UXQuery: Building Updatable XML Views over Relational Databases *

Vanessa P. Braganholo[1] vanessa@inf.ufrgs.br
Susan B. Davidson[2] susan@cis.upenn.edu
Carlos A. Heuser[1] heuser@inf.ufrgs.br

[1]Instituto de Informática - Universidade Federal do Rio Grande do Sul - Brazil
[2]Department of Computer and Information Science - University of Pennsylvania

Abstract

XML has become an important medium for data exchange, and is frequently used as an interface to – i.e. a view of – a relational database. Although much attention has been paid to the problem of querying relational databases through XML views, the problem of updating relational databases through XML views has not been addressed. In this paper we investigate how a subset of XQuery can be used to build updatable XML views, so that an update to the view can be unambiguously translated to a set of updates on the underlying relational database, assuming that certain key and foreign key constraints hold. In particular, we show how views defined in this subset of XQuery can be mapped to a set of relational views, thus transforming the problem of updating relational databases through XML views into a classical problem of updating relational databases through relational views.

1. Introduction

XML has become an important medium for data exchange, and is frequently used as an interface to – i.e. a view of – a relational database. Much attention has been paid to the problem of querying relational databases through XML views [1, 2, 3]: Given a query in some XML query language, how is the query translated to an SQL query against the relational instance and the result then manipulated to produce an XML result? However, the problem of updating the relational database through an XML view has not been addressed: Given an update to an XML view expressed in some XML update language, how is the update translated to an update on the relational instance? In particular, are there classes of XML views which are *updatable* for a given type of update (insertion, deletion or modification) in the sense that the XML update can be translated to an update on the relational instance where the only tuples affected are those who completely satisfy the update specification?

For example, consider the database of Figure 1 which contains information about authors, conferences, papers and books. An XML view of this database which groups papers published by year for each author is shown in Figure 2(a). Suppose we wish to change the title of Mary Jones's paper with id "IR", and reference this element in the update specification by using the path expression `/result/author[@id="1"]/papers/paper[@id="IR"]/title`. Since Charles Green is also a co-author of this paper, translating this update to the relational database would result in Charles Green's IR paper also being updated in the XML view. This view is therefore not updatable with respect to the given update. However, if we update the title of the paper with id "IR" using the path expression `//paper[@id="IR"]/title` (i.e. omit the author in the update path) no such side-effects would occur. Since we are not specifying the author in the update path expression, all titles of papers with id "IR" would be altered in the view, and no side effects would occur.

*Research supported by Capes (BEX 1123/02-5) as well as NSF DBI-9975206.

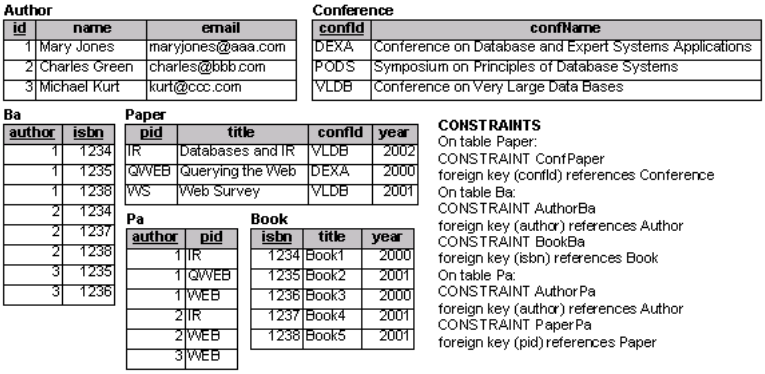

Figure 1: Sample database

In previous work [4], we addressed this problem by considering the nested relational algebra (NRA) [5] as the language defining the XML view, and showed that an NRA view can be mapped to a relational view. In doing so, we were able to build upon previous work on updates to relational views [6, 7], and map a new problem (updating relational databases through NRA views) to a well studied problem.

Although the NRA captures many essential aspects of XML, in particular the notion of tuples and nesting, it does not capture other aspects of XML, in particular the ability to create heterogeneous sets (or lists). As a simple example, consider the XML view of Figure 2(b), which lists papers *and books* published by year. Since the nested set is heterogeneous (papers and books have different attributes), this cannot be specified in the NRA. However, such a view is easily defined in standard XML query languages.

In this paper, we therefore consider a subset of XQuery [8] which allows nesting as well as heterogeneous sets, and show how updates over XML views are propagated to the underlying relational database. The key observation is that XML views with heterogeneous sets can be mapped to a *set* of nested relational views. For example, the view in Figure 2(b) can be mapped to a set consisting of the nested relational view of Figure 2(a) and its counterpart containing only book information by year for each author. Updates to such XML views can then be mapped to a set of updates to the underlying nested relational tables.

We chose XQuery as the XML query language since it is widely accepted, and is becoming somewhat of a standard. We also borrowed some ideas from SQLX [9], an extension to SQL being developed by INCITS: we use the SQLX representation for relational tables (`row`), and define an input function to XQuery called `table` to access relational tables. This function, however, is slightly different from the one proposed in SQLX.

The structure and contributions of this paper are:

1. Section 2: The definition of a subset of XQuery for extracting updatable XML views from relational databases. The subset is augmented with two new features: a function `table` to extract data from relational sources and transform tuples into a set of XML nodes, and a macro operator `xnest` to facilitate nesting. Note that `xnest` does not add anything to the language, and that queries containing `xnest` can be mapped to XQuery.
2. Section 3: A method for mapping XML views to a set of relational views. The relational views can then be used to check for XML view updatability.
3. Section 4: An overview on how updates into XML views are translated to updates on the corresponding relational views.

Related work is given in section 5, and section 6 concludes the paper with a summary and future research.

```
<result>
<author id="1">
   <name> Mary Jones </name>
   <address>
      <email> maryjones@aaa.com </email>
   </address>
   <papers year="2002">
      <paper id="IR">
         <title> Databases and IR </title>
         <confId> VLDB </confId>
      </paper>
   </papers>
   <papers year="2000">
      <paper id="QWEB">
         <title> Querying the Web </title>
         <confId> DEXA </confId>
      </paper>
   </papers>
   <papers year="2001">
      <paper id="WEB">
         <title> Web Survey </title>
         <confId> VLDB </confId>
      </paper>
   </papers>
</author>
<author id="2">
   <name> Charles Green </name>
   <address>
      <email> charels@bbn.com </email>
   </address>
   <papers year="2002">
      <paper id="IR">
         <title> Databases and IR </title>
         <confId> VLDB </confId>
      </paper>
   </papers>
   ...
</result>
```

(a)

```
<result>
<author id="1">
   <name> Mary Jones </name>
   <address>
      <email> maryjones@aaa.com </email>
   </address>
   <papers year="2002">
      <paper id="IR">
         <title> Databases and IR </title>
         <confId> VLDB </confId>
      </paper>
   </papers>
   <papers year="2000">
      <paper id="QWEB">
         <title> Querying the Web </title>
         <confId> DEXA </confId>
      </paper>
      <book isbn="1234">
         <title> Book1 </title>
      </book>
   </papers>
   <papers year="2001">
      <paper id="WEB">
         <title> Web Survey </title>
         <confId> VLDB </confId>
      </paper>
      <book isbn="1235">
         <title> Book2 </title>
      </book>
      <book isbn="1238">
         <title> Book5 </title>
      </book>
   </papers>
</author>
...
</result>
```

(b)

Figure 2: (a) Nested relational XML view (b) XML view

2. A Subset of XQuery to Build XML Views

Our goal is to find a subset of XQuery which produces updatable XML views. As shown in [4], this subset should certainly include queries which produce nested relations. However, we wish to broaden this to queries which allow multiple sets within a nested component. We call such XML views "well-behaved" in the sense that they can be mapped to a set of corresponding relational views, whose updatability can be reasoned about using established techniques.

DEFINITION 1 *A well behaved XML view is an XML tree extracted from a relational database with the following abstract type τ:*

$\tau ::= E_0 : \{E : \tau_T\}$ *(,* $\{E : \tau_T\}$ *)**

$\tau_T ::= [$ *(* $E : \tau_S$, $| E : \tau_C$, $| \{E : \tau_T\}$, *)** *(*$E : \tau_S \mid E : \tau_C$*) (* $, E : \tau_S \mid , E : \tau_C \mid , \{E : \tau_T\}$ *)** $]$

$\tau_C ::= [$ *(* $E : \tau_S \mid E : \tau_C$ *) (*$, E : \tau_S \mid , E : \tau_C$ *)** $]$

(where "[...]" denotes a tuple and "{...}" denotes a set). τ_T *denotes a tuple type,* τ_C *denotes a non-repeating complex type and* τ_S *denotes an atomic type (e.g. #PCDATA or CDATA).* E_0 *is an element name denoting the root of the document, and* E *is an element or attribute name.*

Note that well behaved views always have a root (E_0), have at least one repeating element (right under the root), and that repeating elements are always delimited by an element. We adopt the convention that attribute names start with "@".

Nodes of type τ_T whose descendants are non-repeating nodes (τ_S / τ_C) are renamed to τ_N.

For example, the view in Figure 2(b) is well behaved. We show the schema of the view below, with the abstract type of each element shown to the right. Note that the element *papers* has tuples of two different types (*paper* and *book*). Additionally, *paper* and *book* are repeating nodes whose descendants are non repeating nodes. For this reason, their types are renamed to τ_N.

```
result :                                                    (τ)
   {author:                                                 (τT)
      [@id: CDATA,                                          (τS)
       name: #PCDATA,                                       (τS)
       address:                                             (τC)
          [email: #PCDATA],                                 (τS)
       {papers:                                             (τT)
          [@year: CDATA,                                    (τS)
          {paper:                                           (τN)
              [@id: CDATA,                                  (τS)
               title: #PCDATA,                              (τS)
               confId: #PCDATA]},                           (τS)
          {book:                                            (τN)
              [@isbn: CDATA,                                (τS)
               title: #PCDATA]}                             (τS)
          ]}
      ]
   }
```

XQuery's syntax is very broad and has lots of operators. Some of these operators - such as order related operators - do not really make sense when we are producing views of relational databases in which there is no inherent order. Furthermore, aggregate operators create ambiguity when mapping a given view tuple to the underlying relational database. We will therefore ignore ordering operators and outlaw aggregate operators. This means that the use of `let` in our subset of XQuery must be very carefully controlled, and for this reason we will allow it only as expanded by a new macro called `xnest`.

The subset we have chosen is called UXQuery (*Updatable XQuery*), and contains the following:

- FWOR `for/where/order by/return` expressions (note that we do not allow `let` expressions).
- Element and attribute constructors.
- Comparison expressions.
- An input function `table`, which binds a variable to tuples of a relational table that is specified as a parameter to the function.
- A macro operator called `xnest`, which facilitates the construction of heterogeneous nested sets.

The EBNF of UXQuery is shown in Appendix A. The formal semantics of UXQuery matches the semantics of XQuery [10] with the exception of the new input function `table` and the macro `xnest`, which we discuss next.

Semantics of `table()`. XQuery has three input functions: `input`, `collection` and `document` [11]. In UXQuery, the only input function available to the user is `table`. This function takes as input a table from a relational database and returns a set of tuples in the following form:

```
<row>
   <!-- tuple attributes -->
   ...
</row>
...
```

Following SQLX [9], we translate this input function to XQuery as follows.

```
 1. <conferencePapers>
 2.   {for $c in table("conference")
 3.    return
 4.    <conference id="{$c/confId/text()}">
 5.       {$c/confName}
 6.       {xnest $p in table("paper")
 7.        by $year in ($p/year)
 8.        where $p/confId=$c/confId
 9.        return
10.        <papers year="{$year/text()}">
11.           {
12.           <paper>
13.              {$p/pid}
14.              {$p/title}
15.           </paper>
16.           }
17.        </papers>
18.       }
19.    </conference>
20.   }
21. </conferencePapers>
```

```
22. <conferencePapers>
23.  {for $c in table("conference")
24.   return
25.   <conference id="{$c/confId/text()}">
26.    {$c/confName}
27.    {let $p' := table("paper")
28.     for $year in distinct-values($p'/year)
29.     return
30.     <papers year="{$year/text()}">
31.      {for $p in table("paper")
32.       where $c/confId=$p/confId and $p/year=$year
33.       return
34.       <paper>
35.          {$p/pid}
36.          {$p/title}
37.       </paper>
38.      }
39.     </papers>
40.    }
41.   </conference>
42.  }
43. </conferencePapers>
```

Figure 3: Example of a query that uses the xnest operator (lines 1-22) and its translation to regular XQuery syntax (lines 23-45)

```
define function table($tableName as xs:string) as node*
{
   let $tuples := document(concat($tableName,".xml"))//row
   return $tuples
}
```

Semantics of `xnest`. The `xnest` operator is used to specify possibly heterogeneous sets of nested tuples that agree in the value of one or more attributes. The tuples are clustered according to the value of these attributes, which we call *nesting attributes*. A simple (non-heterogeneous) example of such a query is shown in Figure 3 (lines 1-21). The query specifies a join of tables conference and paper. For each conference, it shows the conference name, the conference Id, and the papers for that conference clustered by year. The nesting attribute in this case is *year*.

The syntax for `xnest` is defined by the following EBNF:

```
Nest       ::= NestClause ByClause WhereClause "return" Header
NestClause ::= "xnest "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
ByClause   ::= "by" "$" VarName "in" UnionExpr ("," "$" VarName "in" UnionExpr)*
Header ::= "<" QName (QName "=" NestAttValue)+ ">" ( "{" ElGroup "}" )+ "</" QName ">"
         | "<" QName ">" ( "{" "$" VarName "}" | "<" QName ">" "{" "$" VarName "/" TextTest "}"
           "</" QName ">")+ ( "{" ElGroup "}" )+ "</" QName ">"
NestAttValue ::= "'" "{" "$" VarName "/" TextTest "}" "'"
                   | '"' "{" "$" VarName "/" TextTest "}" '"'
ElGroup    ::= ElmtConstructor
UnionExpr  ::= "(" "$" VarName "/" QName ( ("union" | "|")  "$" VarName "/" QName)* ")"
```

A query containing a `xnest` operator can be normalized to one using pure XQuery syntax. The normalized query corresponding to the query in Figure 3 (lines 1-21) is shown in Figure 3 (lines 22-43). The normalization process makes sure that the nest variable (in the example, *$year*) appears in the *Header* element as an attribute or a sub-element. In the example, the *Header* element is *papers*. Notice that in the normalized query, we still use the input function `table`.

Continuing with the example, the `xnest` operation (lines 6-18) is normalized to the expression shown in lines 27-40. The expression consists of a `let/for` (lines 27-28) and an additional `for` (lines 31-38) for each *ElGroup* (lines 11-16) specified in the query. In the normalization process, we introduce new variables in the `let` clause. These variables are primed (’), and correspond to the variables specified in the `xnest` operator. There will be one primed variable in the `let` clause

```
<conferencePapers>
  {for $c in table("conference")
   return
   <conference id="{$c/confId/text()}">
      {$c/confName}
      {for $p in table("paper")
       where $c/confId = $p/confId and $p/year > 2000
       return
       <paper>
          {$p/pid}
          {$p/title}
          {$p/year}
       </paper>
      }
   </conference>
  }
</conferencePapers>
```

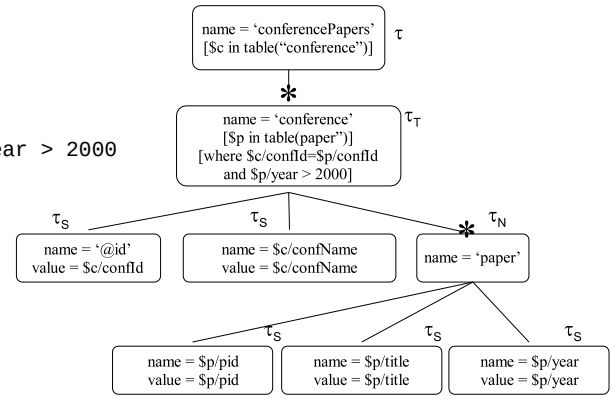

Figure 4: Example of UXQuery that joins two relations and its auxiliary query tree

for each variable specified in the `xnest` operator (XQuery does not accept variable names with (’). However, we use them here for ease of explanation).

The normalization process also makes sure that nested elements are related to the nesting variable. This is done by adding a new condition in the `where` clause. In the example (line 32) we added a condition requiring that the paper was published in the year specified by *$year*.

Note that this example shows a nesting over a single attribute, but that it is possible to specify nests over more than one attribute.

A formal specification of the normalization process can be found in Appendix B.

3. Mapping Well-behaved XML Views to Relational Views

In order to check the updatability of well-behaved XML views constructed by UXQuery, we map a given XML view to a set of corresponding relational views and use the techniques of updating through relational views to determine the XML view updatability. In particular, we use the Dayal and Bernstein technique [6, 12, 13].

We must therefore first map an XML view to its set of corresponding relational views. The main idea behind the mapping process is to unnest the XML view and produce the flat corresponding relational views. In order to do so, we use an auxiliary query tree that carries information about the structure of the XML view, the source of each XML element/attribute and the restrictions applied to build the view. Each non-leaf node of the auxiliary query tree has a name and possible annotation, and each leaf node in the tree has a name and a value.

To illustrate, we give a simple example that does not have the `xnest` operator. Figure 4 shows a query and its corresponding auxiliary query tree. Annotations are shown between brackets (“[ ]”). There are two types of annotations: "where" annotations and "variable binding" annotations. Each XML element specified in the query is represented by a node in the auxiliary query tree. When an XML element is generated by an expression containing a variable, we name the node with the corresponding expression (see node *$p/pid*). Attributes are represented in the auxiliary query tree in the same way as subelements, with the difference that their name starts with "@" (see node *@id*).

Auxiliary query trees are constructed from the view query as follows: For each XML element/attribute returned by the query, a node is created in the tree. For each node, we annotate all variable bindings and `where` conditions found between the node and the next non-leaf node in the query. As an example, node *conferencePapers* in the query tree of Figure 4 has an annotation for the binding of variable *$c*. Node *conference* has annotations about the binding of variable *$p*

and the condition *where $c/confId=$p/confId and $p/year > 2000*.

In the subset of XQuery we are using, leaf nodes can be constructed in two different ways: We can specify the entire element using a variable (e.g `{$c/confName}`), or we can explicitly specify an XML element, and the value of its content using a variable (e.g `<name>{$c/confName/text()}</name>`). Both constructors are mapped to leaf nodes in the auxiliary query tree.

Connections in the auxiliary query tree represents parent/child relationships. A repeating element is connected to its parent by an *-edge, while non-repeating elements are connected by a simple edge. Repeating elements are those returned after a `for` or a `xnest`. Additionally, the root node of an *element group* within a `xnest` also receive an *-edge.

With this auxiliary query tree, we are now able to map an XML view constructed with UXQuery to its set of corresponding relational views. The generic mapping process is as follows:

```
SELECT <leaf value> AS <leaf name>, ..., <leaf value> AS <leaf name>
FROM (<relation> AS <variable> LEFT JOIN <relation> AS <variable> ON <joincond>) LEFT JOIN ...
      <relation> AS <variable> ON <joincond>
WHERE (<"where" annotation> OR <"where" variable> IS NULL) AND ...
       AND (<"where" annotation> OR <"where" variable> IS NULL)
```

For query of Figure 4, the generated relational view is the following:

```
SELECT c.confId AS id, c.confName AS confName, p.pid AS pid, p.title AS title, p.year AS year
FROM conference AS c LEFT JOIN paper AS p ON c.confId=p.confId
WHERE (p.year > 2000 OR p.year IS NULL)
```

The name of each attribute in the relational view (specified after an `AS` expression) is generated by the evaluation of the expression specified in the name of each leaf node. As an example, the node *id* has *name='@id'*, so the name *@id* is copied to the SELECT expression without the "@". The node *confName* specifies *name=$c/confName*. This expression is evaluated as the name of the *confName* attribute pointed by variable *$c*, which is obviously *confName*. The same is done for the other attributes.

The `FROM` clause is constructed using the source table of each variable annotated in the auxiliary query tree. The variable name is used as an alias. For example, *$c* is a variable that is bound to the *conference* table, so *c* is its alias in the `FROM` clause. We use `LEFT JOIN` between ancestor-descendant nodes in the tree because it preserves empty sets in the nesting. For example, if a conference has no papers, the conference will still appear in the XML view.

The `WHERE` clause is generated using the annotations in the tree that were not used as join conditions. For each of these conditions, we add an "OR IS NULL" clause to ensure that empty sets are preserved in the nesting (e.g. otherwise conferences that have no papers would not appear in the view, because they do not satisfy the `WHERE` condition).

The auxiliary tree of queries involving `xnest` are constructed in a slightly different manner. Annotations of variables and where clauses within a `xnest` expression are placed on the node that represents the *header* element of the `xnest` expression. For example, the query in Figure 3 is shown again in Figure 5 together with its auxiliary query tree. Proceeding with the mapping process, the query in Figure 5 corresponds to the following relational view.

```
SELECT c.confId AS id, c.confName AS confName, p.year AS year, p.pid AS pid, p.title AS title
FROM conference AS c LEFT JOIN paper AS p ON c.confId=p.confId
```

There are cases where an XML view is mapped to more than one relational view, as in the query of Figure 6 (the resulting XML instance is shown in Figure 8). This XML view is mapped to two relational views: one containing data about authors and papers, and the other one containing data about authors and books. The decision of where to "split" the view is based on `for`s that appear on the same nesting level in the normalized query (the normalized query for the query in Figure 6 is shown in Figure 7). Each of these `for`s creates a new set of tuples, which

```
<conferencePapers>
  {for $c in table("conference")
   return
   <conference id="{$c/confId/text()}">
      {$c/confName}
      {xnest $p in table("paper")
       by $year in ($p/year)
       where $c/confId=$p/confId
       return
       <papers year="{$year/text()}">
           {<paper>
              {$p/pid}
              {$p/title}
           </paper>}
       </papers>
      }
   </conference>
  }
</conferencePapers>
```

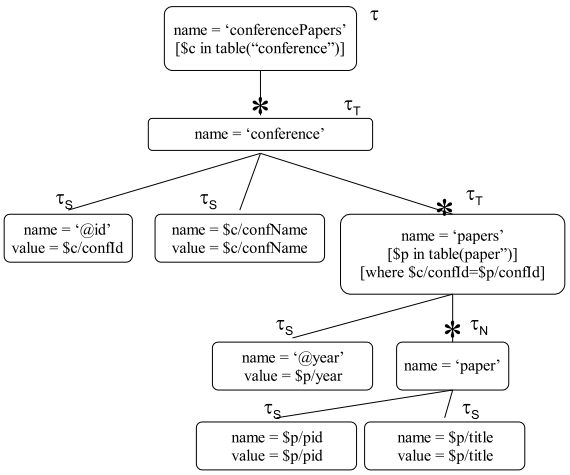

Figure 5: Example of UXQuery that nests elements, and its corresponding auxiliary query tree

should be mapped to distinct relational views. Information on levels above is considered to be common to both set of tuples. The resulting relational views are shown below (we name these views in order to be able to reference them in next section):

```
CREATE VIEW VIEWBOOK AS
SELECT a.id AS id, a.name AS name, a.email AS email, b.year AS year, b.title AS title,
       b.isbn AS isbn
FROM (author AS a LEFT JOIN ba AS ba ON ba.author=a.id) LEFT JOIN book AS b ON b.isbn=ba.isbn

CREATE VIEW VIEWPAPER AS
SELECT a.id AS id, a.name AS name, a.email AS email, p.year AS year, p.title AS title,
       p.pid AS pid
FROM (author AS a LEFT JOIN pa AS pa ON pa.author=a.id) LEFT JOIN paper AS p ON p.pid=pa.pid
```

The algorithms described in this section are available in [14].

4. Checking for XML View Updatability

In this section, we use the mapping from an XML view to its corresponding relational views as explained in section 3 to exemplify update operations over XML views. We further discuss the intuition of determining the updatability of XML views constructed by UXQuery. A complete study of updatability of XML views produced by UXQuery is beyond the scope of this paper.

Before presenting our update strategy, it is necessary to define precisely what we mean by side-effects, or problematic updates.

DEFINITION 2 *Let D be a relational database and V a view query definition over D. Let U be an update over the view produced by V and t its translation to the relational database. We say that U is a side-effect free update when $U(V(D)) = V(t(D))$.*

Our syntax for updates is similar to that of [4]. An update operation is a triple <u,Δ,*ref*>, where u is the type of operation (insert, delete, modify); Δ is the XML tree to be inserted, or (in case of a modification) an atomic value; and *ref* is a simple path expression in XPath [15] which indicates where the update is to occur. Note that the path expression may evaluate to a set of nodes in the tree. Deletions do not need to specify a Δ, since all the nodes under the evaluation of *ref* will be deleted.

In the examples of this section, we use the XML view resulting from the query in Figure 6 as shown in Figure 8. The update operations are also specified in Figure 8.

```
<authors>
  {for $a in table("author")
  return
  <author id="{$a/id/text()}">
     {$a/name}
     <address>
        {$a/email}
     </address>
     {xnest $ba in table("ba"), $b in table("b"),
           $pa in table("pa"), $p in table("paper")
      by $year in ($b/year | $p/year)
      where $ba/author=$a/id and $b/isbn=$ba/isbn and
            $pa/author=$a/id and $p/pid=$pa/pid
      return
      <publications year="{$year/text()}">
         {<book>
            {$b/title}
            {$b/isbn}
         </book>}
         {<conf>
            {$p/title}
            {$p/pid}
         </conf>}
      </publications>
      }
  </author>
  }
</authors>
```

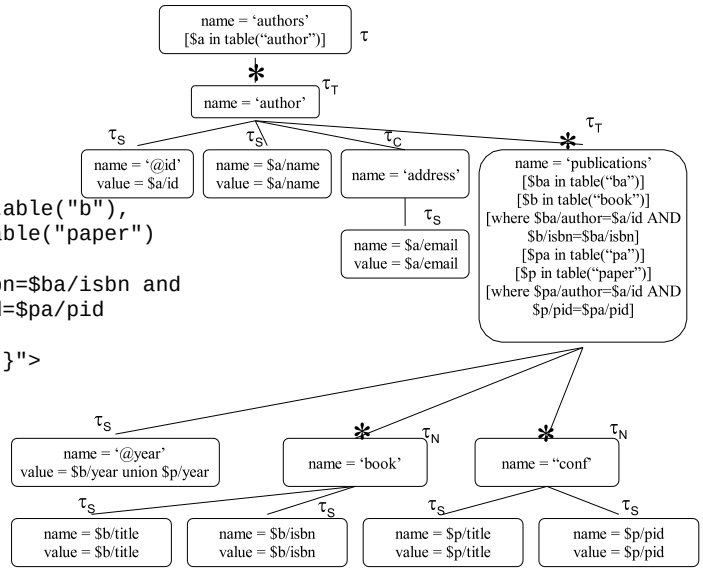

Figure 6: Example of UXQuery that mixes information of different relations in the same nesting level and the corresponding auxiliary query tree

An attempt to insert a new author in this view would be specified as U_1. This would be translated to the following insertions over the relational views:

```
INSERT INTO VIEWBOOK (id, name, email)
VALUES (4, "Robert White", "white@zzz.com")

INSERT INTO VIEWPAPER (id, name, email)
VALUES (4, "Robert White", "white@zzz.com")
```

The translation mechanism also uses the auxiliary query tree of the view definition query. First, the path expression in *ref* (without the filters specified between brackets (if any)) is evaluated against the auxiliary query tree. Then the structure of the view being inserted is "superimposed" on the auxiliary query tree. After this, we check to see what portions of the tree are referenced by the update operation and decide which relational views the operation should be translated to. In this first example, the subtree being inserted is on the "common" part of the tree, so we translate it to both relational views (we discuss alternatives to this method in Section 6). Once we decide which views to map the insertion to, we generate an INSERT SQL statement containing the information specified in the subtree being inserted, and also with information collected from the leaves under the elements along the path from *ref* to the root of the XML tree (this will be clearer in the next example).

An example where we use additional information to generate the INSERT SQL statement is specified in U_2. In this example, since *ref* points to an interior node, the information collected from the leaves under the elements along the path from *ref* to the root of the XML tree are also used in the INSERT statement. In this example, we use the author's *name*, *email* and *id*. The translation is as follows:

```
INSERT INTO VIEWBOOK (id, name, email, year, title, isbn)
VALUES (1, "Mary Jones", "maryjones@aaa.com", 2000, "Book6", 9888)
```

It is also possible to insert an author with publications (U_3). As the structure of the subtree being inserted matches elements in the auxiliary query tree that are split into separate relational views, we split the subtree being inserted in the same way. The resulting translation is:

```
<authors>
  {for $a in table("author")
  return
  <author id="{$a/id/text()}">
     {$a/name}
     <address>
        {$a/email}
     </address>
     {let $ba1 := table("ba"), $b1 := table("book"), $pa1 := table("pa"), $p1 := table("paper")
     for $year in distinct-values($p1/year | $b1/year)
     return
     <publications year="{$year/text()}">
        {for $ba in table("ba"), $b in table("book")
         where $ba/author=$a/id and $b/isbn=$ba/isbn and $b/year=$year
         return
         <book>
           {$b/title}
           {$b/isbn}
         </book>  }
        {for $pa in table("pa"), $p in table("paper")
         where $pa/author=$a/id and $p/pid=$pa/pid and $p/year=$year
         return
         <conf>
           {$p/title}
           {$p/pid}
         </conf>  }
     </publications>
     }
  </author>
  }
</authors>
```

Figure 7: Normalized query corresponding to query in Figure 6

```
INSERT INTO VIEWBOOK (id, name, email, year, title, isbn)
VALUES (5, "James Perez", "james@zzz.com", 2000, "View Updates", 999)

INSERT INTO VIEWPAPER (id, name, email, year, title, pid)
VALUES (5, "James Perez", "james@zzz.com", 2000, "Views and XML", "VIEW")
```

As an example of a modification update, consider U_4 which modifies the title of a given book. The attribute to be modified is the last attribute specified in the path expression in *ref*. The conditions for the modification are the filters used in the path expression. This would be translated as:

```
UPDATE VIEWBOOK SET title="Querying the Web using XML"
WHERE isbn=1234
```

As mentioned in the introduction, a problematic update operation would occur if we had specified an author in the path expression of U_4. Consider the previous example, with the following path expression: `/authors/author[@id="1"]/publications/book[isbn="1234"]/title`. The translation for this operation would include information about the author too, but it would not be possible to translate this to the underlying relational database without causing side effects. More specifically, the book under author with *@id="2"* would also be changed.

An example of deletion would be specified as U_5. As with modifications, we use the filters specified in the path expression to generate the WHERE clause of the delete statement. This would be translated to the relational view as:

```
DELETE FROM VIEWBOOK
WHERE id=1 AND year=2000

DELETE FROM VIEWPAPER
WHERE id=1 AND year=2000
```

As we can easily see, view updatability depends on the update operation being applied. However, it depends also on the structure of the view. We were able to translate most of the update

Figure 8: XML view resulting from query in Figure 6 and examples of update operations

operations specified over the view above because the view has the following properties: it keeps the primary keys of all the tables involved, and joins were made over foreign keys. For a view that does not obey these restrictions, we would not be able to translate most of the sample update operations. For details, please see [4].

We use the technique of Dayal and Bernstein [6, 12, 13] to translate updates on the relational view to updates on the underlying relational database. Their work presents algorithms to update the underlying relational database when a unique, side effect-free translation exists, and detects when such an update does not exist (for a summary of the algorithms we use, please refer to [16]). Using these algorithms, it is possible to determine whether the XML view is or is not updatable with respect to a given update.

5. Related Work

There has been a lot of work addressing the problem of building XML views from relational databases [1, 2, 3, 17, 18]. Most of them approach the problem by building a *default* XML view from the relational source and then using an XML query language to query the default view [1, 2, 3, 17]. Most of these approaches use extended SQL to build the default view. The exception is XPERANTO [2], whose default view is an XML document containing all the database tables represented in XML. This view can then be queried using XQuery augmented with a new input function called `view`. This function accesses the default XML view in the same way that our input function `table` is used to access relational tables. However, we do not have the concept of a default view. Our function `table` accesses the relational tables directly.

Another difference between XPERANTO and our approach is that they generate a single SQL query for each query over the view. Their translation involves transforming an XQuery

into a representation called XQGM, which is very similar to the internal representation of SQL queries in DB2 (QGM). However, the purpose of transforming XQuery into SQL is different in our approach. XPERANTO does this transformation with the goal of using the relational engine to execute the query. We perform the transformation because we want to use the relational view to check for XML view updatability.

None of the above proposals addresses the problem of updating the resulting XML view and mapping the updates to the underlying relational database. [19, 20] presents a proposal for updating XML views over relational databases. The views are constructed by a subset of XQuery. However, the subset is not clearly identified. The concern in this proposal is also to avoid side-effects in updates. Their translation algorithms are based on the Object-View approach [21].

Commercial databases also provide ways of exporting relational data as XML. IBM DB2 XML Extender [22] uses a mapping file called DAD (*Data Access Definition*) to specify how a given SQL query is mapped to XML. This mapping file is very complex, and is generally built using a wizard. Oracle 9i release 2 uses SQL/XML [9]. SQL Server extends SQL with a directive called `FOR XML` [23]. As we can see, most commercial databases have their own way of dealing with XML, which makes it difficult to use them for accessing legacy databases. As for updates, DB2, which allows the creation of XML documents from relational tables, requires that updates be issued directly to the relational tables. In SQL Server an XML view generated by an annotated XML Schema can be modified using *updategrams*. Instead of using INSERT, UPDATE or DELETE statements, the user provides a before image of what the XML view looks like and an after image of the view [24]. The system computes the difference between these images and generates corresponding SQL statements to reflect changes on the relational database. Oracle offers the option of specifying an annotated XML Schema, but the only possible update operation is to insert XML documents that agree with an annotated XML Schema.

There has been a significant amount of work in querying XML documents stored in relational databases [1, 25, 26]. Proposals for updating XML documents stored in relational databases include [27, 26]. These approaches are different from ours because they consider a different question: they query XML documents stored in relational databases, while we query relational databases to extract XML views. Therefore, the underlying assumptions used are different. For example, querying XML documents stored in relational databases must preserve document order, while in our case, order is not important, since the relational model is unordered. On the other hand, the flat nature of relational databases may cause redundancy when translated to XML views, which may cause problems regarding updates as illustrated in the introduction. That is, a well designed relational database does not imply a redundancy-free XML view. This problem is not critical for XML documents stored in relational databases since well designed XML documents [28] tend not to be redundant. Additionally, existing proposals for updating XML documents stored in relational databases do not consider updates through views.

6. Discussion and Future Work

In this paper we propose a subset of XQuery, UXQuery, to build updatable XML views over relational databases. The main contribution of the paper is a mapping from an XML view constructed using UXQuery to a set of corresponding relational views, which are then used to translate updates over the XML view to updates over the corresponding relational views. The approach therefore reduces the problem of updating XML views to a well studied problem, that of updating relational views.

There are a few open problems in our approach. The main problem is related to the allowed

update operations. Currently, we are allowing only updates that can be unambiguously mapped to the relational database without causing side effects. Problematic updates are not allowed. A possible solution to this limitation is to obtain user input for problematic updates. This solution would be based on dialogs with the user, in a way similar to Keller's proposal [7, 21]. These dialogs could occur at view definition time, or when a problematic update is issued. As an example, if a user attempted to update a book title by specifying a path that also includes an author (as in `/authors/author[@id="1"]/publications/book[isbn="1234"]/title`), we would ask the user if he wants to modify all the titles of the book with *isbn="1234"*. If so, the operation would be performed, otherwise, the operation would be cancelled.

UXQuery is obviously less expressive than XQuery. In particular, it is not capable of expressing aggregations and arbitrary restructuring in the XML view. Although this is a trade-off imposed by our goal of updating the relational database through XML views, it may be possible to recognize updatable portions of views expressed in a more general language. For example, if the view presented author information and the total number of papers they had written, it is still possible to update author information even if updating the total number of papers is not allowed.

References

[1] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. Silkroute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS)*, 27(4):438–493, Dec 2002.

[2] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene Shekita, Catalina Fan, and John Funderburk. Querying XML views of relational data. In *Proc. of International Conference on Very Large Databases*, Roma, Italy, September 2001.

[3] P. Bohannon, S. Ganguly, H.F. Korth, P.P.S. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *Proc. of International Conference on Very Large Databases*, Hong Kong, China, August 2002.

[4] Vanessa Braganholo, Susan Davidson, and Carlos Heuser. On the updatability of XML views over relational databases. In *Proc. of WEBDB 2003*, pages 31–36, San Diego, California, June 2003.

[5] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. of Symposium on Principles of Database Systems*, pages 124–138, Los Angeles, CA, March 1982.

[6] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 8(2):381–416, Sep 1982.

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

[8] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Working Draft, May 2003. `http://www.w3.org/TR/2003/WD-xquery-20030502/`.

[9] A. Eisenberg and J. Melton. SQL/XML is making good progress. *SIGMOD RECORD*, 31(2), 2002.

[10] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, May 2003. `http://www.w3.org/TR/2003/WD-xquery-semantics-20030502/`.

[11] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 functions and operators. W3C Working Draft, May 2003. `http://www.w3.org/TR/2003/WD-xpath-functions-20030502/`.

[12] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *Proc. of International Conference of Very Large Databases*, pages 368–377, West Berlin, Germany, Sep 1978.

[13] Umeshwar Dayal and Philip A. Bernstein. On the updatability of network views - extending relational view theory to the network model. *Information Systems*, 7(2):29–46, 1982.

[14] Vanessa Braganholo, Susan Davidson, and Carlos Heuser. Using XQuery to build updatable XML views over relational databases. Technical Report MS-CIS-03-18, Department of Computer and Information Science, University of Pennsylvania, 2003.

[15] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. W3C Recomendation, Nov 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[16] Vanessa Braganholo, Susan Davidson, and Carlos Heuser. Reasoning about the updatability of XML views over relational databases. Technical Report MS-CIS-03-13, Department of Computer and Information Science, University of Pennsylvania, 2003.

[17] Surajit Chaudhuri, Raghav Kaushik, and Jeffrey Naughton. On relational support for XML publishing: Beyond sorting and tagging. In *Proc. of International Conference on Management of Data*, San Diego, California, Jun 2003.

[18] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. *The VLDB Journal*, pages 65–76, 2000.

[19] Ling Wang, Mukesh Mulchandani, Xin Zhang, and Elke A. Rundensteiner. Updating XQuery Views Published over Relational Data. Technical Report WPI-CS-TR-TBA, Worcester Polytechnic Institute, 2003.

[20] Ling Wang, Mukesh Mulchandani, and Elke A. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *Proc. of XML Database Symposium*, Berlin, Germany, Sep 2003.

[21] Thierry Barsalou, Niki Siambela, Arthur M. Keller, and Gio Wiederhold. Updating relational databases through object-based views. In *Proc. of International Conference on Management of Data*, pages 248–257, Denver, Colorado, 1991.

[22] J. Cheng and J. Xu. XML and DB2. In *Proc. of International Conference on Data Engineering*, San Diego, California, 2000.

[23] Andrew Conrad. A Survey of Microsoft SQL Server 2000 XML Features. MSDN Library. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml07162001.asp, July 2001.

[24] Andrew Conrad. Interactive Microsoft SQL Server & XML Online Tutorial. Available at http://www.topxml.com/tutorials/main.asp?id=sqlxml.

[25] David DeHaan, David Toman, Mariano Consens, and M. Tamer Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proc. of International Conference on Management of Data*, San Diego, California, Jun 2003.

[26] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, and E. Shekita. Storing and querying ordered XML using a relational database system. In *Proc. of International Conference on Management of Data*, Madison, Wisconsin, Jun 2002.

[27] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. of International Conference on Management of Data*, Santa Barbara, California, May 2001.

[28] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. In *Proc. of ACM Symposium on Principles of Database Systems*, Madison, Wisconsin, Jun 2002.

[29] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (xml) 1.0 (second edition). W3C Recommendation, Oct 2002. http://www.w3.org/TR/2000/REC-xml-20001006.

[30] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. W3C Recommendation, Jan 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114.

A. UXQuery EBNF

In the definitions of this section we use a set of grammar definitions available in the XML documentation. The basic tokens `Letter` and `Digit` are defined in [29]. The identifier `QName` is defined in [30]. Literals and numbers are defined in [8].

```
[1]  UXQuery         ::= QueryBody
[2]  QueryBody       ::= ElmtConstructor
[3]  ElmtConstructor ::= "<" QName AttList "/>" | "<" QName AttList? ">" ElmtContent+ "</" QName ">"
[4]  ElmtContent     ::= ElmtConstructor | EnclosedExpr+
[5]  AttList         ::= ((QName "=" AttValue)?)+
[6]  AttValue        ::= ('"' AttValueContent '"') | ("'" AttValueContent "'")
[7]  AttValueContent ::= "{" PathExprAtt "}"
[8]  PathExprAtt     ::= "$" VarName "/" QName "/" NodeTest
```

```
[9]  VarName         ::= QName
[10] EnclosedExpr    ::= "{" (FWRExpr | PathExpr | Nest) "}"
[11] Expr            ::= OrExpr
[12] OrExpr          ::= AndExpr ( "or"  AndExpr )*
[13] AndExpr         ::= ComparisonExpr ("and" ComparisonExpr)*
[14] FWRExpr         ::= ((ForClause)+ WhereClause? OrderByClause? "return")* ElmtConstructor
[15] ComparisonExpr  ::= ValueExpr (GeneralComp ValueExpr )?
[16] ValueExpr       ::= PathExpr | PrimaryExpr
[17] PathExpr        ::= "$" VarName "/" QName ("/" NodeTest)?
[18] NodeTest        ::= TextTest
[19] TextTest        ::= "text" "(" ")"
[20] ForClause       ::= "for" "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[21] TableExpr       ::= "table (" '"' QName '"' ")" | "table (" "'" QName "'" ")"
[22] WhereClause     ::= "where" Expr
[23] GeneralComp     ::= "=" |  "!=" |  "<" |  "<=" |  ">" |  ">="
[24] OrderByClause   ::= "order" "by" OrderSpecList
[25] OrderSpecList   ::= OrderSpec ("," OrderSpec)*
[26] OrderSpec       ::= PathExpr
[27] PrimaryExpr     ::= Literal | ParenthesizedExpr
[28] Literal         ::= NumericLiteral |  StringLiteral
[29] NumericLiteral  ::= IntegerLiteral |  DecimalLiteral |  DoubleLiteral
[30] ParenthesizedExpr  ::= "(" Expr? ")"
[31] Nest       ::= NestClause ByClause WhereClause "return" Header
[32] NestClause ::= "xnest "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[33] ByClause   ::= "by" "$" VarName "in" UnionExpr ("," "$" VarName "in" UnionExpr)*
[34] Header ::= "<" QName (QName "=" NestAttValue)+ ">" ( "{" ElGroup "}" )+ "</" QName ">"
       | "<" QName ">" (( "{" "$" VarName "}") | ("<" QName ">" "{" "$" VarName "/" TextTest "}"
         "</" QName ">"))+ ( "{" ElGroup "}" )+ "</" QName ">"
[35] NestAttValue ::= "'" "{" "$" VarName "/" TextTest "}" "'"
                    |  '"' "{" "$" VarName "/" TextTest "}" '"'
[36] ElGroup    ::= ElmtConstructor
[37] UnionExpr  ::= "(" "$" VarName "/" QName ( ("union" | "|")  "$" VarName "/" QName)* ")"
```

B. Normalization Process for the `xnest` Operator

The notation for the normalization process is the same as that in [10].

[xnest Variable$_1$ in TableExpr$_1$, . . . , Variable$_n$ in TableExpr$_n$
by NestVariable$_1$ in (Variable$_{1_1}$/QName$_{1_1}$ | . . . | Variable$_{1_m}$/QName$_{1_m}$),
. . . , NestVariable$_k$ in (Variable$_{k_1}$/QName$_{k_1}$ | . . . | Variable$_{k_m}$/QName$_{k_m}$)
where Expr return
<ElName AttName$_1$="{NestVariable$_1$/text()}" . . . AttName$_k$="{NestVariable$_k$/text()}">
{ElGroup$_1$} . . . {ElGroup$_m$} </ElName> **]**$_{XNest}$
==
let Variable$'_1$:= TableExpr$_1$, . . . , Variable$'_n$:= TableExpr$_n$
for NestVariable$_1$ in distinct-values(Variable$_{1_1}$/QName$_{1_1}$ | . . . | Variable$_{1_m}$/QName$_{1_m}$),
. . . , NestVariable$_k$ in distinct-values(Variable$_{k_1}$/QName$_{k_1}$ | . . . | Variable$_{k_m}$/QName$_{k_m}$)
return
<ElName AttName$_1$="{NestVariable$_1$/text()}", . . . , AttName$_k$="{NestVariable$_k$/text()}">
{for fs:SubVariable(1)
where fs:SubExpr(1) and (Variable$_{1_1}$ = NestVariable$_1$ and . . . and Variable$_{k_1}$ = NestVariable$_k$)
return ElGroup$_1$ }
. . .
{for fs:SubVariable(m)
where fs:SubExpr(m) and (Variable$_{1_m}$ = NestVariable$_1$ and . . . and Variable$_{k_m}$ = NestVariable$_k$)
return ElGroup$_m$ }
</ElName>

This normalization process assumes that:

- {Variable$_{1_1}$, . . . , Variable$_{1_m}$, . . . , Variable$_{k_1}$, . . . , Variable$_{k_m}$} $\subseteq$ {Variable$_1$, . . . , Variable$_n$}
- The auxiliary function *fs:SubVariable(i)* returns all variables V_x referenced in *ElGroup$_i$* and also all variables V_y appearing in a condition of the form "$V_x/QName_x$ **cmp** $V_y/QName_y$" in *Expr* in the `where` clause of the `xnest` operator; **cmp** $\in \{" = ", " < ", " > ", "! = ", " <= ", " >= "\}$.
- The auxiliary function *fs:SubExpr(i)* returns every expression specified in *Expr* in the `where` clause of the `xnest` operator that references a variable in *fs:SubVariable(i)*.