Reasoning about the updatability of XML views over relational databases

Vanessa P. Braganholo[(1)], Susan B. Davidson[(2)], Carlos A. Heuser[(1)]

[(1)] Universidade Federal do Rio Grande do Sul - UFRGS
Instituto de Informática
{vanessa, heuser}@inf.ufrgs.br

[(2)] University of Pennsylvania
Department of Computer and Information Science
{susan}@cis.upenn.edu

Abstract

XML has become an important medium for data exchange, and is also used as an interface to – i.e. a view of – a relational database. While previous work has considered XML views for the purpose of querying relational databases (e.g. Silkroute), in this paper we consider the problem of updating a relational database through an XML view. Using the nested relational algebra as the formalism for an XML view of a relational database, we study the problem of when such views are updatable. Our results rely on the observation that in many XML views of relational databases, the nest operator occurs last and the unnest operator does not occur at all. Since in this case the nest operator is invertible, we can consider this important class of XML views as if they were flat relational views.

1 Introduction

XML is frequently used for publishing as well as exchanging relational data. Due to the highly unintuitive representation of data in the relational model, it is also increasingly being used as a mechanism through which to query and update legacy relational databases. For example, interfaces for gene expression data frequently represent the data to be annotated as an XML view of a relational database (e.g. AGAVE and GAME [1]).

One reason for this use of XML is that it naturally captures many-one relationships between data through the nesting of elements. In contrast, in the relational model nested data becomes fragmented over many relations, with many-one relationships captured in key and foreign key constraints. Thus one of the advantages of XML for conceptualizing information is its connection to nested relations.

As a simple example, consider the nested table of figure 2(a), which represents information about conferences and their location by year. This same information when represented in the relational model would be split over two tables (see figure 1). The nested table of figure 2(a) can also be understood as the XML instance in figure 3.

While other work has considered the problem of querying relational databases through XML views (e.g. Silkroute [13]), in this paper we focus on the problem of updating a relational database through an XML view. More precisely, we wish to be able to translate an update on an XML view to a set of updates on the underlying relations without introducing additional updates to the XML view.

To simplify the problem yet capitalize on the use of XML to capture nested relations, we consider XML views as defined by the nested relational algebra [27, 17]. The nested relational algebra contains the classical relational algebra operators (σ, π, $\cup$, $\times$, $\bowtie$, $-$) as well as the *nest* (ν) and *unnest* (μ) operators. There are several reasons for considering this algebra: First, there is a straightforward mapping between nested relations and XML views in this language [2, 8]. Second, it represents the core of languages such as XQuery when order and aggregate operators are ignored. Third, certain subclasses of expressions in this algebra have good properties regarding updatability. In this paper we will define such a subclass, drawing on classical relational view updatability results [10, 18]. Surprisingly, except for work on normal forms for nested relations [22, 24, 16]

Conferences

IdConf	ConfName
DEXA	Conference on Database and Expert Systems Applications
PODS	Symposium on Principles of Database Systems
VLDB	Conference on Very Large Data Bases

ConfLocation

IdConf	Year	Location
DEXA	2001	Munich, Germany
PODS	2001	Santa Barbara, California
PODS	2002	Madison, Wisconsin
VLDB	2002	Hong Kong, China
VLDB	2003	Berlin, Germany

Papers

IdPaper	Title	IdConf	Year
1	Web Survey	VLDB	2002
2	Querying the Web	DEXA	2001
3	Databases and IR	VLDB	2002

Constraints:

On table Conferences
CONSTRAINT PK1 primary key (IdConf)

On table ConfLocation
CONSTRAINT PK2 primary key (IdCond, Year)
CONSTRAINT ConfLoc foreign key (IdConf)
references Conferences

On table Papers
CONSTRAINT PK3 primary key (IdPaper)
CONSTRAINT PaperLoc foreign key (IdConf, Year)
references ConfLocation

Figure 1: Sample database

IdConf	ConfName	YearLocation	
		Year	Location
DEXA	Conference on Database and Expert Systems Applications	2001	Munich, Germany
PODS	Symposium on Principles of Database Systems	2001	Santa Barbara, California
		2002	Madison, Wisconsin
VLDB	Conference on Very Large Data Bases	2002	Hong Kong, China
		2003	Berlin, Germany

(a)

IdConf	Details		
	Year	Location	Title
DEXA	2001	Munich, Germany	Querying the Web
VLDB	2002	Hong Kong, China	Databases and IR
	2002	Hong Kong, China	Web Survey

(b)

Figure 2: (a) View 1 (b) View 2

which focuses on removing ambiguity in nested relations, we could find no work related to updates through nested relations.

Using the results of this paper, we will show that the view of figure 2(a) is updatable for all insertions, deletions and modifications. That is, there is a unique, side effect free translation from any update on this view to the underlying relations of figure 1. The view is produced by the following query:

VIEW 1

$$\nu_{YearLocation=(Year,\ Location)}(\pi_{(IdConf,\ ConfName,\ Year,\ Location)} (Conferences \bowtie ConfLocation)))$$

This query is an example of a class which we call *well-nested project-select-join* queries. Views of this class are always updatable.

By relaxing restrictions on nesting and the form of the relational algebra expression, we can obtain a more general class of queries of form $\nu \ldots \nu R$, where R is any relational algebra expression. We will call this class *nest-last relational queries*. An example of this class of views is as follows, and the corresponding nested relation is shown in figure 2(b).

VIEW 2

$$\nu_{Details=(Year,Location,Title)}(\pi_{(IdConf,Year,Location,Title)} (\sigma_{(ConfLocation.IdConf=Papers.IdConf\ AND\ ConfLocation.Year=Papers.Year)} (ConfLocation \times Papers)))$$

Although views defined by queries in this class are not in general updatable, we can determine whether or not a given update can be allowed using results from [10]. For example, in the above view we can always delete tuples but the same is not true for insertions or modifications. That is, we would not be able to insert the tuple

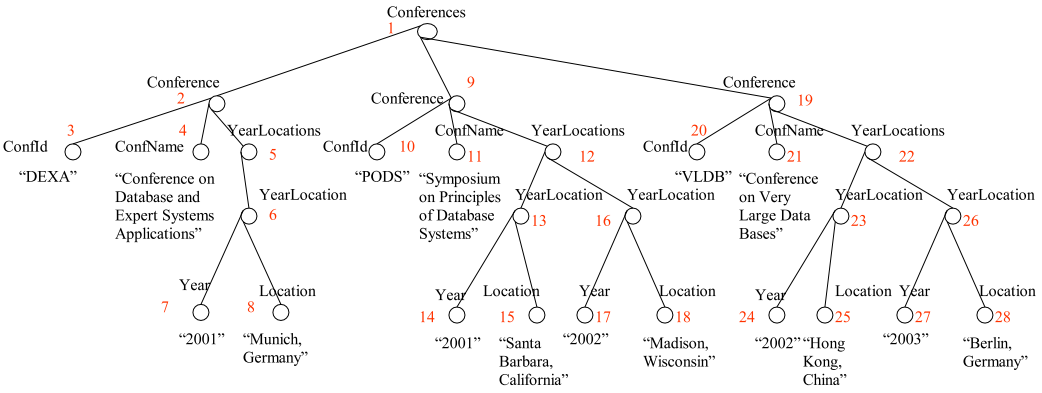

Figure 3: View 1 in XML

`<"NEW",2003,"LocName","Title">` on this view because we do not have the primary key of the Papers relation.

However, for XML views defined by general nested relational algebra (NRA) queries little can be said about updatability. Some general NRA queries can be rewritten to nest-last relational form using the rewrite techniques of [27]. However, there are others that cannot be rewritten and these remain an open problem. We will therefore focus in this paper on views produced by nest-last relational queries, and well-nested project-select-join queries. We further assume that the underlying relational database is well designed and is in BCNF.

The rest of this paper is organized as follows. In section 2 we define what it means for a view to be updatable, and summarize results from the relational case. Section 3 formalizes the notion of an update to an XML view, and discusses results on updates to views defined by nest-last relational queries as well as the special case of well-nested project-select-join queries. Section 4 concludes and presents future directions.

2 Updating relational views

A large amount of work has been done on updates through relational views [15, 14, 23], and several different techniques have been proposed. We summarize them below:

1. View as an abstract data type: In this approach [25, 28], the DBA defines the view together with the updates it supports. The effect of updates on the base relations is explicitly defined.
2. Automatic translations for view updates: In [18], Keller defines five criteria that the translations should respect in order to be correct. Dayal and Bernstein [10] propose a translation mechanism that uses view graphs to decide if a given update translation is correct. The view graphs are constructed based on the syntax of the view definition and on the functional dependencies of base relations. A more recent work is presented in [21], but it does not consider views involving selections.
3. View complement: Bancilhon and Spyratos [4] introduce the notion of view complement to solve the update problem. In this approach, an update translation is considered correct if the complement of the view remains unchanged. Finding a view complement may be NP-complete even for very simple view definitions [7].
4. Views as conditional tables: A more recent technique consists in transforming a view update problem into a Constraint Satisfaction problem [26]. In this approach, views are represented as conditional tables. Each solution to the constraint satisfaction problem corresponds to a possible translation of the view update.
5. Object-based views: An extension of [20] to deal with object-based views is proposed in [5]. In this work, Barsalou propose algorithms for propagating updates in a hierarchical structure of objects. An implementation of object-based views is discussed in the Penguin Project [19].

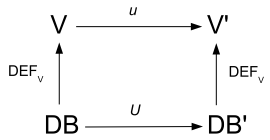

Figure 4: U exactly performs u. View V is defined over database DB using the definition function DEF_V

In this paper, we follow the second approach and attempt to find an automatic translation for XML view updates. In particular, given an update against an XML view V, we wish to find a set of updates against the base relations defining V that does not cause additional side effects in the view. For example, if we were to modify the ConfName of the DEXA conference in the table of figure 2(a) to "NewConf", we would expect the ConfName of PODS or VLDB to remain the same. That is, a translation of this modification to the following SQL update would be incorrect:

```
UPDATE Conferences SET ConfName="NewConf"
```

However, in this case there is a side effect free translation of the view update as follows:

```
UPDATE Conferences SET ConfName="NewConf"
WHERE ConfId="DEXA"
```

DEFINITION 2.1 *A translation U of an update u over a view V is* exact *when the diagram of figure 4 commutes [18].*

Having presented the meaning of *exact* translations, we can now define the notion of view updatability.

DEFINITION 2.2 *A view is updatable with respect to a type of update operation (namely insertion, deletion or modification) if there is an exact translation t for every update u of this type that can be applied over the view, whenever u is syntactically correct and does not violate the semantic consistency of the underlying database.*

This definition establishes two important points. First, view updatability depends on both the structure of the view and on the kind of update operation that is being applied on the view. As an example, a view can be updatable regarding insertions, but not modifications or deletions. Second, update operations must not cause side effects [18, 10].

We now investigate in more details the Dayal and Bernstein's approach, which we use to prove the updatability of a certain class of XML views in section 3.3.

2.1 The view dependency graph approach

Dayal and Bernstein [9, 10, 11] use view graphs to prove the updatability of a certain view. They defined two graphs. The first one captures the structure of the view and the second one captures the functional dependencies of the underlying relational database.

A view $V(Z)$, with $Z = \{D_1, \ldots, D_n\}$ is defined as (notation adapted from [14]):

```
CREATE VIEW V(D1, ..., Dn)
   AS SELECT ti1.Aij1, ..., tin.Aijn
      FROM R1 t1, ..., Rm tm
      WHERE <qual>
```

where `<qual>` contains only clauses of the form "$t.A = c$" or "$t.A = u.B$" (t and u may be the same). We say that the attribute A_{ijk} *generates* the view attribute D_k, for each k in $[1, n]$. We call Rels(V) the set $\{R_1, \ldots, R_m\}$, which is the set of relations over which V is defined.

The structure of the view definition is captured by defining a labelled directed graph $G(V)$, called *View Trace Graph*, constructed as follows:

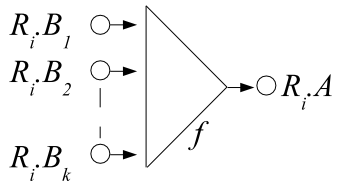

Figure 5: FD-node f corresponding to the FD f: $B_1, \ldots, B_k \rightarrow_{R_i} A$

1. For every attribute A of each relation name R_i occurring in the FROM clause, there is a node in the graph labelled "$R_i.A$";
2. For every view attribute D_i there is a node labelled $V.D_i$;
3. For each view attribute D_k there are arcs ($V.D_k$, $R_{i_k}.A_{i_{jk}}$) and ($R_{i_k}.A_{i_{jk}}$, $V.D_k$), if $A_{i_{jk}}$ generates D_k;
4. For every clause ($t_i.A = t_j.B$) in <qual> there are arcs ($R_i.A$, $R_j.B$) and ($R_j.B$, $R_i.A$);
5. For every clause ($t_i.A = c$) in <qual> introduce a node labelled c (called a constant node) and arcs (c, $R_i.A$) and ($R_i.A$, c).

If $G(V)$ has a path from node D to node $R_i.A$, then we say $R_i.A$ is *traceable* from V, and D is a *V-trace* for $R_i.A$.

The *View Dependency Graph* ($F(V)$) is obtained by enriching $G(V)$ with the information provided by the functional dependencies in the database.

1. For every FD f: $B_1, \ldots, B_k \rightarrow_{R_i} A$, add the FD-node f and the arcs of Figure 5. If A and B are singletons, then for convenience we denote the FD f: $B \rightarrow_{R_i} A$ by a single arc ($R_i.B$, $R_i.A$).
2. If $R_i.A$ is traceable from a constant node c, then for every attribute B of every relation R_i in the FROM clause, draw arc ($R_j.B$, $R_i.A$).

Having defined view graphs, it is necessary to define paths on view graphs.

Let A, B, B_1, B_2, ..., B_n be nodes and W, Y and Z be sets of nodes in $F(V)$. A *path* in $F(V)$ is defined as follows:

- There is a path from every node to itself;
- If there is an arc (B, A), then there is a path from B to A;
- Let f be an FD-node representing f: $B_1, \ldots, B_k \rightarrow_{R_i} A$. If there is a path from Y to every B_j, $1 \leq j \leq k$, then there is a path from Y to A (see Figure 6).
- If there is a path from a subset of Y to A, then there is a path from Y to A.
- If there is a path from Y to every node in Z, then there is a path from Y to Z.
- If there is a path from Y to Z, and a path from Z to W, then there is a path from Y to W.

The notation $Y \rightarrow_V Z$ says that there is a path in $F(V)$ from Y to Z.

In the following sections we analyze each type of update operation (delete, insert and modification) over a view, and give the translation procedure to each one of them based on the view graph an paths.

2.1.1 Deletions

Let V be a view. A *simple deletion* on V is a deletion $u(Y)$ of the form:

```
DELETE
FROM V v
WHERE v.D1 = c1 AND ... AND v.Dk = ck
```

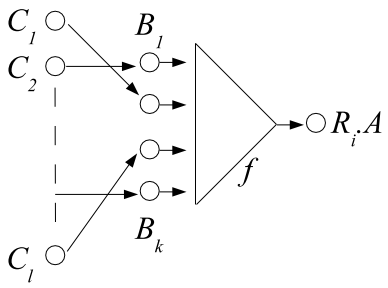

Figure 6: A path from $Y = \{C_1, \ldots, C_n\}$ to A through FD-node f

where Y is the set of the view columns D_i specified in the condition on the WHERE clause.
The translation procedure for deletions is:

Step 1 Choose one relation name R_i occurring in the FROM clause of V.

Step 2 The translation of the deletion will be

```
DELETE
FROM Ri
WHERE Ri.Ki IN
      (SELECT ti.Ki
       FROM R1 t1, ..., Rm tm
       WHERE <qual>
             AND SOURCE(D1) = c1 AND ... AND SOURCE(Dk) = ck)
```

where K_i is the primary key of R_i, and *SOURCE*(D_i) denotes the expression $R_q.A_{i_q}$ if A_{i_q} generates D_j.

Dayal proves that this procedure always exactly translates $u(Y)$ to the underlying relational database iff $X_i \rightarrow_V Y$, where $\mathbf{R_i}(X_i)$ is the relation scheme chosen in Step 1.

2.1.2 Insertions

In Dayal's approach, insertions can have a particular behavior when dealing with NULL values in view tuples. Dayal's translation mechanism works in a way that, if there is a tuple v in the view that has NULL values in some of its attributes, and the insert operation is trying to insert a tuple v' that is the same as v, but has some non-null values in some of the attributes that v has NULLs (that is, v' is more defined than v), then remove v and insert v'. This procedure is called "reduce".

Let u be an insertion of a tuple v on a view V.

```
INSERT INTO V (D1, ..., Dn)
VALUES (c1, ..., cn)
```

This insertion is translated as follows.

Step 1 For each relation $R_i \in$ Rels(V) (recall that Rels(V) is the set of relations over which view V is defined), define a tuple t_i to be inserted in R_i as.

```
INSERT INTO Ri (A1,...,Ak)
(FindValue(A1), ..., FindValue(Ak))
```

Where the procedure `FindValue(A)` is as follows:

```
function FindValue(A)
begin
    if A has a V-trace v.D AND v[D] <> NULL
    then return(v[D])
    else if (Exists an FD L → A in R_i AND
            Exists t_i' in R_i AND
            t_i'[L] = t_i[L] AND
            t_i'[A] <> NULL)
         then return t_i'[A]
         else if Exists R_j.B, such as there is a path from R_j.B to R_i.A in G(V)
              then return(t_j[B])
              else return NULL;
end;
```

Assuming that u specifies non-NULL values for the V-traces of the primary keys of each $R_i \in$ Rels(V), and that the insertion of t_i does not violate any FD in R_i, this procedure will produce exact translations for u if the following conditions are satisfied:

1. the primary key of each $R_i \in Rels(V)$ must be traceable from V; and
2. The definition of V can be expressed as a sequence of definitions of views $V_1, \ldots, V_k$, where each $V_i(Z_i)$ is defined over two base relations $R(X), S(Y)$, such that
   (a) $X \rightarrow_{V_i} Z_i$ and
   (b) $Y \rightarrow_{V_i} Z_i$ or R and S are equijoined on A, B respectively and $R[A] \subseteq S[B]$ and $B \rightarrow Y$ in S and EXISTS$(S.B, S.Y)$[1].

2.1.3 Modifications

Let u be a replacement on a view V. Let W be the set of view attributes specified for replacement in u. Let Y be the set of attributes specified in the qualification of u.

```
UPDATE V
SET W_i = r_i
WHERE Y_i = v_i
```

Let TLRels(u) be the set of relations R_i which has some attribute with a V-trace $D \in W$.
The translation procedure is as follows:

Step 1 For each relation $R_i \in$ TLRels(u), do.

```
UPDATE Ri
SET SOURCE(Wi) = ri
WHERE R_i.K_i IN
     (SELECT t_i.K_i
      FROM R_1 t_1, ..., R_m t_m
      WHERE <qual>
      AND SOURCE(Yi) = vi)
```

where K_i is the primary key of R_i, and *SOURCE*(D_i) denotes the expression $R_q.A_{i_q}$ if A_{i_q} generates D_j. Dayal [10] proves that this procedure will always exactly translate the replacement u iff

1. For all $R_i(X_i) \in$ TLRels(u), there is a path $X_i \rightarrow_V Y$; and

[1] EXISTS$(S.B, S.Y)$ is a constraint which states that for every tuple s in S, if $s[B]$ is non-NULL, then every component of $S[Y]$ is non-NULL.

2. for all $D \in W$, if D is a V-trace of some $R_i.A$ that appears in a join clause in the view qualification, then there is a path $X_i \rightarrow_V Z$, where Z is the set of attributes in the view V.

Also. u must not set to NULL a V-trace of any primary key attribute of any relation name R_i occurring in the FROM clause of V.

Having presented Dayal and Bernstein's approach for exactly translating relational view updates to database updates we now investigate the specific problem of updating relational databases through XML views. Our solution uses the results of Dayal and Bernstein presented in this section.

3 Updating XML views

Our model of XML updates is very simple, and allows the insertion of a subtree at a given node, the deletion of the subtree rooted at a given node, or the modification of a node.

DEFINITION 3.1 *An update operation u over an XML view V is a tuple <u,*ref*,Δ>, where u is the type of operation (insert, delete, modify);* ref *is the address of a node in the XML tree; and Δ is the XML tree to be inserted, or (in case of a modification) an atomic value. Deletions do not need to specify a Δ, since all the nodes under* ref *will be deleted.*

The reference *ref* can be obtained by an addressing scheme such as DOM. In our examples, we use the node numbering shown in figure 3.

Since we are considering XML views of relational databases, not all XML updates will be valid since the schema of the XML view is fixed by the view definition. Therefore, updates must *respect* the schema and be *nesting compliant*.

EXAMPLE 3.1 Suppose we wish to insert a new conference location for the DEXA conference in the XML view of figure 3. In this case, we would have:

u = *insertion*,
ref=5,

```
Δ = {<YearLocation>
        <Year>2002</Year>
        <Location>Aix en Provence, France</Location>
     </YearLocation>}.
```

Note that the insertion respects the schema of figure 2(a).

On the other hand, the following insertion does not respect the schema.

EXAMPLE 3.2 Suppose we wish to add the fact that Adam Clark was General Chair of DEXA in 2001.

u = *insertion*,
ref = 6,

```
Δ = {<GeneralChair>Adam Clark</GeneralChair>}.
```

As an example of a nesting violation, consider the following.

EXAMPLE 3.3 Suppose we insert information about DEXA 2003 at the root as follows:

u = *insertion*,
ref = 1,

```
Δ = {<Conference>
        <ConfId>DEXA</ConfId>
        <ConfName>Conference on Database and Expert Systems Applications</ConfName>
        <YearLocations>
           <YearLocation>
              <Year>2003</Year>
              <Location>Prague, Czech Republic</Location>
           </YearLocation>
        </YearLocations>
     </Conference>}.
```

This violates nesting since the resulting view has two tuples that represent DEXA. If this update were translated to a relational update and the view reconstructed, the resulting view would be different since it would have only one tuple representing the DEXA conference.

Deletions and modification are somewhat simpler, and are allowed as long as they do not violate the semantics (e.g. key, foreign key and non-null constraints) of the underlying database or the schema of the nested relational view.

EXAMPLE 3.4 Delete the subtree that has information about the location of VLDB 2002.

u = *deletion*,
ref = 23,
$\Delta = \{\}$.

EXAMPLE 3.5 Modify the name of the conference VLDB.

u = *modification*,
ref = 21,
Δ = {"New VLDB name"}.

In this example, *ref* points to a text node. Modifications are allowed only on leaves of the XML tree (text nodes).

3.1 Nest-last XML views

We now consider a class of XML views for which exact updates can be automatically translated.

DEFINITION 3.2 *A nest-last view is a view defined by a nested relational algebra expression (NRA) with the form*

$$\nu_{B_1=(X_1)}...\nu_{B_n=(X_n)}(\pi_{(A_1,A_2,...,A_k)}(\sigma_{(<qual>)}(R_1 \ominus R_2 \ominus \ldots \ominus R_m)))$$

where ν stands for the nest *operator; $\ominus$ is one of the binary operators of the classical relational algebra (union, intersection, difference, cartesian product, or equi-join); $A_1, A_2, \ldots, A_k$ are attributes of relations $R_1, R_2, \ldots, R_m$; <$qual$> is set of qualifications over the attributes of $R_1, R_2, \ldots, R_m$; and $X_i \subseteq \{A_1, \ldots, A_k\}$, $i = 1, 2, \ldots, n$.*

For shorthand, we represent a nest-last view as $\nu \ldots \nu R$, where R is any relational algebra expression.

We claim that this class of views can be treated by considering only the expression R, and that the nesting introduces *sets* of tuples to be inserted, deleted or modified in the underlying relational instance. Examples of this translation will be given in section 3.2.

CLAIM 3.1 *Let $\nu \ldots \nu R$ be a nest-last view and u an update over this view. Let $t(u)$ be the translation of u into an update over R. If R is updatable wrt $t(u)$, then $\nu \ldots \nu R$ is updatable wrt to u.*

Proof: The proof is based on the fact that the nest (ν) operator is invertible [17, 27]. That is, after a nest operation, it is always possible to obtain the original relation by applying an unnest (μ) operation. Since in this type of view the nest operation is always the last operation to be applied, we can apply a reverse sequence of unnest operators to obtain the (flat) relational expression. ■

As an example, by unnesting on YearLocation in view 1, we would obtain a flat relational expression:

$$\pi_{(IdConf,\ ConfName,\ Year,\ Location)}(Conferences \bowtie ConfLocation)$$

Claim 3.1 reduces the problem of investigating updatability of XML views to the problem of updates through relational views. Consequently, it is possible to use all the work in relational views for XML views of this class.

3.2 Translating XML updates into relational view updates

For nest-last views, we can translate XML updates into updates to the corresponding relational (flat) view. This section briefly introduces our technique based on examples.

Insertions. We unnest the subtree specified in Δ and create one relational tuple for each corresponding unnested tuple to be inserted into the relational view. If there is any missing information, we fill it in with information collected from the leaves under the elements along the path from *ref* to the root of the XML tree. In the case of example 3.1, the insertion would be translated to an insertion in the relational component of view 1 (V1) as:

```
INSERT INTO VIEW V1 (IdConf,ConfName,Year,Location)
VALUES ("DEXA", "Conference on Database and Expert Systems Applications",
         2002, "Aix en Provence,France")
```

As another example, suppose we insert a new conference with no information about YearLocations. This would be translated as:

```
INSERT INTO VIEW V1 (IdConf,ConfName,Year,Location)
VALUES ("NEW", "New Conference", NULL, NULL)
```

Insertions may also be translated to a *set* of insertions in the relational view. As an example, consider the insertion of the following subtree at *ref* = 1.

```
Δ = {<Conference>
          <IdConf>ER</IdConf>
          <ConfName>Conference on Conceptual Modeling</ConfName>
          <YearLocations>
             <YearLocation>
                <Year>2002</Year>
                <Location>Tampere, Finland</Location>
             </YearLocation>
             <YearLocation>
                <Year>2003</Year>
                <Location>Chicago, Illinois</Location>
             </YearLocation>
          </YearLocations>
     </Conference>}
```

This would be translated to

```
INSERT INTO VIEW V1 (IdConf,ConfName,Year,Location)
VALUES ("ER", "Conference on Conceptual Modeling",
        2002, "Tampere, Finland")

INSERT INTO VIEW V1 (IdConf,ConfName,Year,Location)
VALUES ("ER", "Conference on Conceptual Modeling",
        2003, "Chicago, Illinois")
```

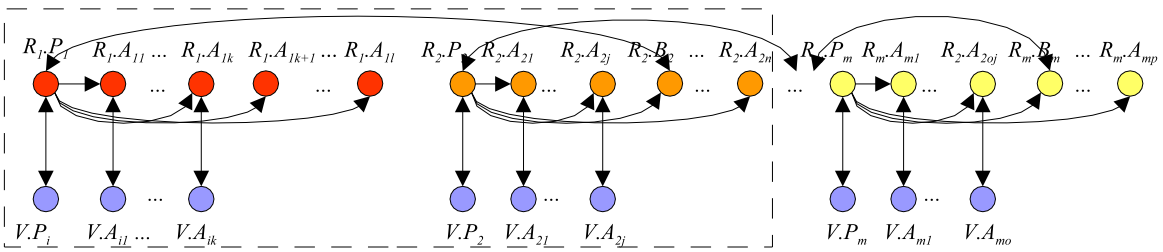

Figure 7: View graph

Deletions. Deletions are translated in a similar way. To build the DELETE SQL statement, we use the subtree of information rooted at *ref* as well as information collected along the path from *ref* to the way to the document root. Each value found in this path becomes a condition in the WHERE clause of the deletion.

In the case of example 3.4, we would translate it using the information of the node being deleted as well as its parent (in this case, the VLDB conference). The translation would be:

```
DELETE FROM VIEW V1 WHERE Year=2002 AND
Location= "Hong Kong, China" AND IdConf= "VLDB" AND
ConfName= "Conference on Very Large Data Bases"
```

A deletion can also affect more than one tuple in the relational view. An example would be the attempt to delete node *ref* = 9. This would be translated to:

```
DELETE FROM VIEW V1 WHERE IdConf= "PODS"  AND
ConfName= "Symposium on Principles of Database Systems"
```

Modifications. Modifications are treated in the same way as deletions. That is, we use information about the node and its ancestors to build the WHERE clause. In the case of example 3.5, the translation is:

```
UPDATE VIEW V1
WHERE IdConf= "VLDB"  AND
ConfName= "Conference on Very Large Data Bases"
SET ConfName= "New VLDB name"
```

We have shown how to translate updates over an XML view to updates over the corresponding relational view. The techniques of [18, 10] can then be used to translate these updates to the underlying relational database.

3.3 Nest-last Project-Select-Join Views

We now investigate a special subset of nest-last views that are well behaved with respect to updates.

DEFINITION 3.3 *A* nest-last project-select-join view *(NPSJ) is a nest-last view with the following restrictions: the relational expression is a project-select-join; the keys of the base relations are not projected out; and joins are made only through foreign keys.*

LEMMA 3.1 *NPSJ views are always updatable for insertions.*

Proof: Claim 3.1 shows how to reduce an XML view to a relational view. Based on this result, we are now able to use the technique of Dayal and Bernstein [10] to prove that there is always an exact translation for insertions and deletions for NPSJ views. Since the nest can be ignored, we start by defining a general PSJ view that is the join of relations $R_1, R_2, \ldots, R_m$, where the the keys of $R_1, R_2, \ldots, R_m$ are preserved in the view and joins are done over foreign keys.

Let a project, select, join view $PSJ(Z)$ be defined as:

$$\pi_{(P_1,P_2,...,P_m,A_1,A_2,...,A_k)} \\ (\sigma_{(<qual>)} \\ (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m))$$

where $P_1, P_2, \ldots, P_m$ are the primary keys of $R_1, R_2, \ldots, R_m$, respectively; Z = {P_1, P_2, ..., P_m, A_1, A_2, ..., A_k} are attributes of the relations $R_1, R_2, \ldots, R_m$; and <$qual$> is set of qualifications over the attributes of $R_1, R_2, \ldots, R_m$.

We then draw a view graph for this view, as illustrated in figure 7. Nodes in this graph represent attributes. The upper nodes represent attributes of the base relations, and the lower ones represent view attributes. Primary keys are represented as Ps and foreign keys as Bs. As seen in section 2.1, the proofs for insertions are based on finding paths in this directed graph.

Dayal and Bernstein [11] claim that insertions are always exactly translatable if we can express the view definition as a sequence of views definitions, each one defined over only two relations. Let's review the conditions that must be satisfied in order to exist an exact translation for a given insertion u.

1. The primary key of each $R_i \in \{R_1, R_2, \ldots, R_m\}$ must be traceable from PSJ; and
2. The definition of PSJ can be expressed as a sequence of definitions of views $V_1, \ldots, V_k$, where each $V_i(Z_i)$ is defined over two base relations $R(X), S(Y)$, such that
   (a) $X \rightarrow_{V_i} Z_i$ and
   (b) $Y \rightarrow_{V_i} Z_i$ or R and S are equijoined on A, B respectively and $R[A] \subseteq S[B]$ and $B \rightarrow Y$ in S and $EXISTS(S.B, S.Y)$.

CONDITION 1: Holds from the definition of PSJ.

CONDITION 2: Also holds from the definition of PSJ. Since PSJ is defined as projections and selections over ($R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$), it is possible to define PSJ as V_i, with $1 \leqslant i \leqslant (m-1)$, where V_i is

$$\pi_{(P_i,P_{i+1},A_{i_1},A_{i_2},...,A_{i_k},A_{(i+1)_1},A_{(i+1)_2},...,A_{(i+1)_l})} \\ (\sigma_{(<qual_i>)} \\ (R_i \bowtie R_{i+1}))$$

where P_i, P_{i+1} are the primary keys of R_i, R_{i+1}, respectively; Z_i = {P_i, P_{i+1},A_{i_1}, A_{i_2}, ..., A_{i_k}, $A_{(i+1)_1}$, $A_{(i+1)_2}$, ..., $A_{(i+1)_l}$} is the set of all attributes of the relations R_i, R_{i+1} that appears in Z (the set of attributes of PSJ); $\{P_i, A_{i_1}, A_{i_2}, \ldots, A_{i_k}\} \subseteq R_i(X_i)$, {$P_{i+1}$,$A_{(i+1)_1}$, $A_{(i+1)_2}$, ..., $A_{(i+1)_l}$} $\subseteq R_{i+1}(X_{i+1})$;and <$qual_i$> is set of qualifications over the attributes of R_i, R_{i+1} that appears in <$qual$>. In the same way as PSJ, V_i is joined on the foreign keys of R_i and R_{i+1}. So, one attribute of R_{i+1} is the attribute that implements this foreign key constraint. Let's say this attribute is B_{i+1}.

CONDITION 2A: This condition requires us to check the view dependency graph for view V_i, which is shown on figure 8 (additionally, view V_1 is shown on the doted box of figure 7). By analyzing the view dependency graph, it is easy to see that condition 2a is satisfied: $X \rightarrow_{V_i} Z_i$ holds for V_i when we make $R(X) = R_{i+1}(X_{i+1})$. Thus, $X_{i+1} \rightarrow_{V_i} Z_i$. As an additional example, consider the two relations inside the dotted box in figure 7. There is a path from the attributes of R_2 to all attributes in the view that originated from R_1 or R_2.

CONDITION 2B: Condition 2b is an OR of two sub-conditions. The first sub-condition is not true. Since we made $R(X) = R_{i+1}(X_{i+1})$ in the proof of condition 2a, we now have to make $S(Y) = R_i(X_i)$, and $X_i \rightarrow_{V_i} Z_i$ obviously does not hold (as we can see by analyzing the view graph of Figure 8. Consequently, the only way for condition 2b to be truth is that the second sub-condition holds. We will prove that this is indeed the case.

The second sub-condition is an AND of 3 clauses. Let's analyze each one of them:

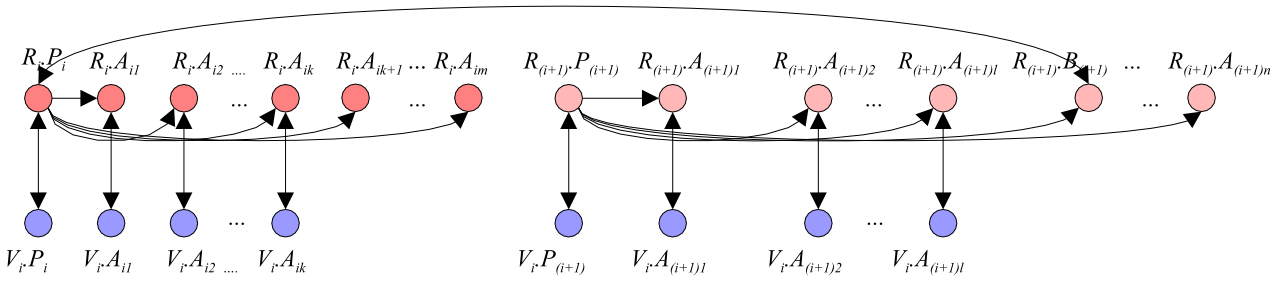

Figure 8: View Dependency graph of V_i

- R and S are equijoined on A, B respectively and $R[A] \subseteq S[B]$:
  R_{i+1} and R_i are equijoined through attributes $R_{i+1}.B_{i+1}$ and $R_i.P_i$ by definition of V_i. The condition $R_{i+1}[B_{i+1}] \subseteq R_i[P_i]$ trivially holds from the definition of foreign key ($R_{i+1}.B_{i+1}$ is a foreign key that references $R_i.P_i$).
- $B \rightarrow Y$ in S:
  As we are using $S = R_i$, we have to prove that $P_i \rightarrow X_i$. This holds from the fact that P_i is the primary key of R_i by definition.
- $EXISTS(S.B, S.Y)$:
  This is only required so that $P_i \rightarrow X_i$ is not true only for the trivial case where all the values in X_i are NULL. In fact, this is not the case. $P_i \rightarrow X_i$ holds by the definition of primary keys.

As all the three clauses are true, we can conclude that the second sub-condition of condition 2b is true, which makes the hole condition true as well. Thus, PSJ satisfies all the conditions for the existence of exact translations for insertions and deletion, as we wanted to show. ■

For modifications and deletions, even in the relational case there may fail to be an exact translation for certain types of updates over a PSJ view. This type of update attempts to change (or delete) some but not all occurrences of data that is repeated in the view, and thus causes side effects. As an example, consider the unnested version of the view 1. This view has the values of IdConf and ConfName repeated in several tuples. An attempt to modify a conference name could be stated as

```
UPDATE VIEW V1
SET ConfName= "New Name"  WHERE IdConf= "VLDB"
```

This is exact, since it modifies all occurrences of VLDB tuples. However, consider this same example with a slight modification.

```
UPDATE VIEW V1 SET ConfName= "New Name"
WHERE IdConf= "VLDB" AND Year=2002
```

As one can easily see, there is no way to translate this request without causing side effects, because a tuple that does not satisfy the qualification of this modification request would also be affected (more specifically, the tuple with IdConf="VLDB" and Year=2003). The same problem happens for deletions.

Fortunately, proper application of the nest operator can be used to avoid this type of ambiguity. For example, for the view shown in figure 3 this kind of bad modification (or deletion) request cannot happen. Recall the translation of the modification update example 3.5, which translates the modification to update all VLDB tuples.

However, if we had nested this view in a different way, the same update would fail to be exact. As an example, consider the same view, now nested by {*IdConf, ConfName*} instead of {*Year,Location*}. The same IdConf and ConfName appear several times in the view, as in the relational case. Thus, not all modifications and deletions over this view would be exactly translatable.

The updatability of NPSJ views with respect to modifications and deletions depends on the way in which we traverse the foreign key constraints when nesting. In view 1, we traverse the foreign key constraint from 1

to n. That is, for each Conference tuple there are many ConfLocation tuples, so we nest ConfLocation tuples (the n's) under their corresponding Conference tuple (the 1's). In the second example (where we nested over {*IdConf, ConfName*}), we nested the 1's under the n's, causing the 1's to appear several times in the resulting view.

To define when a NPSJ view is well-nested, we reason about the foreign keys of the underlying relations. Recall that the syntax of a foreign key constraint C on table R_1 is given by C_{R_1} FOREIGN KEY ($FK_1, \ldots, FK_n$) REFERENCES R_2 ($K_1, \ldots, K_n$). When the attribute names ($K_1, \ldots, K_n$) are the same as ($FK_1, \ldots, FK_n$), they can be ommited, as in the example of figure 1.

DEFINITION 3.4 *Let C_{R_1} be a foreign key constraint, and $V(R_1)$ be the set of attributes of R_1 that appear in the view. An* ambiguity eliminating nest *with respect to C_{R_1} is a nest of the form $\nu_{X=(D)}$, where $D = \{V(R_1)\} - \cup_i FK_i$.*

The idea behind this definition is that by omitting the foreign keys of R_1 and the keys of R_2 in the nest, we collect their values together thus eliminating ambiguity. That is, each value appears just once in the view.

The view of example 1 has an ambiguity eliminating nest since R_1=*ConfLocation*, R_2= *Conferences*, FK={*IdConf*}, $V(R_1)$={*IdConf, Year, Location*} and we are nesting over $\nu_{YearLocations=(Year,Location)}$.

DEFINITION 3.5 *A NPSJ view that involves more than two base relations is* well nested *if*

1. *It has one ambiguity eliminating nest for each foreign key constraint that was used to join the base relations; and*
2. *The nests are executed in the opposite order of the joins.*

An example of well-nested NPSJ is view 1. Another example is given by the following NRA expression:

$$\nu_{Papers=(IdPaper,Title)}(\nu_{YearLocations=(Year,Location)}$$
$$(\pi_{(IdConf,\ ConfName,\ Year,\ Location,\ IdPaper,\ Title)}$$
$$(Conference \bowtie (ConfLocation \bowtie Papers))))$$

This expression differs from previous examples because it contains two nested relations in the same nesting level. The resulting view has the following structure: (*IdConf, ConfName,* {*YearLocations*}, {*Papers*}), were *YearLocations* and *Papers* are nested relations. This example shows that NPSJ views are capable of expressing complex structures.

LEMMA 3.2 *Well nested NPSJ views are always updatable with respect to modifications and deletions.*

Proof: We divide the proof in two steps, one for deletions and one for modifications.

Modifications. In order to simplify the proof, we consider a view defined over two base relations, say $R_1 \bowtie R_2$. The graph of this view corresponds to the dotted box of figure 7. Using the technique of [10], there must be a path from the attributes of the relation whose attributes are being modified to all view attributes that were specified in the WHERE clause. In the case of well-nested NPSJ views, this is directly related to how we specify the update against the relational view. In order for R_1 and R_2 to be well-nested, R_2 must be nested under R_1. If we want to modify an attribute from R_1, the WHERE clause will have only attributes generated from R_1. Obviously, there is a path from the attributes in R_1 to the view attributes generated from R_1. If we want to modify attributes from R_2, the WHERE clause will have attributes generated both from R_1 and R_2. Since it is possible to use the arrow $R_1.P_1$-$R_2.B_2$ to reach all the view attributes, the condition is satisfied. The proof can be easily generalized to views defined over more than two base relations.

Deletions. Deletions have a WHERE clause that specifies conditions that view tuples must satisfy in order to be deleted. The condition for exact translation for deletions says that there must be a path in the view graph

from the relation R_i chosen to translate this deletion to all attributes specified in the WHERE clause. We call this set of attributes Y. So, $X_i \rightarrow_V Y$ must hold, where X_i is the set of attributes of the relation R_i. Our proof supposes that all attributes of the view were specified in the WHERE clause, since this is the "worst case". More specifically, we make $Y = Z$. It is easy to see that one can always choose the last relation joined to translate the deletion to the database because there is always a path from the attributes on this relation to all view attributes (see R_m in figure 7) due to the edges introduced by join conditions. ■

4 Conclusions

We have investigated the problem of how to translate updates on XML views over relational databases to updates on the underlying relations. In particular, we showed how updates to a nest-last view can be translated to updates on the corresponding relational view. Techniques from the relational model can then be used to determine if the nest-last XML view is updatable for a given update.

For the special class of NPSJ views, we showed that it is always possible to find exact translations for insertions. When these views are well-nested it is also possible to find exact translations for deletions and modifications. Thus, well-nested NPSJ views are updatable for all valid updates.

Well-nested NPSJ views are a very significant class of XML views. If we store an XML view of this class in a relational database exploiting the keys and semantic constraints of the document, we would be able to reconstruct the XML view using only joins over foreign keys [6]. That is, the relational instance represents a natural storage scheme for the XML view when constraints are taken into account.

Since our focus was on XML views of legacy relational databases rather than XML views of XML documents, it was reasonable to make some simplifications. First, the schema of the view was fixed which meant that limited forms of insertions and deletions were allowed. Second, it was sufficient to consider the nested relational algebra as the basis of view expressions rather than something like the XQuery algebra.

The XQuery algebra [12] expresses all the operators of NRA, as well as aggregation, quantification, sorting and iteration. It also has operators to deal with XML specific features - ordering, comments, processing instructions. It is clear that since aggregation loses information, views involving aggregation will not be updatable [20]. Furthermore, operators involving ordering are not relevant when the underlying representation is relational.

We claim that NRA is general enough to be able to represent the same type of structures as object-based views [5]. In particular, object-based views include only relations that are related by integrity constraints, and can therefore be expressed as nest-last views. The main difference between object-based views and our approach based on NRA is related to side effects. Object views can be formed by creating relationships (pointers) between simple objects and may therefore avoid repeating information. For example, a view can be defined as a set of objects representing papers, where each paper is connected to an object that represents the conference in which the paper was published. Information about conferences is not repeated, as it would be in the corresponding NRA view. Thus, changing the name of a conference would affect a single object, which is referenced by several papers, and would be side effect free. Note that by considering `ID` and `IDREF` in XML and using “normalized” XML views [3] we can achieve the same result.

In future work we plan to explore general XML views.

Acknowledgments. We would like to thank Capes for supporting this research (BEX 1123/02-5).

References

[1] Xml for molecular biology as compiled by paul gordon. http://www.visualgenomics.ca/gordonp/xml.

[2] ABITEBOUL, S., AND BIDOIT, N. Non first normal form relations to represent hierarchically organized data. In *PODS* (1984), pp. 191–200.

[3] ARENAS, M., AND LIBKIN, L. A normal form for XML documents. In *Proceedings of PODS 2002* (Madison, Wisconsin, Jun 2002).

[4] BANCILHON, F., AND SPYRATOS, N. Update semantics of relational views. *ACM TODS 6*, 4 (Dec 1981).

[5] BARSALOU, T., SIAMBELA, N., KELLER, A. M., AND WIEDERHOLD, G. Updating relational databases through object-based views. In *SIGMOD* (Denver, CO, 1991), pp. 248–257.

[6] CHEN, Y., DAVIDSON, S. B., AND ZHENG, Y. 3XNF: Redundancy eliminating XML storage in relations. In *VLDB* (Berlin, Germany, 2003).

[7] COSMADAKIS, S. S., AND PAPADIMITRIOU, C. H. Updates of relational views. *Journal of the Association for Computing Machinery 31*, 4 (Oct 1984), 742–760.

[8] DA SILVA, A. S., FILHA, I. M. E., LAENDER, A. H. F., AND EMBLEY, D. W. Using nested tables for representing and querying semistructured data. In *Proceedings of ER* (Tampere, Finland, 2002).

[9] DAYAL, U., AND BERNSTEIN, P. A. On the updatability of relational views. In *Proceedings of VLDB* (West Berlin, Germany, Sep 1978), pp. 368–377.

[10] DAYAL, U., AND BERNSTEIN, P. A. On the correct translation of update operations on relational views. *ACM TODS 7*, 3 (Sep 1982), 381–416.

[11] DAYAL, U., AND BERNSTEIN, P. A. On the updatability of network views - extending relational view theory to the network model. *Information Systems 7*, 2 (1982), 29–46.

[12] FANKHAUSER, P., FERNÁNDEZ, M., MALHOTRA, A., RYS, M., SIMÉON, J., AND WADLER, P. The xml query algebra. W3C Working Draft, Feb 2001. www.w3.org/TR/2001/WD-query-algebra-20010215.

[13] FERNÁNDEZ, M., TAN, W.-C., AND SUCIU, D. Silkroute: Trading between relations and xml. In *Nineth Internation World Wide Web Conference* (2000).

[14] FURTADO, A. L., AND CASANOVA, M. A. Updating relational views. In *Query Processing in Database Systems*, W. Kim, D. S. Reiner, and D. S. Batory, Eds. Springer, Berlin, Heidelberg, 1985, pp. 127–142.

[15] FURTADO, A. L., SEVCIK, K. C., AND SANTOS, C. S. D. Permitting updates through views of data bases. *Information Systems 4*, 4 (Oct 1979), 269–283.

[16] HULIN, G. On restructuring nested relations in partitioned normal form. In *16th VLDB Conference* (Brisbane, Australia, 1990), pp. 626–636.

[17] JAESCHKE, G., AND SCHEK, H.-J. Remarks on the algebra of non first normal form relations. In *PODS* (Los Angeles, CA, March 1982), pp. 124–138.

[18] KELLER, A. M. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of PODS* (Portland, Oregon, Mar. 1985), ACM, pp. 154–163.

[19] KELLER, A. M., AND WIEDERHOLD, G. Penguin: Objects for programs, relations for persistence. In *Succeeding with Object Databases*, A. B. Chaudhri and R. Zicari, Eds. John Wiley & Sons, 2001.

[20] KELLER, M. The role of semantics in translating view updates. *IEEE Computer 19*, 1 (1986), 63–73.

[21] LANGERAK, R. View updates in relational databases with an independent scheme. *ACM TODS 15*, 1 (1990), 40–66.

[22] MAKINOUCHI, A. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of VLDB* (Tokio, Japan, 1977), pp. 447–453.

[23] MEDEIROS, C., AND TOMPA, F. Undestanding the implications of view update policies. In *13th International Conference on Very Large Databases* (1985), pp. 316–323.

[24] MOK, W. Y., NG, Y.-K., AND EMBLEY, D. W. A normal form for precisely characterizing redundancy in nested relations. *ACM TODS 21*, 1 (Mar 1996), 77–106.

[25] ROWE, L. A., AND SHOENS, K. A. Data abstraction, views and updates in rigel. In *Proceedings of SIGMOD* (Boston, Massachusetts, 1979), pp. 71–81.

[26] SHU, H. Using constraint satisfaction for view update translation. In *Proc. of European Conference on Artificial Intelligence (ECAI)* (Brighton, UK, 1998).

[27] THOMAS, S. J., AND FISCHER, P. C. Nested relational structures. *Advances in Computing Research 3* (1986), 269–307.

[28] TUCHERMAN, L., FURTADO, A. L., AND CASANOVA, M. A. A pragmatic approach to structured database design. In *Proceedings of VLDB* (Florence, Italy, Oct 1983), pp. 219–231.