University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

January 2003

Using XQuery to Build Updatable XML Views Over Relational Databases

Vanessa P. Braganholo
Universidade Federal do Rio Grande do Sul

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Carlos A. Heuser
Universidade Federal do Rio Grande do Sul

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser, "Using XQuery to Build Updatable XML Views Over Relational Databases", . January 2003.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-03-18.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/28
For more information, please contact repository@pobox.upenn.edu.

Using XQuery to Build Updatable XML Views Over Relational Databases

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.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-03-18.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/28

Using XQuery to build updatable XML views over relational databases*

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

[(1)] Universidade Federal do Rio Grande do Sul - UFRGS

Instituto de Informática

{vanessa, heuser}@inf.ufrgs.br

[(2)] University of Pennsylvania

Department of Computer and Information Science

{susan}@cis.upenn.edu

Abstract

XML has become an important medium for data exchange, and is 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 [20, 24, 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 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`.

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

Author

id	name	email
1	Mary Jones	maryjones@aaa.com
2	Charles Green	charles@bbb.com
3	Michael Kurt	kurt@ccc.com

Conference

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

Ba

author	isbn
1	1234
1	1235
1	1238
2	1234
2	1237
2	1238
3	1235
3	1236

Paper

pid	title	confId	year
IR	Databases and IR	VLDB	2002
QWEB	Querying the Web	DEXA	2000
WEB	Web Survey	VLDB	2001

Pa

author	pid
1	IR
1	QWEB
1	WEB
2	IR
2	WEB
3	WEB

Book

isbn	title	year
1234	Book1	2000
1235	Book2	2001
1236	Book3	2000
1237	Book4	2001
1238	Book5	2001

CONSTRAINTS
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

Figure 1: Sample database

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.

In previous work [5], we addressed this problem by considering the nested relational algebra (NRA) [21] 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, 22], 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 [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`): 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

```
<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>
   ...
</author>
...
</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

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 with the following abstract type τ:*

$\tau = E_0$: $\{E_1 : \tau_T\}, \ldots, \{E_n : \tau_T\}$, $n \geqslant 1$

$\tau_T = [E_1 : \tau_1, \ldots, E_k : \tau_k, \{E_{k+1} : \tau_T\}, \ldots, \{E_{k+m} : \tau_T\}]$ *($k \geqslant 1$, $m \geqslant 0$) and τ_i ($1 \leqslant i \leqslant k$) is τ_S or τ_C.*

$\tau_C = [E_1 : \tau_1, \ldots, E_k : \tau_k]$ *($k \geqslant 1$) and τ_i ($1 \leqslant i \leqslant k$) is τ_S or τ_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_i ($i \geqslant 1$) 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 ($E_1, ..., E_n$ in the definition of τ and $E_{k+1}, ...E_{k+m}$ in the definition of τ_T). We adopt the convention that attribute names start with "@".

PROPOSITION 1 *UXQuery views are always well behaved.*

Proof: The proof of this proposition is based on the syntax and semantics of UXQuery. We postpone this proof until the end of this section. ■

In our mapping approach, it will be important to recognize nodes of type τ_T all of whose descendants are non-repeating nodes (τ_S and/or τ_C). For this reason, we will rename such τ_T nodes 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)
          ]}
      ]
   }
```

Throughout this paper, we use the convention of referring to the abstract type of an element by the abstract type that was used to generate it followed by the element name. As an example, the abstract type of the element *book* is referred to as τ_N*(book)*, and its type (DTD) is [*@isbn: CDATA, title: #PCDATA*].

PROPOSITION 2 *The type of any well behaved view must contain at least one τ_N node.*

Proof: The proof of this proposition follows from the definition of well-behaved views (definition 1). Well-behaved views have abstract type τ, where $\tau = E_0$: $\{E_1 : \tau_T\}, \ldots, \{E_n : \tau_T\}$, $(n \geqslant 1)$. Since the condition $(n \geqslant 1)$ is imposed, the smallest possible well-behaved view has the form $\tau = E_0$: $\{E_1 : \tau_T\}$. Now it is necessary to analyze the structure of E_1, whose type is τ_T. From definition 1, we have that $\tau_T = [E_1 : \tau_1, \ldots, E_k : \tau_k, \{E_{k+1} : \tau_T\}, \ldots, \{E_{k+m} : \tau_T\}]$ $(k \geqslant 1,\ m \geqslant 0)$ and τ_i $(1 \leqslant i \leqslant k)$ is τ_S or τ_C. Again, lets take the minimal scenario, where ($k = 1$ and $m = 0$). We therefore have $\tau_T = [E_1 : \tau_1]$, where τ_1 is τ_S or τ_C. Supposing element names A, B and C, the type of an well-behaved view conforming to this structure would be:

$\tau = A$: $\{B : \tau_T(B)\}$

$\tau_T(B) = [C : \tau_S]$

As defined previously, τ_N nodes are nodes of type τ_T such that all of their descendants are non-repeating nodes (τ_S and/or τ_C). Clearly, this condition is satisfied by node B, and consequently its type is renamed to $\tau_N(B)$. Therefore, the simplest possible well-behaved view has one node of type τ_N.

For more complicated views, it is obvious that there will be at least one node of type τ_N. It turns out that a well-behaved view has to have at least one node of type τ_T. The definition of type τ_T is recursive (τ_T can be composed of nodes τ_T, τ_S and/or τ_C). The recursion only ends when all children of a node with type τ_T are τ_S and/or τ_C. Since this is the condition to a node τ_T to be renamed to τ_N, this proves that every well-behaved view has at least one node τ_N. ■

PROPOSITION 3 *There is at most one τ_N node along any path from the root to a leaf in the abstract type of a well behaved view.*

Proof: In this proof, we consider a bottom-up path from any leaf node to the document root. For this proposition to hold, there must be at most one node τ_N along this path.

Analyzing definition 1, it's easy to see that:

- Leaf nodes have type τ_S. The parent of a leaf node can have types τ_C, τ_T or τ_N.
- Nodes of type τ_C can have parent of types τ_C, τ_T or τ_N.
- Nodes of type τ_N can only have parent of type τ_T or τ, which is the type of the document root. The parent of a node of type τ_N can never have type τ_N, since by definition, nodes of type τ_N are the ones whose children have type τ_S and/or τ_C.
- Nodes of type τ_T can only have parent of type τ_T or τ, which is the type of the document root.

Based on these rules, starting from a leaf node, we can have the following path p to the document root (we show just the types of nodes in p). We denote a step in p by "," (comma), since we are going bottom-up – using "/" would probably create confusion.

$p = \tau_S,\tau_{C_1},...,\tau_{C_n},\tau_{T_1},...,\tau_{T_m},\tau$ ($n \geqslant 0$ and $m \geqslant 1$).

In this path, the only candidate of having type τ_N is the node whose type is τ_{T_1}. This node will be renamed to τ_N if all of its children are of type τ_C and/or τ_S. Despite not being possible to determine the type of the children of this node just by looking at p, it is enough to know that this is the only node that can have type τ_N. Consequently, as we wanted to demonstrate, there is at most one node of type τ_N in p. ■

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 (*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 `nest`, which facilitates the construction of heterogeneous nested sets.

```
 1. <conferencePapers>
 2.   {for $c in table("conference")
 3.    return
 4.    <conference id="{$c/confId/text()}">
 5.       {$c/confName}
 6.       {nest $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 nest operator (lines 1-22) and its translation to regular XQuery syntax (lines 23-45)

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.

Semantics of `table()`. XQuery has three input functions: `input()`, `collection()` and `document()` [23]. 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>
...
```

We translate this input function to XQuery as follows.

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

Semantics of `Nest`. The `nest` 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 *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 syntax for `nest` is defined by the following EBNF:

```
[31] Nest       ::= NestClause ByClause WhereClause "return" Header
[32] NestClause ::= "nest "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[33] ByClause   ::= "by" "$" VarName "in" UnionExpr ("," "$" VarName "in" UnionExpr)*
[34] Header  ::= "<" QName (QName "=" NestAttValue)+ ">" ( "{" ElGroup "}" )+ "</" QName S? ">"
               | "<" QName ">" ( "{" "$" VarName "}" | "<" QName ">" "{" "$" VarName "/" TextTest "}" "</" QName ">")+
                 ( "{" ElGroup "}" )+ "</" QName S? ">"
[35] NestAttValue ::= "'" "{" "$" VarName "/" TextTest "}" "'"
                    | '"' "{" "$" VarName "/" TextTest "}" '"'
[36] ElGroup    ::= ElmtConstructor
[37] 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 1-21) 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`. This is possible because before processing it, we add the function definition in the top of the query.

Continuing with the example, the `nest` 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 `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 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.

Having presented the semantics and syntax of UXQuery, we are now able to prove proposition 1.

Proof of Proposition 1: An UXQuery view always have a root. This root is produced by the first two productions in UXQuery grammar:

```
[1] UXQuery ::= QueryBody
[2] QueryBody ::= ElmtConstructor
```

The `ElmtConstructor` production produces a node of type τ, which is the document root. By definition 1, $\tau = E_0$: $\{E_1 : \tau_T\}, \ldots, \{E_n : \tau_T\}$, $n \geqslant 1$. E_0 is the name of the element constructed by `ElmtConstructor`.

Now it is necessary to make sure that all children of E_0 are of type τ_T. The grammar production of `ElmtConstructor` is:

```
[3]  ElmtConstructor ::= "<" QName AttList "/>" | "<" QName AttList? ">" ElmtContent+ "</" QName S? ">"
```

This production is a choice. However, the first option in this choice (`"<" QName AttList "/>"`) can not be taken at this point, since `AttList` is expanded to `PathExprAtt`:

```
[8]  PathExprAtt ::= "$" VarName "/" QName "/" NodeTest
```

This production references a variable, and since at this point there is no clause that bounds the variable to a table, this construction is not semantically correct. Variables are bound only in `FWRExpr` or `Nest` productions.

Consequently, the only possible choice in production `[3]` at this point is `"<" QName AttList? ">" ElmtContent+ "</" QName S? ">"`. Here, `AttList` will be empty for the same reason explained above. The children of E_0 are then determined by `ElmtContent+`. Since this production is marked with "+", it is guaranteed that E_0 will have at least one child, that is, the restriction ($n \geqslant 1$) in the definition of E_0's type is guaranteed by the EBNF. Each repetition of this production will generate an element E_i, children of E_0. Remember that E_0: $\{E_1 : \tau_T\}, \ldots, \{E_n : \tau_T\}$, ($n \geqslant 1$). In this case, ($1 \leqslant i \leqslant n$).

The production that defines `ElmtContent` is:

```
[4]  ElmtContent ::= ElmtConstructor | EnclosedExpr+
```

Again, we have a choice. However, choosing the first option would lead to a non well-behaved view. For this reason, we consider this a semantic error, and the parser raises an exception if this production is chosen at this point. Thus, the only correct choice is `EnclosedExpr`.

```
[10] EnclosedExpr ::= "{" (FWRExpr | PathExpr | Nest) "}"
```

E_i must be a repeating element of type τ_T. The repetition is determined by the use of "{ }" in the definition of E_0. Repeating elements are generated by `FWRExpr` or `Nest`. At this point, since we still do not have variable bindings, the production `PathExpr` can not be chosen because a path expression contains a variable reference:

```
[17] PathExpr ::= "$" VarName "/" QName ("/" NodeTest)?
```

As a result, the children of E_0 are constructed by production `FWRExpr` or `Nest`. Let's analyze both alternatives.

1(a) FWRExpr: A `FWRExpr` has the form:

```
[14] FWRExpr ::= ((ForClause)+ WhereClause? OrderByClause? "return")* ElmtConstructor
```

In this case, the element name of E_i is determined by `ElmtConstructor`.

1(b) Nest: A `Nest` has the form:

```
[31] Nest ::= NestClause ByClause WhereClause "return" Header
```

In this case, the element name of E_i is determined by `Header`.

Let's review what we have until now: $\tau = E_0$: $\{E_1 : \tau_T\}, \ldots, \{E_n : \tau_T\}$, $(n \geqslant 1)$. Since each E_i $(1 \leqslant i \leqslant n)$ has type τ_T, it is necessary to analyze the structure of this type and make sure that UXQuery produces it correctly.

According to definition 1, $\tau_T = [F_1 : \tau_1, \ldots, F_k : \tau_k, \{F_{k+1} : \tau_T\}, \ldots, \{F_{k+m} : \tau_T\}]$ $(k \geqslant 1,\ m \geqslant 0)$ and τ_j $(1 \leqslant j \leqslant k)$ is τ_S or τ_C.

As shown previously, node E_i of type τ_T is constructed by `FWRExpr` or `Nest`. Consequently, the structure of its type is determined by `ElmtConstructor` (in case the node was generated by a `FWRExpr`) or `Header` (in case the node was generated by a `Nest`). Again, we analyze both cases:

2(a) ElmtConstructor: `ElmtConstructor` is defined as:

```
[3]  ElmtConstructor ::= "<" QName AttList "/>" | "<" QName AttList? ">" ElmtContent+ "</" QName S? ">"
```

Here, `QName` corresponds to the name of element E_i. Since we have already variables bindings, any of these two options are valid. In the first case (`"<" QName AttList "/>"`), the children of E_i are generated by `AttList`.

```
[5]  AttList ::= (S (QName S? "=" S? AttValue)?)+
```

Attributes are atomic elements, so their type is τ_S. The "+" in the definition of `AttList` indicates that there must be at least one attribute. In order words, E_i must have at least one child of type τ_S. This conforms to the definition of E_i's type ($\tau_T = [F_1 : \tau_1, \ldots, F_k : \tau_k, \{F_{k+1} : \tau_T\}, \ldots, \{F_{k+m} : \tau_T\}]$ $(k \geqslant 1,\ m \geqslant 0)$ and τ_j $(1 \leqslant j \leqslant k)$ is τ_S or τ_C). In this case, we ensure the restriction $(k \geqslant 1)$, and τ_j is τ_S. Also, there is no repeating child, so $(m = 0)$.

The second case is (`"<" QName AttList? ">" ElmtContent+ "</" QName S? ">"`). In this case, if `AttList` is present, it generates children of type τ_S. However, since `AttList` is optional, we must analyze the structure produced by `ElmtContent+`.

```
[4]  ElmtContent ::= ElmtConstructor | EnclosedExpr+
```

Again, we have a choice. Let's analyze `EnclosedExpr`. According to production `[10]`, it can be `FWRExpr`, `PathExpr` or `Nest`. As seen before, `FWRExpr` and `Nest` produces children of type τ_T. So in this case, we have that E_i have repeating children, and consequently $(m > 0)$. A path expression produces a leaf node of type τ_S, so we have $(k \geqslant 1)$. However, since `EnclosedExpr` is a choice, it may be the case that its content does not have any `PathExpr`, and in this case, our condition $(k \geqslant 1)$ would fail. This is a semantic restriction, and it is enforced by the UXQuery parser. Since `ElmtContent` has a "+" sign, E_i can have several children resulting from the repetition of this rule. If after the parsing process E_i does not have any children of type τ_S or τ_C, the parser raises an exception.

Another option for the content of `ElmtContent` is `ElmtConstructor`. This production generates an element F_j, children of E_i, whose type is determined by production `[3]`. The determination of F_j's type is done recursively by rule 2(a). Notice that F_j will have type τ_C if `ElmtConstructor` derives only `PathExprs` and/or `AttList`. If a single `PathExpr` is derived from `ElmtConstructor`, them F_j has type τ_S.

2(b) Header: `Header` is defined as:

```
[34] Header ::= "<" QName (QName "=" NestAttValue)+ ">" ( "{" ElGroup "}" )+ "</" QName S? ">"
        | "<" QName ">"
          (( "{" "$" VarName "}")
            | ("<" QName ">" "{" "$" VarName "/" TextTest "}" "</" QName ">"))+
          ( "{" ElGroup "}" )+ "</" QName S? ">"
```

In this case, there are two options, but they are equivalent. The first one places the nest values as attributes of E_i, and the second one places the nest values as subelements of E_i. These attributes or subelements are of type τ_S. Since they have a "+", we guarantee that $(k \geqslant 1)$. When E_i is constructed by a `Header`, it is also guaranteed that it has at least one child of type τ_T. This is the child constructed by `ElGroup` (notice the "+" sign in this production). So we have $(m \geqslant 1)$. The name of this child is determined by `ElmtConstructor`, since we have:

```
[36] ElGroup ::= ElmtConstructor
```

The type of this child is determined recursively by rule 2(a).

As we can see, the EBNF and also the semantics of UXQuery guarantee the production of a view whose type conforms to definition 1. ■

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 query tree has a name and possible annotation, and each leaf node in the tree has a name and a value.

```
<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>
```

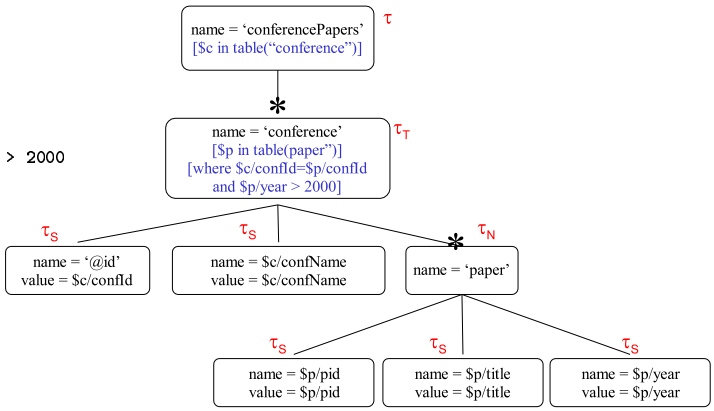

Figure 4: Example of UXQuery that joins two relations and its auxiliary query tree

DEFINITION 2 *An auxiliary query tree is a tree whose nodes represent element tags and whose edges are either simple or *-edges which reflect element-subelement relationships in the result. Each interior node is annotated with the variable bindings and conditions that were introduced at that level. Each leaf node is annotated with a path from which to construct its value.*

To illustrate, we give a simple example that does not have the `nest` 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 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 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 explicitly specify an XML element, and the value of its content using a variable (e.g `<name>{$c/confName/text()}</name>`), or we can specify the entire element using a variable (e.g `{$c/confName}`). 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 `nest`. Additionally, the root node of an *element group* within a `nest` 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:

```
<conferencePapers>
  {for $c in table("conference")
   return
   <conference id="{$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}
           </paper>}
       </papers>
      }
   </conference>
  }
</conferencePapers>
```

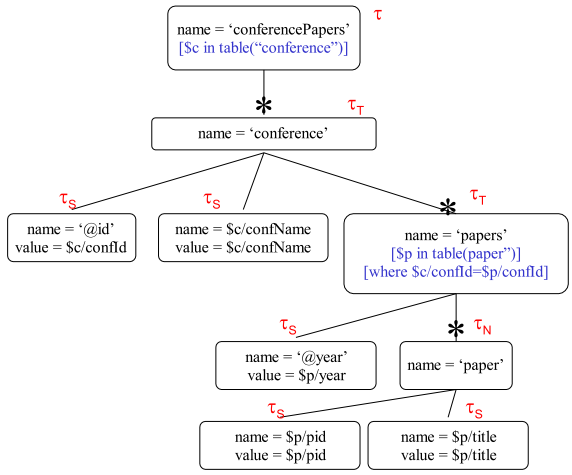

Figure 5: Example of UXQuery that nests elements, and its corresponding auxiliary query tree

```
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 `nest` are constructed in a slightly different manner. Annotations of variables and where clauses within a nest expression are placed on the node that represents the *header* element of the `nest` 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 should be mapped to distinct relational views. Information on levels above is considered to be common to

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

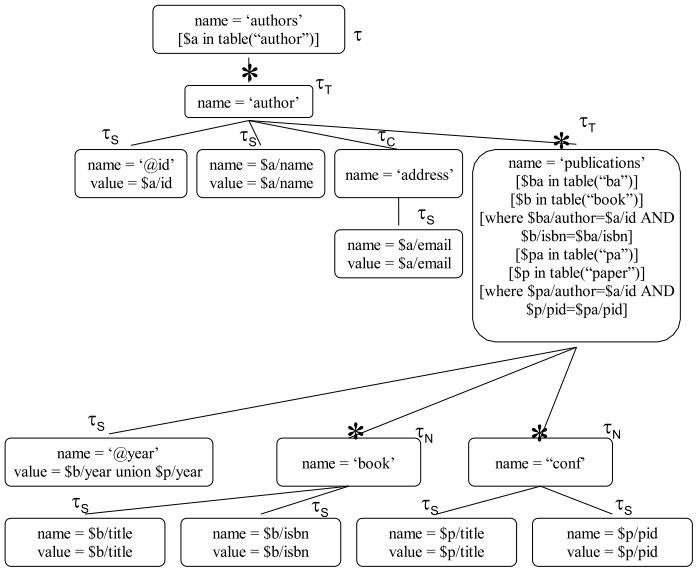

Figure 6: Example of UXQuery that mixes information of different relations in the same nesting level and the corresponding auxiliary query tree

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 appendix C.

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 out of 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 3 *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 [5]. Basically, an update operation is a triple $<u,\Delta,ref>$, where u is the type of operation (insert, delete, modify); Δ is the XML tree to be inserted, or (in case of a modification)

```
<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

an atomic value; and *ref* is a simple path expression in XPath [11] 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.

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).

```
<authors>
  <author id="1">
    <name>Mary Jones</name>
    <address>
       <email>maryjones@aaa.com</email>
    </address>
    <publications year="2000">
      <book><title>Book1</title><isbn>1234</isbn></book>
      <conf>
        <title>Querying the Web</title><pid>QWEB</pid>
      </conf>
    </publications>
    <publications year="2001">
      <book><title>Book2</title><isbn>1235</isbn></book>
      <book><title>Book5</title><isbn>1238</isbn></book>
      <conf><title>Web Survey</title><pid>WEB</pid></conf>
    </publications>
    <publications year="2002">
      <conf>
        <title>Databases and IR</title><pid>IR</pid>
      </conf>
    </publications>
  </author>
  <author id="2">
    <name>Charles Green</name>
    <address>
       <email>charles@bbb.com</email>
    </address>
    <publications year="2000">
      <book><title>Book1</title><isbn>1234</isbn></book>
    </publications>
  ...
  </author>
  ...
</authors>
```

U_1: $u = insert$, ref = `/authors`,

```
Δ = {<author id="4">
        <name>Robert White</name>
        <address>
           <email>white@zzz.com</email>
        </address>
     </author>}.
```

U_2: $u = insert$,
ref = `//author[@id="1"]/publications[@year="2000"]`,

```
Δ = {<book>
        <title>Book6</title><isbn>9888</isbn>
     </book>}.
```

U_3: $u = insert$, ref = `/authors`,

```
Δ = {<author id="5">
        <name>James Perez</name>
        <address>
           <email>james@zzz.com</email>
        </address>
        <publication year="2000">
           <book>
             <title>Updating Relational Views</title>
             <isbn>999</isbn></book>
           <conf>
             <title>Views and XML</title>
             <pid>VIEW</pid></conf>
        </publication>
     </author>}.
```

U_4: $u = modify$, $\Delta = \{$Querying the Web using XML$\}$,
ref = `//book[isbn="1234"]/title`.

U_5: $u = delete$,
ref = `//author[@id="1"]/publications[@year="2000"]`.

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

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:

```
INSERT INTO VIEWBOOK (id, name, email, year, title, isbn)
VALUES (5, "James Perez", "james@zzz.com", 2000, "Updating Relational Views", 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 (see figure 8).

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
```

The algorithms for translating updates in the XML view to updates in the corresponding relational views are presented in appendix C.3.

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].

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 exits (for a summary of their algorithms, please refer to [6]).

5 Related Work

There has been a lot of work addressing the problem of building XML views from relational databases [20, 24, 4, 9, 25]. 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 [20, 24, 4, 9]. Most of these approaches use extended SQL to build the default view. The exception is XPERANTO [24], 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 [10] 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. 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 [12]. 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 [20, 17, 27]. Proposals for updating XML documents stored in relational databases include [26, 27]. 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, 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 first 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 which inserts the author "Robert White"). When translating these updates to the underlying relational database, redundant insertions are generated. However, the relational system will only perform one of them and the others will fail. Thus, we rely on the relational system to eliminate redundant updates. 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 (over the base tables) to remove redundancy. We leave 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 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 [22, 2]. 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] 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* (San Diego, California, June 2003), pp. 31–36.

[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] Bray, T., Hollander, D., and Layman, A. Namespaces in XML. W3C Recommendation, Jan 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114.

[8] 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.

[9] 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).

[10] Cheng, J., and Xu, J. XML and DB2. In *Proceedings of ICDE'00* (San Diego, California, 2000).

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

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

[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%l/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., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W.-C. Silkroute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS) 27*, 4 (Dec 2002), 438–493.

[21] 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.

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

[23] 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/.

[24] 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).

[25] 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.

[26] Tatarinov, I., Ives, Z., Halevy, A., and Weld, D. Updating XML. In *Proceedings of SIGMOD 2001* (Santa Barbara, California, May 2001).

[27] 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 UXQuery EBNF

In the definitions of this section we use a set of grammar definitions available in the XML documentation. The basic tokens `Letter`, `Digit`, and `S` (whitespace) are defined in [8]. The identifier `QName` is defined in [7]. Literals and numbers are defined in [3] (`IntegerLiteral`, `DecimalLiteral`, `DoubleLiteral`, `StringLiteral`).

```
[1]  UXQuery         ::= QueryBody
[2]  QueryBody       ::= ElmtConstructor
[3]  ElmtConstructor ::= "<" QName AttList "/>" | "<" QName AttList? ">" ElmtContent+ "</" QName S? ">"
[4]  ElmtContent     ::= ElmtConstructor | EnclosedExpr+
[5]  AttList         ::= (S (QName S? "=" S? 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 ::= "nest "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[33] ByClause   ::= "by" "$" VarName "in" UnionExpr ("," "$" VarName "in" UnionExpr)*
[34] Header     ::= "<" QName (QName "=" NestAttValue)+ ">" ( "{" ElGroup "}" )+ "</" QName S? ">"
                  | "<" QName ">" (( "{" "$" VarName "}") | ("<" QName ">" "{" "$" VarName "/" TextTest "}" "</" QName ">"))+
                    ( "{" ElGroup "}" )+ "</" QName S? ">"
[35] NestAttValue ::= "'" "{" "$" VarName "/" TextTest "}" "'"
                    |  '"' "{" "$" VarName "/" TextTest "}" '"'
[36] ElGroup    ::= ElmtConstructor
[37] UnionExpr  ::= "(" "$" VarName "/" QName ( ("union" | "|")  "$" VarName "/" QName)* ")"
```

B Normalization process for the nest operator

The notation for the normalization process is the same adopted in [18].

```
[nest 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> ]_Nest
==
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 supposes that:

- $\{\text{Variable}_{1_1}, \ldots, \text{Variable}_{1_m}, \ldots, \text{Variable}_{k_1}, \ldots, \text{Variable}_{k_m}\} \subseteq \{\text{Variable}_1, \ldots, \text{Variable}_n\}$
- 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 `nest` operator; **cmp** $\in \{" = ", " < ", " > ", "! = ", " <= ", " >= "\}$.
- The auxiliary function *fs:SubExpr(i)* returns every expression specified in *Expr* in the `where` clause of the `nest` operator that references a variable in *fs:SubVariable(i)*.

C Algorithms

This section presents the algorithms described in section 3.

C.1 Auxiliary query tree

The algorithm *auxiliary-tree(el)* constructs the auxiliary query tree from a given UXQuery. The parameter *el* is the XML element specified as the root of the query. The algorithm is recursive, and each execution builds a single node of the auxiliary query tree.

```
auxiliary-tree(el)

Create node(n)
if el is element then
  n.name = el
else
  n.name = '@' + el;
end if
Let X denote the set of XML elements constructed as a direct child or attribute of el
if X is empty then
  if el contains a nesting variable V then
    n.value = expression that V is bound to
```

```
    else
      if el is element then
        n.value = el
      else
        n.value = el - "/text()"
      end if
    end if
  else
    Let A denote all variable bindings bellow el and above the next non-leaf element, ignoring variables inside nests
    for each a in A do
      annotate n(a)
    end for
    Let W denote all where conditions bellow el and above the next non-leaf element, ignoring conditions inside nests
    for each w in W do
      annotate n(w)
    end for
    if el is a header of a nest then
      Let N denote the nest expression of which el is the header
      Let A denote all variable bindings in N
      for each a in A do
        annotate n(a)
      end for
      Let W denote all where conditions in N
      for each w in W do
        annotate n(w)
      end for
    end if
    for each x in X do
      n1 = auxiliary-tree(x)
      n1.parent = n
    end for
  end if
  return n
```

C.2 Mapping an auxiliary query tree to a set of relational views

The *map(aux)* algorithm maps a given auxiliary query tree to a set of corresponding relational views. It works as follows: generates one sub-tree for each relational view that should be produced. Then it calls the procedure *generate-relational-view* for each one of these subtrees.

The parameter for the *map* algorithm is the root of the auxiliary query tree produced by the *auxiliary-tree* algorithm.

```
map(aux)

Let i be the number of subtrees corresponding to the auxiliary tree aux
Initialize i with 0
Let Z denote the set of nodes in aux
for each node z in Z do
  Let X be the set of non-leaf direct children of aux
  if size(X) > 1 then
    for each x in X do
      inc i
      n = x.parent
      Create node(ni) = clone(n)
      x.parent = ni
      delete-annotation(ni,x)
      ri = n.parent
      while ri <> root(aux) do
        Create node(c) = clone(ri)
        {clone ri's children}
        Let Y be the set of direct children of the node ri (except n)
        for each y in Y do
          Create node(c') = clone(y)
          c'.parent = c
        end for
        {if it's the first time, connect ni with c}
        if ri = n.parent then
```

```
            n_i.parent = c
          end if
          r_i = r_i.parent
        end while
        Create note(c') = clone(r_i)
        c.parent = c'
        r_i = c
        {Deal with nest attributes}
        Let Nest be the set of all leaf children of n
        for each nest in Nest do
          Create node(z) = clone(nest)
          z.parent = n_i
          delete-value(z,n_i)
        end for
      end for
    end if
end for
if i = 0 then
  generate-relational-view(aux)
else
  for j from 1 to i do
    generate-relational-view(r_j)
  end for
end if
```

The algorithm *map* uses the sub-routines *delete-annotation(n,x)* and *delete-value(n,x)*.

delete-annotation(n,x)

```
{deletes all the annotations in n that are not related with x}
Let W be the set of annotations of where clauses in n
Let C be the set of children of n
Let V be the variables referenced in the value of the nodes in C
Let V' = V
for each w in W do
  if (w has the form V_x/QName_x cmp V_y/QName_y,
  where V_x, V_y are variables, V_x ∈ V,
  cmp ∈ {"=", "<", ">", "!=", "<=", ">="}) then
    V' = V' + V
  end if
end for
for each w that does reference a variable in V' do
  delete-where-annotation(w,W)
end for
Let A be the set of annotations of variable bindings in n
Let W' be the set of where clauses in n {this set was modified by the previous loop}
for each a in A do
  if a does not reference a variable in W' then
    delete(a)
  end if
end for
```

The algorithm *delete-annotation* uses the algorithm *delete-where-annotation*.

delete-where-annotation(w,W)

```
if size(W) > 1 then
  if w is connected in W by an expression e of the form e_1 and w then
    replace e by e_1
  else
    if w is connected in W by an expression e of the form w and e_1 then
      replace e by e_1
    else
      if w is connected in W by an expression e of the form w or e_1 then
        replace e by e_1
      else
        if w is connected in W by an expression e of the form e_1 or w then
          replace e by e_1
```

```
        end if
      end if
    end if
  end if
end if
```

delete-value(n,x)

```
{deletes variable references in n that is not in x}
Let n.value be v1/QName union v2/QName union ... union vk/QName
Let A be the set of variable bindings in x
for i from 1 to k do
  if vi ∉ A then
    delete (vi)
  end if
end for
Delete words "union" {in the end, there will be only one variable in n.value - this is guaranteed by the nest normalization process}
```

The algorithm *generate-relational-view* generates an SQL statement corresponding to a relational view. The parameter t is a sub-tree produced be the *map* algorithm.

generate-relational-view(t)

```
String view, select, from, where
select = "SELECT "
Let L denote the set of leaf nodes children of t
for each l in L do
  if l is attribute then
    delete "@" from l.name
  else
    delete "$*/" from l.name
  end if
  replace "/" by "." in l.value
  if select <> "SELECT " then
    select = select + ", " + l.value + " AS " + l.name
  else
    select = select + " " + l.value + " AS " + l.name
  end if
end for
from = " FROM "
Let J = "";
Let newJoin = false;
Let nl = t;
Let A be the set of variable bindings in nl
Let a be a variable in A
if nl has a non-leaf child c then
  if size(A) > 1 then
    a = first variable binding in A
  end if
  from = "(" + a.tablename + " AS " + a.variablename + " LEFTJOIN "
  mark(a); {mark a as already used}
  while c is not NULL do
    Let VC be the set of variable bindings in c
    Let W be the set of where annotations in nl and c
    while there is an unmarked variable vc in VC do
      Let joinCond = ""
      if not(newJoin) then
        v = findJoinTable(a, W, VC, joinCond)
        {joinCond returns with the join condition found by the procedure findJoinTable}
        from = from + v.tablename + " AS " + v.variablename + " ON " + joinCond + ") "
        newJoin = true
        a = v
        mark(v)
        J = J + joinCond
      else
        v = findJoinTable(a, W, VC, joinCond)
        from = "(" + from + " LEFT JOIN " + v.tablename + " AS " + v.variablename + " ON " + joinCond + ")
        a = v
```

```
        mark(v)
        J = J + joinCond
      end if
    end while
    nl = c
    c = non-leaf child of nl
  end while
else
  if size(A) = 1 and nl has no leaf child then
    from = a.tablename + " AS " + v.variablename
  else
    if size(A) > 1 and nl has no leaf child then
      Let W be the set of where annotations in nl
      a = first variable binding in A
      from = "(" + a.tablename + " AS " + a.variablename + " LEFTJOIN "
      mark(a)
      Let newJoin = false
      while there is an unmarked variable in A do
        if not(newJoin) then
          v = findJoinTable(a,W,A,joinCond)
          from = from + v.tablename + " AS " + v.variablename + " ON " + joinCond + ")"
          newJoin = true
          a = v
          mark(v)
          J = J + joinCond
        else
          v = findJoinTable(a,W,A,joinCond)
          from = "(" + from + " LEFT JOIN " + v.tablename + " AS " + v.variablename + " ON " + joinCond + ")"
          a = v
          mark(v)
          J = J + joinCond
        end if
      end while
    end if
  end if
end if
where = "WHERE "
Let W be the set of where annotations in all the tree
for each w in W that is not in J do
  where = where + w;
end for
view = select + " " + from + " " + where
return view
```

The *generate-relational-view* algorithm calls the subroutine *findJoinTable*.

findJoinTable(a,W,VC,$joinCond$)

```
joinCond = ""
for each v in VC do
  if there is a w in W of the form V_a/QName_a cmp V_v/QName_v,
  (where V_a, V_v are variables, V_a ∈ a, V_v ∈ v
  cmp ∈ {"=", "<", ">", "!=", "<=", ">="}) then
    for each w in W that involves variables a and v do
      joinCond = joinCond + w + ";"
    end for
    exit {exit the for loop}
  end if
end for
{treats the join expression including ANDs and ORs}
for each w_1 ; w_2 in joinCond do
  if w_1 and w_2 are connected in W by an expression of the form "w_1 and e and w_2" then
    replace ; by and
  else
    if w_1 and w_2 are connected in W by an expression of the form "w_1 or e or w_2" then
      replace ; by or
    else
      if w_1 and w_2 are connected in W by an expression of the form "w_1 or e and w_2" then
        replace ; by or
```

```
        else
          if w1 and w2 are connected in W by an expression of the form "w1 and e or w2" then
            replace ; by and
          end if
        end if
      end if
    end if
  end for
  return joinCond
  return v
```

C.3 Mapping updates on the XML view to updates in the relational views

C.3.1 Insertions

The algorithm *translateInsert* takes an insertion specification against the XML view and translates it to insertions on the corresponding relational views. The parameters are:

- V: the XML view
- aux: the auxiliary query tree
- ref: the update path
- Δ: the subtree to be inserted

The procedure supposes that the update specification was already checked for schema conformance. The subroutine *type(n)* returns the type of a given node (τ_N, τ_T, τ_C or τ_S). The subroutine *view(n)* returns the name of the relational view associated with node n, assuming that n is of type τ_N.

```
translateInsert(V, aux,ref,Δ)

Insert Δ in V using ref as insertion point. Δ must be inserted under every node resulting from the evaluation of ref in V
Let p be the unqualified portion of ref
Let m be the node resulting from the evaluation of p against aux
Let N be the set of nodes resulting from the evaluation of ref in V
for each n in N do
  if type(m) = τN then
    generateInsertSQL(view(m), root(Δ), n, V)
  else
    Let X be the set of nodes of type τN in Δ
    for each x in X do
      generateInsertSQL(view(x), x, n, V)
    end for
  end if
end for
```

The algorithm *translateInsert* calls the subroutine *generateInsertSQL*, which generates a single SQL insert statement. The parameters are:

- $RelView$: the relational view where the insertion is to take place
- r: the root of the subtree being inserted
- $InsertionPoint$: the node in V under which the subtree is being inserted
- V: the XML view

```
generateInsertSQL(RelView, r, InsertionPoint, V)

sql = "INSERT INTO" + RelView + getAttributes(RelView)
sql = sql + " VALUES ("
for i = 0 to getTotalNumberAttributes(RelView) - 1 do
  att = getAttribute(RelView, i)
  if att is a child n of r then
    sql = sql + getValue(n)
  else
    Find att in V, starting from InsertionPoint examining the leaf nodes until V's root is found
```

```
      Let the node found be m
      sql = sql + getValue(m)
    end if
    if i < getTotalNumberAttributes(RelView) - 1 then
      sql = sql + ", "
    else
      sql = sql + ")"
    end if
  end for
```

C.3.2 Deletions

The algorithm *translateDelete* takes a deletion specification against the XML view and translates it to deletions on the corresponding relational views. The parameters are:

- V: the XML view
- aux: the auxiliary query tree
- ref: the update path

The procedure supposes that the update specification was already checked for schema conformance. The subroutine *type(n)* returns the type of a given node (τ_N, τ_T, τ_C or τ_S). The subroutine *view(n)* returns the name of the relational view associated with node n, assuming that n is of type τ_N.

```
translateDelete(V, aux,ref)

Let p be the unqualified portion of ref
Let m be the node resulting from the evaluation of p against aux
if type(m) = τ_N then
  generateDeleteSQL(view(m), ref)
else
  Let X be the set of nodes of type τ_N under m
  for each x in X do
    generateDeleteSQL(view(x), ref)
  end for
end if
```

The algorithm *translateDelete* calls the subroutine *generateDeleteSQL*, which generates a single SQL delete statement. The parameters are:

- $RelView$: the relational view where the insertion is to take place
- ref: the update path

```
generateDeleteSQL(RelView, ref)

sql = "DELETE FROM " + RelView + " WHERE "
for each filter f in ref do
  if f is the first filter in ref then
    sql = sql + f
  else
    sql = sql + "AND " + f
  end if
end for
```

C.3.3 Modifications

The algorithm *translateModify* takes a modification specification against the XML view and translates it to modifications on the corresponding relational views. The parameters are:

- V: the XML view
- aux: the auxiliary query tree
- ref: the update path

- Δ: the new value

The procedure supposes that the update specification was already checked for schema conformance. The subroutine *type(n)* returns the type of a given node (τ_N, τ_T, τ_C or τ_S). The subroutine *view(n)* returns the name of the relational view associated with node n, assuming that n is of type τ_N.

```
translateModify(V, aux,ref,Δ)

Let p be the unqualified portion of ref
Let m be the node resulting from the evaluation of p against aux
Let r be the ancestor of m whose type is τ_T or τ_N
Let N be the set of nodes resulting from the evaluation of ref in V
for each n in N do
  if type(r) = τ_N then
    generateModifySQL(view(r), Δ, ref)
  else
    Let X be the set of nodes of type τ_N under r
    for each x in X do
      generateModifySQL(view(x), Δ, ref)
    end for
  end if
end for
```

The algorithm *translateModify* calls the subroutine *generateModifySQL*, which generates a single SQL update statement. The parameters are:

- $RelView$: the relational view where the insertion is to take place
- $Delta$: the new value
- ref: the update path

```
generateDeleteSQL(RelView, Δ, ref)

sql = "UPDATE " + RelView + " SET "
Let t be the terminal node in ref
sql = sql + t + "=" + Δ + " WHERE "
for each filter f in ref do
  if f is the first filter in ref then
    sql = sql + f
  else
    sql = sql + "AND " + f
  end if
end for
```