the one from bookpool. Now should we delete the book tuple underneath? It is unclear. An additional condition, such as an extra translation rule like "*No underlying tuple is deleted if it is still referenced by any other part*

```
<bib>
  <price_info>
    <amount>63.70</amount>
    <website>www.amazon.com</website>
  </price_info>
  <price_info>
    <amount>56.00</amount>
    <website>www.amazon.com</website>
    <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
    </book_info>
  </price_info>
  ...
</bib>
```

(a) $V2'$

(b) U^R. Same with Fig.5(b).

(c) D'. Same with Fig.5(c).

```
<bib>
  <price_info>
    <amount>56.00</amount>
    <website>www.amazon.com</website>
    <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
    </book_info>
  </price_info>
  ...
</bib>
```

(d) $Q2(D')$

Fig. 6. Translate u_2^V (a) $V2'$: The user expected updated view, (b) U^R: The translated update, (c) D': The updated relational database, (d) $Q2(D')$: The regenerated view.

of the view" could make the update u_3^V translatable by keeping the book tuple untouched. This update is thus called *conditionally translatable*.

This ambiguous *content duplication* is introduced by the XQuery "FOR" expression. This property could also arise in relational *Join* views.

Example 3. **: Structural duplication.**
Given $Q4$ in Fig. 3 with each "bookid" exposed twice in the single "book_info" element. The update u_4^V in Fig. 4, which deletes the first price of the specified book, is classified as *conditionally translatable*. Since the primary key "bookid" is touched by u_4^V, we cannot decide whether to delete the book-tuple underneath. With an additional condition, such as knowledge of the user intention about the update, u_4^V becomes translatable.

Structural duplication, as illustrated above, is special to XML view updating. While it also exists in the relational context, it would not cause any ambiguity. The flat relational data model only allows *tuple-based* view insertion/deletion. The update touches all not just some of the duplicates within a view tuple. Instead of always enforcing an update on the biggest view element "book_info", the flexible hierarchical structure of XML allows a "partial" update on subelements inside it. Inconsistency between the duplicated parts thus occurs.

Example 4. **: Update granularity.**
Compared with the failure of translating u_2^V in Example 1, the update u_5^V in Fig. 4 on the same view $V2$ is *conditionally translatable*. u_5^V deletes the whole "price_info" element instead of just the sub-element "book_info". The translated relational update sequence U^R is the same as in Fig. 6(b). The regenerated view is the same as what the user expects. Due to content duplication, u_5^V is said to be *conditionally translatable*.

XML hierarchical structure offers an opportunity for different *update granularity*, an issue that does not arise for relational views.

3 Formalizing the Problem of XML View Updatability

The structure of a relation is described by a **relation schema** $\mathcal{R}(\mathcal{N}, \mathcal{A}, \mathcal{F})$, where $\mathcal{N}$ is the name of the relation, $\mathcal{A} = \{a_1, a_2, ..., a_m\}$ is its attribute set, and $\mathcal{F}$ is a set of constraints. A **relation** R is a finite subset of $dom(\mathcal{A})$, a *product* of all the attribute domains. A **relational database**, denoted as D, is a set of n relations $R_1, ..., R_n$. A **relational update operation** $u^R \in \mho^R$ is a deletion, insertion or replacement on a relation R. A **sequence** of relational update operations, denoted by $U^R = \{u_1^R, u_2^R, ..., u_p^R\}$ is also modeled as a function $U^R(D) = u_p^R(u_{p-1}^R(..., u_2^R(u_1^R(D))))$.

Table 1. Notations for XML view update problem

D	relational database	$\mathcal{R}(\mathcal{N}, \mathcal{A}, \mathcal{F})$	schema of relation
R	relation	$\mho^R$	domain of relational update operations
u^R	relational update operation	U^R	sequence of relational update operations
V	XML view	DEF^V	XML view definition
u^V	view update	$\mho^V$	domain of view update operations

An **XML view** V over a relational database D is defined by a **view definition** DEF^V (an XQuery expression in our case). The domain of the view is denoted by $dom(V)$. Let *rel* be a function to extract the relations in D referenced by DEF^V, then $rel(DEF^V) = \{R_{i_1}, R_{i_2}, ..., R_{i_p}\} \subseteq D$. An **XML view schema** is extracted from both DEF^V and $rel(DEF^V)$. See [13] for details.

Let $u^V \in \mho^V$ be an update on the view V. A **valid view update** (e.g., Fig. 4) is an insertion or deletion that satisfies all constraints in the view schema.

Definition 1. *Given an update translation policy. Let D be a relational database and V be a virtual view defined by DEF^V. A relational update sequence U^R is a* **correct translation** *of u^V iff (a) $u^V(DEF^V(D)) = DEF^V(U^R(D))$ and (b) if $u^V(DEF^V(D)) = DEF^V(D) \Rightarrow U^R(D) = D$.*

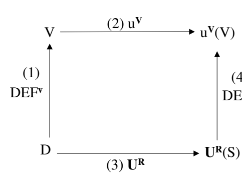


Fig. 7. Correct translation of view update to relational update

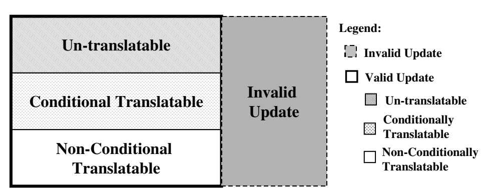


Fig. 8. The partition of view update domain $\mho^V$

First, a correct translation means the "rectangle" rule holds (Fig. 7). Intuitively, it implies the translated relational updates do not cause any *view side effects*. Second, if an update operation does not affect the view, then it should not affect the relational base either. This guarantees any modification of the relational base is indeed done for the sake of the view.

Fig. 8 shows a typical partition of the view update domain $\mho^V$. The **XML view updatability** classifies a valid view update as either *unconditionally translatable*, *conditionally translatable* or *un-translatable*.

4 Theoretical Foundation for XML View Updatability

Dayal and Bernstein [7] show that a correct translation exists in the case of a "clean source", when only considering functional dependencies inside a single relation. In the context of XML views, we now adopt and extend this work to also consider functional dependencies between relations.

Definition 2. *Given a relational database D and an XML view V defined over several relations $rel(DEF^V) \subseteq D$. Let v be a view element of V. Let $g = (t_1, ..., t_p)$ be a generator of v, where $t_i \in R_x$ for $R_x \in rel(DEF^V)$. Then t_i is called a* **source tuple** *in D of v.*

Further, $t_j \in R_y$ is an **extended source tuple** *in D of v iff $\exists t_i \in g$ that $t_i.a_k$ is a foreign key of $t_j.a_z$, where $a_k \in \mathcal{R}_x(\mathcal{A})$, $a_z \in \mathcal{R}_y(\mathcal{A})$ and $R_x, R_y \in rel(DEF^V)$. $g_e = g \cup \{t_j \mid t_j$ is an extended source tuple of $v\}$ is called an* **extended generator** *of v.*

A *source tuple* is a relational row used to compute the view element. For instance, in $V1$ of Fig. 3, the first view element v_1 is *book_info* element with bookid=98001. Let R_1 and R_2 denote the *book* and *price* relations respectively, then the generator g of v_1 is $g = (t_1, t_2)$, where $t_1 \in R_1$ is the book tuple *(98001,TCP/IP Illustrated)* and $t_2 \in R_2$ is the price tuple *(98001, 63.70, www.amazon.com)*. Let the view-element v_2 be the title of v_1. Then the source tuple of v_2 is t_1. Since t_1.*bookid* is a foreign key of t_2.*bookid*, we say t_2 is an extended source tuple of v_2, and $g_e = (t_1, t_2)$ is an extended generator of v_2.

Definition 3. *Let V^0 be a part of a given XML view V. Let $G(V^0)$ be the set of generators of V^0 defined by $G(V^0) = \{g \mid g$ is a generator of a view-element in $V^0\}$. For each $g = (t_1, ..., t_p) \in G(V^0)$, let $H(g)$ be some nonempty subset of $\{t_i \mid t_i \in g\}$. Then any superset of $\cup_{g \in G(V^0)} H(g)$ is a* **source** *in D of V^0. (If $G(V^0) = \emptyset$, then V^0 has no source in D.)*

Similarly, let $G_e(V^0)$ be the set of extended generators for view elements in V^0. Then any superset of $\cup_{g \in G_e(V^0)} H(g)$ is an **extended source** *in D of V^0, denoted by S_e.*

A *source* includes the underlying relational part of a view "portion" V^0 which consists of multiple view-elements. For example, let $V^0 = V1$ (Fig. 3), $G(V^0) = \{g_1, g_2\}$, where $g_1 = \{$*(98001,TCP/IP Illustrated),(98001, 63.70,www.amazon.com)*$\}$, $g_2 = \{$*(98003,Data on the Web),(98003,56.00,www.amazon.com),(98003,45.60, www.bookpool.com)*$\}$. That is, $G(V^0)$ includes all the generators for view elements in V^0. Let $H(g_1) = \{$*(98001, TCP/IP Illustrated)*$\}$ and $H(g_2) = \{$*(98003, 56.00, www.amazon.com)*$\}$. Then $\{$*(98001, TCP/IP Illustrated),(98003, 56.00, www.amazon.com)*$\}$ is a source of V^0, also an extended source of V^0.

Definition 4. *Let $D = \{R_1, ..., R_n\}$ be a relational database. Let V^0 be part of a given XML view V and S_e be an extended source in D of V^0. S_e is a* **clean extended source** *in D of V^0 iff $(\forall v \in V - V^0)$, $(\nexists S'_e)$ such that S'_e is an extended source in $(R_1 - S_{e1}, ..., R_n - S_{en})$ of v. Or, equivalently, S_e is a clean extended source in D of V^0 iff $(\forall v \in V - V^0)(S_e$ is not an extended source in D of $v)$.*

A *clean extended source* defines a source that is only referenced by the given view element itself. For instance, given the view-element v in $V2$ (Fig. 3) representing the *book_info* element (bookid = 98001), its extended source {*(98001, TCP/IP Illustrated),(98001, 63.70,www.amazon.com)*} is not a *clean extended source* since it is also an extended source of the *price* element.

The **clean extended source theory** below captures the connection between *clean extended source* and update translatability (Proofs in [13]). It serves as a conservative solution for identifying the (*unconditionally) translatable* updates.

Theorem 1. *Let u^V be the deletion of a set of view elements $V^d \subseteq V$. Let τ be a translation procedure, $\tau(u^V, D) = U^R$. Then τ* **correctly translates** *u^V to D iff U^R deletes a clean extended source of V^d.*

By Definition 1, a correct delete translation is one without any view side effect. This is exactly what deleting a clean extended-source guarantees by Definition 4. Thus Theorem 1 follows.

Theorem 2. *Let u^V be the insertion of a set of view elements V^i into V. Let $V^- = V - V^i$, $V^u = V^i - V$. Let τ be a translation procedure, $\tau(u^V, D) = U^R$. Then τ* **correctly translates** *u^V to D iff (i) $(\forall v \in V^u)$(U^R inserts a source tuple of v) and (ii) $(\forall v \in dom(V) - (V^u \cup V^-))$($U^R$ does not insert a source tuple of v).*

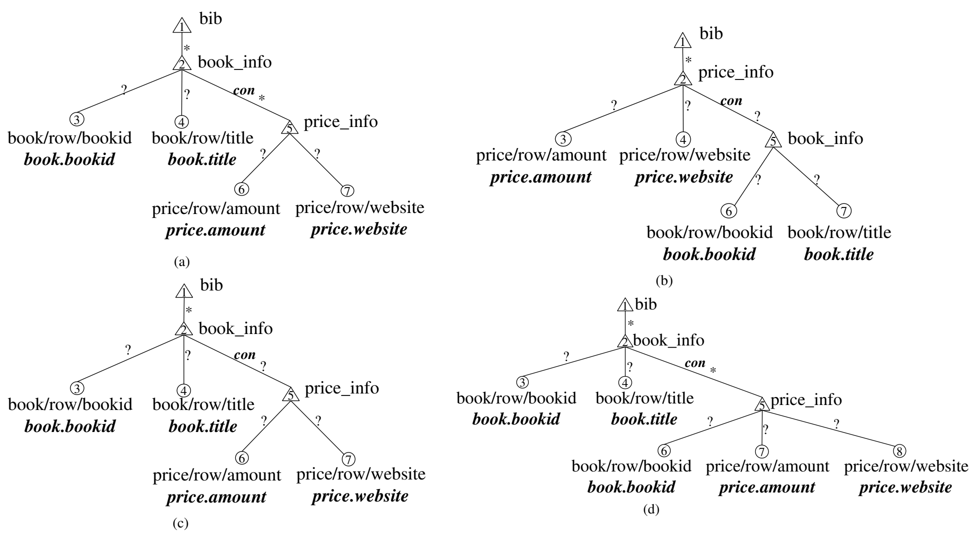
Since $dom(V) - (V^u \cup V^-) = (dom(V) - (V^i \cup V)) \cup (V^i \cap V)$, Theorem 2 indicates a correct insert translation is the one without any duplicate insertion (insert a source of $V^i \cap V$) and any extra insertion (insert a source of $dom(V) - (V^i \cup V)$). That is, it inserts a clean extended source for the new view-element. Duplicate insertion is not allowed by BCNF, while extra insertion will cause a view side effect. For example, for u_8^V in Fig. 4, let u_1^R = {*Insert (98003,Data on the Web) into book*}, u_2^R = {*Insert (98003,56.00,www.ebay.com) into price*}. Then $U^R = \{u_1^R, u_2^R\}$ is not a correct translation since it inserts a duplicate source tuple into *book*. While $U^{R'} = \{u_2^R\}$ is a correct translation.

5 Graph-based Algorithm for Deciding View Updatability

We now propose a graph-based algorithm to identify the factors and their effects on the update translatability based on our clean extended source theory. We assume the relational database is in the BCNF form. No cyclic dependency caused by integrity constraints among relations exists. Also, the predicate used in the view query expression is a conjunction of *non-correlation* (e.g., \$*price/website = "www.amazon.com"*) or *equi-correlation* predicates (e.g., \$*book/bookid* = \$*price/bookid*).

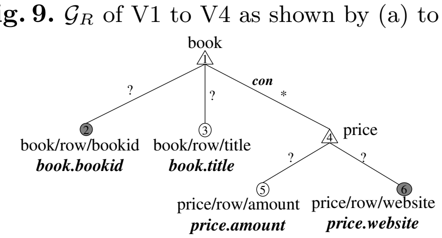
5.1 Graphic Representation of XML Views

Two graphs capture the update related features in the view V and relational base D. The **view relationship graph** $\mathcal{G}_R(N_{\mathcal{G}_R}, E_{\mathcal{G}_R})$ is a forest representing the hierarchical and cardinality constraints in the *XML view schema*. An internal node, represented by a triangle △, identifies a view element or attribute labeled by its name. A leaf node (represented by a small circle ∘) is an atomic type, labeled by both the XPath binding and the name of its corresponding relational column $R_x.a_k$. An edge $e(n_1, n_2) \in E_{\mathcal{G}_R}$ represents that n_1 is a parent of n_2 in the view hierarchy. Each edge is labeled by the cardinality relationship and condition (if any) between its end nodes. A label "?" means each parent node can only have one child, while "$*$" shows multiple children are possible. Figures 9(a) to 9(d) depict the view relationship graphs for $V1$ to $V4$ in Fig. 3 respectively.



*Note: con = (book/row/bookid=price/row/bookid)

Fig. 9. $\mathcal{G}_R$ of V1 to V4 as shown by (a) to (d)



*Note: con = (book.bookid=price.bookid)

Fig. 10. $\mathcal{G}_T$ of V1 – V4

Definition 5. *The* **hierarchy implied in relational model** *is defined as:*
(1) Given a relation schema $\mathcal{R}(\mathcal{N}, \mathcal{A}, \mathcal{F})$*, with* $\mathcal{A} = \{a_i | 1 \leq i \leq m\}$*, then* $\mathcal{N}$ *is called the* **parent** *of the attribute* a_i *(* $1 \leq i \leq m$ *).*
(2) Given two relation schemas $\mathcal{R}_i(\mathcal{N}_i, \mathcal{A}_i, \mathcal{F}_i)$ *and* $\mathcal{R}_j(\mathcal{N}_j, \mathcal{A}_j, \mathcal{F}_j)$*, with foreign key constraints defined as* $PK(\mathcal{R}_i) \leftarrow FK(\mathcal{R}_j)$*, then* $\mathcal{N}_i$ *is the* **parent** *of* $\mathcal{N}_j$*.*

The **view trace graph** $\mathcal{G}_T(N_{\mathcal{G}_T}, E_{\mathcal{G}_T})$ represents the hierarchical and cardinality constraints in the *relational schema* underlying the XML view. The set of leaf nodes of $\mathcal{G}_T$ correspond to the union of all leaves of $\mathcal{G}_R$. Specially, a leaf node labeled by the primary key attribute of a relation is called a *key node* (depicted by a black circle $\bullet$). An internal node, depicted by a triangle $\triangle$, is labeled by the relation name. Each edge $e(n_1, n_2) \in E_{\mathcal{G}_T}$ means n_1 is the parent of n_2 by Definition 5. An edge is labeled by its foreign key condition (if it is generated by rule (2) in Definition 5), and the cardinality relationship between its end nodes. The view trace graphs of $V1$ to $V4$ are identical (Fig. 10), since they all defined over the same attributes of base relations.

The concept of closure in $\mathcal{G}_R$ and $\mathcal{G}_T$ is used to represent the "effect" of an update on the view and on the relational database respectively. Intuitively, their relationship indicates the updatability of the given view.

The **closure** of a node $n \in N_{\mathcal{G}_R}$, denoted by $n^+_{\mathcal{G}_R}$, is defined as follows: (1) If n is a leaf node, $n^+_{\mathcal{G}_R} = \{n\}$. (2) Otherwise, $n^+_{\mathcal{G}_R}$ is the union of its children's closures grouped by their hierarchical relationship and marked by their cardinality (for simplicity, not shown when cardinality mark is ?). For example, in Figure 9(a), $(n_3)^+_{\mathcal{G}_R} = \{n_3\}$, while $(n_5)^+_{\mathcal{G}_R} = \{n_6, n_7\}$, $(n_2)^+_{\mathcal{G}_R} = \{n_3, n_4, (n_6, n_7)^*\}$.

The **closure** of a node $n \in N_{\mathcal{G}_T}$ is defined in the same manner as in $\mathcal{G}_R$, except for leaf nodes. Each leaf node has the same closure as its parent node. For instance, in Fig. 10, $(n_2)^+_{\mathcal{G}_T} = (n_3)^+_{\mathcal{G}_T} = (n_1)^+_{\mathcal{G}_T} = \{n_2, n_3, (n_5, n_6)^*\}$. This closure definition in $\mathcal{G}_T$ is based on the pre-selected update policy in Section 2.1. If a different policy were used, then the definition needs to be adjusted accordingly. For example, if we pick the *mixed type*, the closure will be "only the *key node* has the same closure definition as its parent node, while any other leaf node has itself as the closure". Consequently in Fig. 10, $(n_3)^+_{\mathcal{G}_T} = \{n_3\}$, while $(n_2)^+_{\mathcal{G}_T} = \{n_2, n_3, (n_5, n_6)^*\}$. The delete on these non-key leaf nodes can be translated as a replacement on the corresponding relational column.

To reduce the closure definition, the group mark "()" can be eliminated if its cardinality mark is "?". For example, in Figure 9(c), $(n_2)^+_{\mathcal{G}_T} = \{n_3, n_4, (n_6, n_7)\} = \{n_3, n_4, n_6, n_7\}$. The closure of a set of nodes N, denoted by N^+, is defined as $N^+ = \bigcup_{(n_i \in N)} n_i^+$, where $\bigcup$ is a "Union-like" operation that combines not only the nodes but their shared occurrence. For instance, in Fig. 10, $\{n_2, n_5\}^+_{\mathcal{G}_T} = (n_2)^+_{\mathcal{G}_T} \bigcup (n_5)^+_{\mathcal{G}_T} = \{n_2, n_3, (n_5, n_6)^*\} \bigcup \{n_5, n_6\} = \{n_2, n_3, (n_5, n_6)^*\}$. Two leaf nodes in $\mathcal{G}_R$ or $\mathcal{G}_T$ are *equal* if and only if the relational attribute labels in their respective node labels are the same.

5.2 A Graph-based Algorithm for View Updatability Identification

Definition 6. *Two closures C_1 and C_2* **match***, denoted by $C_1 \cong C_2$, iff the node set of C_1 and C_2 are equal. Further, C_1 and C_2 are* **equal***, denoted by $C_1 \equiv C_2$, iff the node groups, cardinality marks of each group, and conditions on each "*" edge are all the same.*

For two closures to match means that the same view schema nodes are included. While equality indicates that the same instances of XML view elements will be included. For example, $(n_2)^+_{\mathcal{G}_R}$ in Fig. 9(c) and $(n_1)^+_{\mathcal{G}_T}$ in Fig.

10 *match*. That is, both closures include the same XML view schema nodes: *book.bookid, book.title, price.amount, price.website*. However, $(n_2)^+_{\mathcal{G}_R}$ in Figure 9(a) and $(n_1)^+_{\mathcal{G}_T}$ in Figure 10 are *equal*, namely {*book.bookid, book.title, (price.amount, price.website)**}. This is because their group partition (marked by "()"), cardinality mark (* or ?) and conditions for each "*" edge are all the same. Both closures touch exactly the same XML view-element instances.

Theorem 3. *Let V be a view defined by DEF^V over a relational database D with the view relationship graph $\mathcal{G}_R(N_{\mathcal{G}_R}, E_{\mathcal{G}_R})$ and view trace graph $\mathcal{G}_T(N_{\mathcal{G}_T}, E_{\mathcal{G}_T})$. Let $Y \subseteq N_{\mathcal{G}_R}$ and $X \subseteq N_{\mathcal{G}_T}$. ($\forall$ generators g, g' of view elements v and v' respectively, $g[X] = g'[X] \Rightarrow v[Y] = v'[Y]$) iff their closures $X^+_{\mathcal{G}_T} \equiv Y^+_{\mathcal{G}_R}$.*

Theorem 3 indicates that two equal generators always produce the identical view elements iff the respective closures of the view schema nodes in $\mathcal{G}_R$ and $\mathcal{G}_T$ are equal. Theorem 3 now enables us to produce an algorithm for detecting the *clean extended sources* S_e of a view element based on schema knowledge captured in $\mathcal{G}_R$ and $\mathcal{G}_T$.

Theorem 4. *Let $V, DEF^V, D, \mathcal{G}_R, \mathcal{G}_T, Y$ be defined as in Theorem 3. Given a view element $v \in V(Y)$, there is a clean extended source S_e of v in D iff ($\exists X \subseteq N_{\mathcal{G}_T}$) such that $X^+_{\mathcal{G}_T} \equiv Y^+_{\mathcal{G}_R}$.*

Theorem 4 indicates that a given view element v has a clean extended source iff the closure of its schema node in $\mathcal{G}_R$ has an equal closure in $\mathcal{G}_T$. As indicated by Theorems 1 and 2, the existence of a clean extended source for a given XML view element implies that the update touching this element is unconditionally translatable. The following observation thus serves as a general methodology for view updatability determination.

Observation 1 *Let $D, V, \mathcal{G}_R, \mathcal{G}_T, Y$ be defined as in Theorem 3. (1) Updates that touch Y are* **unconditionally translatable** *iff ($\exists X \subseteq N_{\mathcal{G}_T}$) such that $X^+_{\mathcal{G}_T} \equiv Y^+_{\mathcal{G}_R}$. (2) Updates that touch Y are* **conditionally translatable** *iff ($\exists X \subseteq N_{\mathcal{G}_T}$) such that $X^+_{\mathcal{G}_T} \cong Y^+_{\mathcal{G}_R}$. (3) Otherwise, updates on Y are* **un-translatable**.

However, searching all node closures in $\mathcal{G}_T$ to find one equal to the closure of a given view-element is expensive. According to the generation rules of $\mathcal{G}_T$, the nodes in the closure of v also serve as leaf nodes in $\mathcal{G}_T$. We thus propose to start searching from leaf nodes within the closure, thus reducing the search space. Observation 2 utilized the following definition to determine the translatability of a given view update.

Definition 7. *Let n be a node in $\mathcal{G}_R(V)$, with its closure in $\mathcal{G}_R$ denoted by $C_R = n^+_{\mathcal{G}_R}$. Let $C_T = \bigcup_{(n_i \in C_R)} (n_i)^+_{\mathcal{G}_T}$, where $(n_i)^+_{\mathcal{G}_T}$ be the closure of n_i in $\mathcal{G}_T$. We say n is a* **clean node** *iff $C_R \equiv C_T$, a* **consistent node** *iff $C_R \cong C_T$ and an* **inconsistent node** *otherwise.*

For a node to be *inconsistent* means that the effect of an update on the view (node closure in $\mathcal{G}_R$) is different from the effect on the relational side (node closure in $\mathcal{G}_T$) based on the selected policy (closure definition in $\mathcal{G}_T$). It is thus un-translatable. A *clean* node is guaranteed to be safely updatable without any view side-effects. A dirty *consistent* node, however, needs an additional condition to be updatable. For example, n_5 in Fig. 9(a) is a clean node. In Fig. 9(b), n_5 is an inconsistent node and n_2 is a dirty consistent node.

Observation 2 *An update on a clean node is unconditionally translatable, on a consistent node it is conditionally translatable, while on an inconsistent node it is un-translatable.*

Algorithm 1 Optimized Update Translatability Checking Algorithm

/*Given $\mathcal{G}_R$ and $\mathcal{G}_T$ of a view V, determine the translatability of a view update u*/

Procedure checkTranslatability(u, $\mathcal{G}_R$, $\mathcal{G}_T$**)**
Node n = **identifyNodeToUpdate(**u, $\mathcal{G}_R$**)**
classifyNode(n, $\mathcal{G}_R$, $\mathcal{G}_T$**)**
if n is a *clean* node **then**
  n is *unconditionally* translatable
else
  if n is a *consistent* node **then**
    n is *conditionally* translatable
  else
    n is *untranslatable*
  end if
end if

/*Classify the node $n \in \mathcal{G}_R$ to be updated*/
Procedure classifyNode (n, $\mathcal{G}_R$, $\mathcal{G}_T$**)**
Initiate C_R and C_T empty
C_R = **computeClosure(**n, $\mathcal{G}_R$**)**
while C_R has more node **do**
  get the next node $n_i \in C_R$
  $C_T = C_T \cup$ **computeClosure(**n_i, $\mathcal{G}_T$**)**
end while
if $C_R \cong C_T$ **then**
  if $C_R \equiv C_T$ **then**
    n is a *consistent* node
  else
    n is a *clean* node
  end if
else
  n is an *inconsistent* node
end if

Algorithm 1 shows our optimized update translatability checking algorithm using Observation 2. It first identifies the deleting/inserting $\mathcal{G}_R$ node. Then, using Definition 7 the procedure *classifyNode (n,* $\mathcal{G}_R$, $\mathcal{G}_T$*)* determines the type of the node to be updated. Thereafter the given view update can be classified as un-translatable, conditionally or unconditionally translatable by Observation 2. Using this optimized update translatability checking algorithm, a concrete case study on the translatability of deletes and inserts is also provided in [13].

6 Conclusion

In this paper, we have identified the factors determining *view updatability* in general and also in the context of XQuery views in particular. The *extended clean-source theory* for determining translation correctness is presented. A graph-based algorithm has also been presented to identify the conditions under which a correct translation of a given view update exists.

Our solution is general. It could be used by an update translation systems such as [4] to identify the translatable update before translation of it is attempted. This way we would guarantee that only a "well-behaved" view update is passed down to the next translation step. [4] assumes the view is always *well-formed*, that is, joins are through keys and foreign keys, and nesting is controlled to agree with the integrity constraints and to avoid duplication. The update over such a view is thus always translatable. Our work is *orthogonal* to this work by addressing new challenges related to the decision of translation existence when conflicts are possible, that is a view cannot always be guaranteed to be well-formed (as assumed in this prior work).

Our view updatability checking solution is based on schema reasoning, thus utilizes only view and database schema and constraints knowledge. Note that the translated updates might still conflict with the actual base data. For example, an update inserting a book (bookid = 98002) to $V1$ is said to be unconditionally translatable by our schema check procedure, while conflicts with the base data in Fig. 1 may still arise. Depending on selected update translation policy, the translated update can then be either rejected or executed by replacing the existing tuple with the newly inserted tuple. This run-time updatability issue can only be resolved at execution time by examining the actual data in the database.

References

1. A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.
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. On the Updatability of XML Views over Relational Databases. In *WEBDB*, pages 31–36, 2003.
5. 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.
6. J. M. Cheng and J. Xu. XML and DB2. In *ICDE*, pages 569–573, 2000.
7. 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.
8. J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, September 1999.
9. M. Fernandez et al. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.
10. 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.
11. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, May 2001.
12. L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *XML Database Symposium (VLDB Workshop)*, pages 223–237, 2003.
13. L. Wang and E. A. Rundensteiner. Updating XML Views Published Over Relational Databases: Towards the Existence of a Correct Update Mapping. Technical Report WPI-CS-TR-04-19, Computer Science Department, WPI, 2004.
14. X. Zhang, K. Dimitrova, L. Wang, M. EL-Sayed, B. Murphy, L. Ding, and E. A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of SIGMOD*, page 671, 2003.