

# Using XQuery to build updatable XML views over relational databases

Vanessa P. Braganholo<sup>(1)</sup>, Susan B. Davidson<sup>(2)</sup>, Carlos A. Heuser<sup>(1)</sup>

<sup>(1)</sup> Universidade Federal do Rio Grande do Sul - UFRGS

Instituto de Informática

{vanessa, heuser}@inf.ufrgs.br

<sup>(2)</sup> 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 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 [21, 20, 25, 4]: 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 query 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 without introducing additional updates to the XML view?

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 specify this update 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

Author			Conference	
id	name	email	confid	confName
1	Mary Jones	maryjones@aaa.com	DEXA	Conference on Database and Expert Systems Applications
2	Charles Green	charles@bbb.com	PODS	Symposium on Principles of Database Systems
3	Michael Kurt	kurt@ccc.com	VLDB	Conference on Very Large Data Bases

Ba		Paper				
author	isbn	pid	title	confid	year	
1	1234	IR	Databases and IR	VLDB	2002	<b>CONSTRAINTS</b> On table Paper: CONSTRAINT ConfPaper foreign key (confid) references Conference On table Ba: CONSTRAINT AuthorBa foreign key (author) references Author CONSTRAINT BookBa foreign key (isbn) references Book On table Pa: CONSTRAINT AuthorPa foreign key (author) references Author CONSTRAINT PaperPa foreign key (pid) references Paper
1	1235	QWEB	Querying the Web	DEXA	2000	
1	1238	WEB	Web Survey	VLDB	2001	
2	1234					
2	1237					
2	1238					
3	1235					
3	1236					

Pa		Book			
author	pid	isbn	title	year	
1	IR	1234	Book1	2000	
1	QWEB	1235	Book2	2001	
1	WEB	1236	Book3	2000	
2	IR	1237	Book4	2001	
2	WEB	1238	Book5	2001	
3	WEB				

Figure 1: Sample database

expression, all titles of papers with id "IR" would be altered in the view, and no side effects would occur.

In previous work [5], we addressed this problem by considering the nested relational algebra (NRA) [22] 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 [15, 23], and map a new problem (updating relational databases through NRA views) to a well studied problem.

---

<pre> &lt;Result&gt; &lt;Author ID="1"&gt;   &lt;Name&gt; Mary Jones &lt;/Name&gt;   &lt;Email&gt; maryjones@aaa.com &lt;/Email&gt;   &lt;Papers year="2002"&gt;     &lt;Paper Id="IR"&gt;       &lt;Title&gt; Databases and IR &lt;/Title&gt;       &lt;ConfID&gt; VLDB &lt;/ConfID&gt; &lt;/Paper&gt; &lt;/Papers&gt;     &lt;Papers year="2000"&gt;       &lt;Paper Id="QWEB"&gt;         &lt;Title&gt; Querying the Web &lt;/Title&gt;         &lt;ConfID&gt; DEXA &lt;/ConfID&gt; &lt;/Paper&gt; &lt;/Papers&gt;     &lt;Papers year="2001"&gt;       &lt;Paper Id="WEB"&gt;         &lt;Title&gt; Web Survey &lt;/Title&gt;         &lt;ConfID&gt; VLDB &lt;/ConfID&gt; &lt;/Paper&gt; &lt;/Papers&gt;   &lt;/Author&gt; &lt;Author ID="2"&gt;   &lt;Name&gt; Charles Green &lt;/Name&gt;   &lt;Email&gt; charels@bbn.com &lt;/Email&gt;   &lt;Papers year="2002"&gt;     &lt;Paper Id="IR"&gt;       &lt;Title&gt; Databases and IR &lt;/Title&gt;       &lt;ConfID&gt; VLDB &lt;/ConfID&gt; &lt;/Paper&gt; &lt;/Papers&gt;   ... &lt;/Author&gt; ... &lt;/Result&gt;           </pre>	<pre> (a)           </pre>	<pre> &lt;Result&gt; &lt;Author ID="1"&gt;   &lt;Name&gt; Mary Jones &lt;/Name&gt;   &lt;Email&gt; maryjones@aaa.com &lt;/Email&gt;   &lt;Papers year="2002"&gt;     &lt;Paper Id="IR"&gt;       &lt;Title&gt; Databases and IR &lt;/Title&gt;       &lt;ConfID&gt; VLDB &lt;/ConfID&gt; &lt;/Paper&gt; &lt;/Papers&gt;     &lt;Papers year="2000"&gt;       &lt;Paper Id="QWEB"&gt;         &lt;Title&gt; Querying the Web &lt;/Title&gt;         &lt;ConfID&gt; DEXA &lt;/ConfID&gt; &lt;/Paper&gt;       &lt;Book ISBN="1234"&gt;         &lt;Title&gt; Book1 &lt;/Title&gt;       &lt;/Book&gt;     &lt;/Papers&gt;     &lt;Papers year="2001"&gt;       &lt;Paper Id="WEB"&gt;         &lt;Title&gt; Web Survey &lt;/Title&gt;         &lt;ConfID&gt; VLDB &lt;/ConfID&gt; &lt;/Paper&gt;       &lt;Book ISBN="1235"&gt; &lt;Title&gt; Book2 &lt;/Title&gt;       &lt;/Book&gt;       &lt;Book ISBN="1238"&gt; &lt;Title&gt; Book5 &lt;/Title&gt;       &lt;/Book&gt;     &lt;/Papers&gt;   &lt;/Author&gt;   ... &lt;/Result&gt;           </pre>	<pre> (b)           </pre>
---	----------------------------	---	----------------------------

---

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

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 [3] 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 [19], an extension to SQL being developed by INCITS ([http://www.ncits.org/tc\\_home/h2.htm](http://www.ncits.org/tc_home/h2.htm)): 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 `nest` to facilitate nesting. Note that `nest` does not add anything to the language, and that queries containing `nest` 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.

## 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 [5], 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) whose structure conforms to  $\tau$ , where:*

$$\begin{aligned} \tau &= \text{root} : \{E_1 : \tau_T\}, \dots, \{E_n : \tau_T\} \\ \tau_T &= [E_1 : \tau_S, \dots, E_k : \tau_S, \{E_{k+1} : \tau_{k+1}\}, \dots, \{E_m : \tau_m\}], \text{ where } (k \geq 1), (m \geq 0) \text{ and } (\tau_i \text{ is } \\ &\tau_S \text{ or } \tau_T) \end{aligned}$$

The symbol  $\tau_T$  denotes a tuple type and  $\tau_S$  denotes an atomic type (e.g. #PCDATA or CDATA). In this definition, *root* is an element name, and  $E_i$  is an element or attribute name. We adopt the convention that attribute names start with “@”.

As an example, the view in figure 2(b) would be represented as:

```
Result : {Author: [@ID: CDATA,
                Name: #PCDATA,
                Email: #PCDATA,
                {Papers: [@year: CDATA,
                        {Paper: [@Id : CDATA, Title: #PCDATA, ConfId: #PCDATA]},
                        {Book:  [@ISBN : #PCDATA, Title: #PCDATA]},
                        ]}
                ]
}
```

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 “nest”.

The subset we have chosen is called UXQuery, 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 *nest*, 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 [18] with the exception of the new input function *table* and the macro *nest*, which we discuss next.

## 2.1 Semantics of *table()*

XQuery has three input functions: *input()*, *collection()* and *document()* [24]. 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>
...
```

---

```

1. <conferencePapers>
2.   {for $c in table("conference")
3.     order by $c/confId
4.     return
5.     <conference id="{ $c/confId/text()}">
6.       { $c/confName
7.         {nest $p in table("paper")
8.           by $year in ($p/year)
9.           where $p/confId=$c/confId
10.          return
11.          <papers year="{ $year/text()}">
12.            {
13.              <paper>
14.                { $p/pid
15.                  { $p/title
16.                }
17.              }
18.            }
19.          }
20.        }
21.      }
22.    }
23. <conferencePapers>
24.   {for $c in table("conference")
25.     order by $c/confId
26.     return
27.     <conference id="{ $c/confId/text()}">
28.       { $c/confName
29.         {let $p' := table("paper")
30.           for $year in distinct-values($p'/year)
31.           return
32.           <papers year="{ $year/text()}">
33.             {for $p in table("paper")
34.               where $c/confId=$p/confId and $p/year=$year
35.               return
36.               <paper>
37.                 { $p/pid
38.                   { $p/title
39.                 }
40.               }
41.             }
42.           }
43.         }
44.       }
45.     }

```

---

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

### 2.1.1 Semantics of Nest

The `nest` operator is used to specify possibly heterogeneous sets of nested tuples. A simple (non-heterogeneous) example of such a query is shown in figure 3 (lines 1-22). 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 nested by year.

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

```

[34] Nest      ::= NESTClause BYClause WHEREClause "return" Header
[35] NESTClause ::= "nest" "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[36] BYClause  ::= "by" "$" VarName "in" UnionExpr ("," "$" VarName "in" UnionExpr)*
[37] Header   ::= "<" QName (QName "=" "{" "$" VarName "/" TextTest "}")+ ">" ( "{" ElGroup "}" )+ "</" QName S? ">"
                | "<" QName ">" ( "{" "$" VarName "}" | "<" QName ">" "{" "$" VarName "/" TextTest "}" "</" QName ">" )+
                ( "{" ElGroup "}" )+ "</" QName S? ">"
[38] ElGroup  ::= ElmtConstructor
[39] UnionExpr ::= "$" VarName "/" QName ( ("union" | "|") "$" VarName "/" QName)*

```

A query containing a `nest` operator can be normalized to one using pure XQuery syntax. The normalized query corresponding to the query in figure 3 (lines 1-22) is shown in figure 3 (lines 23-45). 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`.

Continuing with the example, the `nest` operation (lines 7-19) is normalized to the expression shown in lines 29-42. The expression consists of a `let/for` (lines 29-30) and an additional `for` (lines 33-40) for each *ElGroup* (lines 12-17) 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 `nest` operator. There will be one primed variable in the `let` clause for each variable specified in the `nest` operator (XQuery does not accept variable names with (`'`). However, we use them here for easy 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 34)

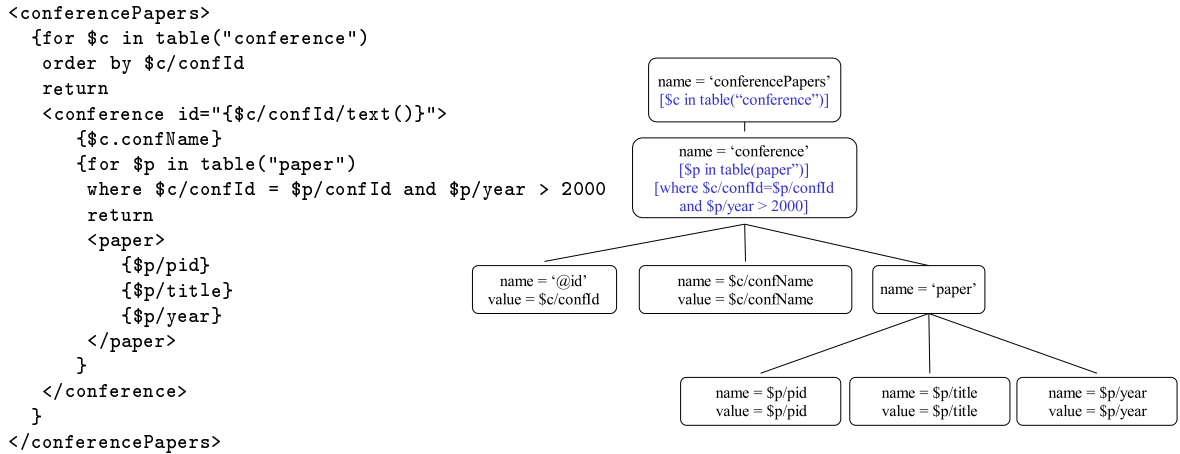


Figure 4: Example of UXQuery that joins two relations and its auxiliary tree

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 (i.e. specify heterogeneous sets in the nesting).

A formal specification of the normalization process can be found in [7].

### 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 [14, 15, 16].

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 XML view, the source of each XML element/attribute and the restrictions applied to build the view. Each non-leaf node of the auxiliary 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 `nest` operator. Figure 4 shows a query and its corresponding auxiliary tree. Annotations are shown between brackets (“[ ]”). Each XML element specified in the query is represented by a node in the auxiliary 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 tree in the same way as subelements, with the difference that their name starts with “@” (see node `@id`).

Auxiliary trees are constructed from the view query as follows: For each XML element specified in 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 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 explicitly specify an XML element, and the value of its content using a variable (e.g. `<name>{ $\$c/confName/text()$ }`), or we can specify the entire element using a variable (e.g. `{ $\$c/confName$ }`). Both constructors are mapped to leaf nodes in the auxiliary tree.

Connections in the auxiliary tree represents parent/child relationships.

For queries involving nests, the auxiliary tree is built based on the normalized query.

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 <cond>) LEFT JOIN ...
    <relation> AS <variable> ON <cond>
WHERE <where annotation> OR <annotation> IS NULL ... <where annotation> OR <annotation> 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 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 `nest` are based on the corresponding normalized query. 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 7). 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

---

```

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

```

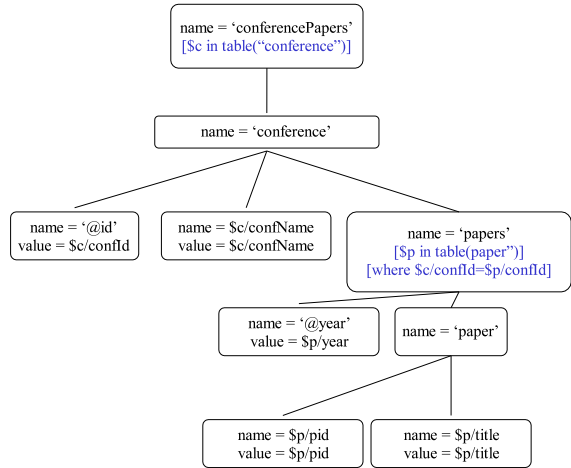


Figure 5: Example of UXQuery that nests elements, and its corresponding auxiliary tree

based on fors that appear on the same nesting level in the normalized query. Each of these fors creates a new set of tuples, which should be mapped to distinct relational views. In the example of figure 6, there are two fors on the same nesting level (lines 10-20 and 21-31). 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, 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, 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

```

## 4 Checking for XML view updatability

In this section we present examples of update operations over XML views and 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 out of the scope of this paper.

Our syntax for updates is similar to that of [5]. Basically, an update operation is a triple  $\langle u, \Delta, ref \rangle$ , where  $u$  is the type of operation (insert, delete, modify);  $\Delta$  is the XML tree to be inserted, or (in case of a modification) an atomic value; and  $ref$  is a path expression in XPath [12] 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  $\Delta$ , 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 7. The update operations are also specified in figure 7.

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:

```

<authors>
  {for $a in table("author")
  return
  <author id="{ $a/id/text()}">
    { $a/name }
    {nest $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>

```

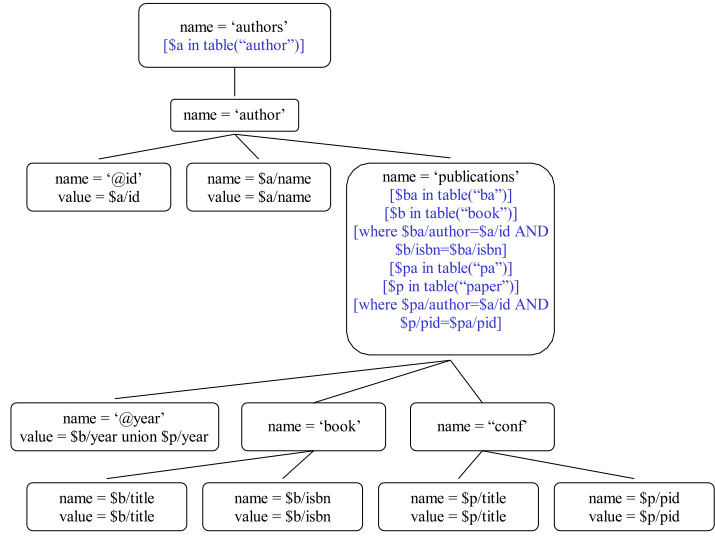


Figure 6: Example of UXQuery that mixes information of different relations in the same nesting level and the corresponding auxiliary tree

```

INSERT INTO VIEWBOOK (id, name)
VALUES (4, "Robert White")

INSERT INTO VIEWPAPER (id, name)
VALUES (4, "Robert White")

```

The translation mechanism also uses the auxiliary 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 tree. Then the structure of the view being inserted is “superimposed” on the auxiliary 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).

We use the technique of Dayal and Bernstein [14, 15, 16] to translate updates on the relational view to updates on the underlying relational database. Their work presents an algorithm 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 their algorithms, please refer to [6]).

When translating the second insertion to the underlying database, the Dayal and Bernstein technique will detect that the author is already in the database, and will not perform the insert

<pre> &lt;authors&gt;   &lt;author id="1"&gt;     &lt;name&gt;Mary Jones&lt;/name&gt;     &lt;publications year="2000"&gt;       &lt;book&gt;&lt;title&gt;Book1&lt;/title&gt;&lt;isbn&gt;1234&lt;/isbn&gt;&lt;/book&gt;       &lt;conf&gt;         &lt;title&gt;Querying the Web&lt;/title&gt;&lt;pid&gt;QWEB&lt;/pid&gt;       &lt;/conf&gt;     &lt;/publications&gt;     &lt;publications year="2001"&gt;       &lt;book&gt;&lt;title&gt;Book2&lt;/title&gt;&lt;isbn&gt;1235&lt;/isbn&gt;&lt;/book&gt;       &lt;book&gt;&lt;title&gt;Book5&lt;/title&gt;&lt;isbn&gt;1238&lt;/isbn&gt;&lt;/book&gt;       &lt;conf&gt;&lt;title&gt;Web Survey&lt;/title&gt;&lt;pid&gt;WEB&lt;/pid&gt;&lt;/conf&gt;     &lt;/publications&gt;     &lt;publications year="2002"&gt;       &lt;conf&gt;         &lt;title&gt;Databases and IR&lt;/title&gt;&lt;pid&gt;IR&lt;/pid&gt;       &lt;/conf&gt;     &lt;/publications&gt;   &lt;/author&gt;   &lt;author id="2"&gt;     &lt;name&gt;Charles Green&lt;/name&gt;     &lt;publications year="2000"&gt;       &lt;book&gt;&lt;title&gt;Book1&lt;/title&gt;&lt;isbn&gt;1234&lt;/isbn&gt;&lt;/book&gt;     &lt;/publications&gt;   ... &lt;/author&gt; ... &lt;/authors&gt; </pre>	<pre> U<sub>1</sub>: u = insertion, ref = /authors, Δ = {&lt;author id="4"&gt;   &lt;name&gt;Robert White&lt;/name&gt; &lt;/author&gt;}. </pre> <hr/> <pre> U<sub>2</sub>: u = insertion, ref = //author[@id="1"]/publications[@year="2000"], Δ = {&lt;book&gt;   &lt;title&gt;Book6&lt;/title&gt;&lt;isbn&gt;9888&lt;/isbn&gt; &lt;/book&gt;}. </pre> <hr/> <pre> U<sub>3</sub>: u = insertion, ref = /authors, Δ = {&lt;author id="5"&gt;   &lt;name&gt;James Perez&lt;/name&gt;   &lt;publication year="2000"&gt;     &lt;book&gt;       &lt;title&gt;Updating Relational Views&lt;/title&gt;       &lt;isbn&gt;999&lt;/isbn&gt;&lt;/book&gt;     &lt;conf&gt;       &lt;title&gt;Views and XML&lt;/title&gt;       &lt;pid&gt;VIEW&lt;/pid&gt;&lt;/conf&gt;     &lt;/publication&gt;   &lt;/author&gt;}. </pre> <hr/> <pre> U<sub>4</sub>: u = modification, Δ = {Querying the Web using XML}, ref = //book[isbn="1234"]/title. </pre> <hr/> <pre> U<sub>5</sub>: u = deletion, ref = //author[@id="1"]/publications[@year="2000"]. </pre>
---	--

Figure 7: XML view resulting from query in figure 6 and examples of update operations

operation.

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 and id. The translation is as follows:

```

INSERT INTO VIEWBOOK (id, name, year, title, isbn)
VALUES (1, "Mary Jones", 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 tree that are split into separate relational views, we split the subtree being inserted in the same way. The resulting translation is:

```

INSERT INTO VIEWBOOK (id, name, year, title, isbn)
VALUES (5, "James Perez", 2000, "Updating Relational Views", 999)

```

```

INSERT INTO VIEWPAPER (id, name, year, title, pid)
VALUES (5, "James Perez", 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 above path expression. 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 (see figure 7).

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 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 [5].

## 5 Related Work

There are several works addressing the problem of building XML views from relational databases [21, 25, 4, 10, 26]. 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 [21, 25, 4, 10]. Most of these approaches use extended SQL to build the default view. The exception is XPERANTO [25], 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. We simply supply the `table` function to access 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.

Commercial databases also provide ways of exporting relational data as XML. IBM DB2 XML Extender [11] 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 [19]. SQL Server extends SQL with a directive called `FOR XML` [13]. 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. SQL Server uses *updatagrams* to inform before and after views of XML views generated by an annotated XML Schema that specifies the mapping from the XML view to the underlying 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 [17, 28]. Proposals for updating XML documents stored in relational databases include [27, 28]. 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 [1] 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 to build updatable XML views over relational databases. The main contribution of the paper resides on mapping an XML view to corresponding relational views, and use them to manage updates over the XML view. There are a few open problems in our approach that are worth discussing.

The first one is related to translating insertions to "common" parts of the view. Our present method maps insertions to common parts of the view to one insertion in each corresponding relational view (recall the first example of section 4 - inserting author "Robert White"). When translating these updates to the underlying relational database, the translation mechanism will detect that there are redundant insertions, and will perform only one of them. This postpones the problem to be solved by the algorithm that translates updates to the relational view into updates to the relational database. An alternative to this approach is to detect these cases when generating the INSERT statements - choose one of the views and translate the insertion only once. Another alternative is to generate all the insertions and analyze the generated SQL

statements to remove redundancy. We let this to future work, since both alternatives need careful reasoning about how to correctly detect redundancy.

The second open issue is related to the allowed update operations. Currently, we are allowing only updates that can be automatically mapped to the relational database without causing side effects. Problematic updates are not allowed. A possible solution to this limitation is to use user input for problematic updates. This solution would be based on dialogs with the user, in a way similar to Keller's proposal [23, 2]. However, instead of applying these dialogs at view definition time, we would present them when a problematic update is issued. As an example, the attempt 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 yes, the operation would be performed, otherwise, the operation would be cancelled.

The expressive power of the subset of XQuery we consider in this paper is obviously smaller than the expressive power of XQuery. In particular, our subset is not capable of expressing aggregations and arbitrary structure in the XML view. However, we believe this is a trade off imposed by our goal of updating the relational database through XML views. We have given up expression power to gain updatability.

We are working on a precise characterization of the updatability of views produced by the subset of XQuery proposed here. We believe the results will be similar to the ones we presented in [5]. The main difference resides in dealing with more than one relational view for each XML view.

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

## References

- [1] ARENAS, M., AND LIBKIN, L. A normal form for XML documents. In *Proceedings of PODS 2002* (Madison, Wisconsin, Jun 2002).
- [2] BARSALOU, T., SIAMBELA, N., KELLER, A. M., AND WIEDERHOLD, G. Updating relational databases through object-based views. In *Proceedings of SIGMOD* (Denver, Colorado, 1991), pp. 248–257.
- [3] BOAG, S., CHAMBERLIN, D., FERNANDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. XQuery 1.0: An XML query language. W3C Working Draft, May 2003. <http://www.w3.org/TR/2003/WD-xquery-20030502/>.
- [4] BOHANNON, P., GANGULY, S., KORTH, H., NARAYAN, P., AND SHENOY, P. Optimizing view queries in ROLEX to support navigable result trees. In *Proceedings of VLDB 2002* (Hong Kong, China, Aug. 2002).
- [5] BRAGANHOLO, V., DAVIDSON, S., AND HEUSER, C. On the updatability of XML views over relational databases. In *Proceedings of WEBDB 2003 (to appear)* (San Diego, California, June 2003).
- [6] BRAGANHOLO, V., DAVIDSON, S., AND HEUSER, C. Reasoning about the updatability of XML views over relational databases. Tech. Rep. MS-CIS-03-13, Department of Computer and Information Science, University of Pennsylvania, 2003.
- [7] BRAGANHOLO, V., DAVIDSON, S., AND HEUSER, C. Using XQuery to build updatable XML views over relational databases. Tech. Rep. MS-CIS-03-18, Department of Computer and Information Science, University of Pennsylvania, 2003.

- [8] BRAY, T., HOLLANDER, D., AND LAYMAN, A. Namespaces in XML. W3C Recommendation, Jan 1999. <http://www.w3.org/TR/1999/REC-xml-names-19990114>.
- [9] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. Extensible markup language (xml) 1.0 (second edition). W3C Recommendation, Oct 2002. <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [10] CHAUDHURI, S., KAUSHIK, R., AND NAUGHTON, J. On relational support for XML publishing: Beyond sorting and tagging. In *Proceedings of SIGMOD 2003* (San Diego, California, Jun 2003).
- [11] CHENG, J., AND XU, J. XML and DB2. In *Proceedings of ICDE'00* (San Diego, California, 2000).
- [12] CLARK, J., AND DEROSE, S. Xml path language (xpath) version 1.0. W3C Recommendation, Nov 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116>.
- [13] CONRAD, A. A survey of microsoft sql server 2000 xml features. MSDN Library. Available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxm%1/html/xml07162001.asp>, July 2001.
- [14] DAYAL, U., AND BERNSTEIN, P. A. On the updatability of relational views. In *Proceedings of VLDB 1978* (West Berlin, Germany, Sep 1978), pp. 368–377.
- [15] DAYAL, U., AND BERNSTEIN, P. A. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems* 8, 2 (Sep 1982), 381–416.
- [16] 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.
- [17] DEHAAN, D., TOMAN, D., CONSENS, M., AND OZSU, M. T. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD 2003* (San Diego, California, Jun 2003).
- [18] DRAPER, D., FANKHAUSER, P., FERNÁNDEZ, M., MALHOTRA, A., ROSE, K., RYS, M., SIMÉON, J., AND WADLER, P. XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, May 2003. <http://www.w3.org/TR/2003/WD-xquery-semantics-20030502/>.
- [19] EISENBERG, A., AND MELTON, J. SQL/XML is making good progress. *SIGMOD RECORD* 31, 2 (2002).
- [20] FERNÁNDEZ, M., MORISHIMA, A., SUCIU, D., AND TAN, W. Publishing relational data in xml: the silkroute approach. *IEEE Data Engineering Bulletin* 24, 2 (2001), 12–19.
- [21] FERNÁNDEZ, M., TAN, W.-C., AND SUCIU, D. Silkroute: Trading between relations and XML. In *Nineth Internation World Wide Web Conference* (2000).
- [22] 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.
- [23] KELLER, M. The role of semantics in translating view updates. *IEEE Computer* 19, 1 (1986), 63–73.
- [24] MALHOTRA, A., MELTON, J., AND WALSH, N. 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/>.
- [25] SHANMUGASUNDARAM, J., KIERNAN, J., SHEKITA, E., FAN, C., AND FUNDERBURK, J. Querying XML views of relational data. In *Proceedings of VLDB 2001* (Roma, Italy, Sept. 2001).
- [26] SHANMUGASUNDARAM, J., SHEKITA, E. J., BARR, R., CAREY, M. J., LINDSAY, B. G., PIRAHESH, H., AND REINWALD, B. Efficiently publishing relational data as XML documents. *The VLDB Journal* (2000), 65–76.
- [27] TATARINOV, I., IVES, Z., HALEVY, A., AND WELD, D. Updating XML. In *Proceedings of SIGMOD 2001* (Santa Barbara, California, May 2001).
- [28] TATARINOV, I., VIGLAS, E., BEYER, K., SHANMUGASUNDARAM, J., AND SHEKITA, E. Storing and querying ordered XML using a relational database system. In *Proceedings of SIGMOD 2002* (Madison, Wisconsin, Jun 2002).

## A XQuery EBNF

This section shows the EBNF of the subset of XQuery we discussed in this paper. We use a set of grammar definitions available in the XML documentation. The basic tokens `Letter`, `Digit`, `CombiningChar` and `Extender` are defined in [9]. The identifier `QName` is defined in [8]. Literals and numbers are defined in [3] (`IntegerLiteral`, `DecimalLiteral`, `DoubleLiteral`, `StringLiteral`).

The following is a list of defined tokens for the XQuery grammar. We use the same notation used in [3] for grammar productions.

```
[1] S           ::= WhitespaceChar+
[2] EscapeQuot ::= '","''
[3] VarName     ::= QName
[4] EscapeApos  ::= ''''
[5] WhitespaceChar ::= ([#x0009] | [#x000D] | [#x000A] | [#x0020])
```

The grammar of XQuery is as follows:

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

## B Normalization process for the nest operator

This section presents the normalization process for expressions containing nests. We use the same notation adopted in [18].

$$\begin{aligned} & \left[ \text{nest } \text{Variable}_1 \text{ in } \text{TableExpr}_1, \dots, \text{Variable}_n \text{ in } \text{TableExpr}_n \right. \\ & \left. \text{by } \text{NestVariable}_1 \text{ in } (\text{Variable}_{1_1} / \text{QName}_{1_1} \mid \dots \mid \text{Variable}_{1_m} / \text{QName}_{1_m}), \right. \\ & \quad \dots, \\ & \left. \text{NestVariable}_k \text{ in } (\text{Variable}_{k_1} / \text{QName}_{k_1} \mid \dots \mid \text{Variable}_{k_m} / \text{QName}_{k_m}) \right] \\ & \quad \text{where Expr} \\ & \quad \text{return} \end{aligned}$$

```

<ElName AttName1="{NestVariable1/text()}" ... AttNamek="{NestVariablek/text()}">
  {ElGroup1} ... {ElGroupm} </ElName> ]Nest
  ==
  let Variable'1 := TableExpr1, ..., Variable'n := TableExprn
  for NestVariable1 in distinct-values(Variable1/QName1 | ... | Variable1m/QName1m),
  ..., NestVariablek in distinct-values(Variablek1/QNamek1 | ... | Variablekm/QNamekm)
  return
  <ElName AttName1="{NestVariable1/text()}" ..., AttNamek="{NestVariablek/text()}">
    {for fs:SubVariable(1)
  where fs:SubExpr(1) and (Variable11 = NestVariable1 and ... and Variablek1 = NestVariablek)
    return ElGroup1 }
    ...
    {for fs:SubVariable(m)
  where fs:SubExpr(m) and (Variable1m = NestVariable1 and ... and Variablekm = NestVariablek)
    return ElGroupm }
  </ElName>

```

This normalization process supposes that:

- $\{\text{Variable}_{1_1}, \dots, \text{Variable}_{1_m}, \dots, \text{Variable}_{k_1}, \dots, \text{Variable}_{k_m}\} \subseteq \{\text{Variable}_1, \dots, \text{Variable}_n\}$
- The auxiliary function  $fs:\text{SubVariable}(i)$  returns all variables  $V_x$  referenced in  $ElGroup_i$  and also all variables  $V_y$  appearing in a condition of the form  $V_x/QName_x = V_y/QName_y$  in  $Expr$  in the where clause of nest.
- The auxiliary function  $fs:\text{SubExpr}(i)$  returns every expression specified in  $Expr$  in the where clause of the nest operator that references a variable in  $fs:\text{SubVariable}(i)$ .