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.

1
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.nciits.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 with the following abstract type $\tau$:

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

$\tau_T = [E_1 : \tau_1, \ldots, E_k : \tau_k, \{E_{k+1} : \tau_T\}, \ldots, \{E_{k+m} : \tau_T\}]$  \(k \geq 1, m \geq 0\) and $\tau_i \ (1 \leq i \leq k)$ is $\tau_S$ or $\tau_C$.

$\tau_C = [E_1 : \tau_1, \ldots, E_k : \tau_k] (k \geq 1)$ and $\tau_i \ (1 \leq i \leq k)$ is $\tau_S$ or $\tau_C$

(where “…” denotes a tuple and “[...]” denotes a set). $\tau_T$ denotes a tuple type, $\tau_C$ denotes a non-repeating complex type and $\tau_S$ denotes an atomic type (e.g. #PCDATA or CDATA). $E_0$ is an element name denoting the root of the document, and $E_i \ (i \geq 1)$ 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 \(\tau\) and \(E_{k+1},...E_{k+m}\) in the definition of \(\tau_T\)). 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 \(\tau_T\) all of whose descendants are non-repeating nodes \((\tau_S\) and/or \(\tau_C\)). For this reason, we will rename such \(\tau_T\) nodes to \(\tau_N\).

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 \(\tau_N\).

```json
result:
{author:
  {@id: CDATA,
   name: #PCDATA,
   address:
     [email: #PCDATA],
   {papers:
    {paper:
      {@id: CDATA,
       title: #PCDATA,
       confId: #PCDATA}],
    {book:
     [isbn: CDATA,
      title: #PCDATA}}
  }
}
```

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 \(\tau_N\(\text{book}\)\), and its type (DTD) is \[@isbn: CDATA, title: #PCDATA\].

**Proposition 2** The type of any well behaved view must contain at least one \(\tau_N\) node.

**Proof:** The proof of this proposition follows from the definition of well-behaved views (definition 1). Well-behaved views have abstract type \(\tau\), where \(\tau = E_0: \{E_1 : \tau_T\},...\{E_n : \tau_T\}, (n \geq 1)\). Since the condition \((n \geq 1)\) 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 \(\tau_T\). From definition 1, we have that \(\tau_T = \{E_1 : \tau_1, \ldots, E_k : \tau_k, \{E_{k+1} : \tau_T\}, \ldots, \{E_{k+m} : \tau_T\}\} (k \geq 1, m \geq 0)\) and \(\tau_1 (1 \leq i \leq k)\) is \(\tau_S\) or \(\tau_C\). Again, lets take the minimal scenario, where \((k = 1 \) and \(m = 0\)). We therefore have \(\tau_T = \{E_1 : \tau_1\}\), where \(\tau_1\) is \(\tau_S\) or \(\tau_C\). 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, \(\tau_N\) nodes are nodes of type \(\tau_T\) such that all of their descendants are non-repeating nodes \((\tau_S\) and/or \(\tau_C\)). 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 \(\tau_N\).

4
For more complicated views, it is obvious that there will be at least one node of type $\tau_N$. It turns out that a well-behaved view has to have at least one node of type $\tau_T$. The definition of type $\tau_T$ is recursive ($\tau_T$ can be composed of nodes $\tau_T$, $\tau_S$ and/or $\tau_C$). The recursion only ends when all children of a node with type $\tau_T$ are $\tau_S$ and/or $\tau_C$. Since this is the condition to a node $\tau_T$ to be renamed to $\tau_N$, this proves that every well-behaved view has at least one node $\tau_N$.

Proposition 3 There is at most one $\tau_N$ 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 $\tau_N$ along this path.

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

- Leaf nodes have type $\tau_S$. The parent of a leaf node can have types $\tau_C$, $\tau_T$ or $\tau_N$.
- Nodes of type $\tau_C$ can have parent of types $\tau_C$, $\tau_T$ or $\tau_N$.
- Nodes of type $\tau_N$ can only have parent of type $\tau_T$ or $\tau$, which is the type of the document root. The parent of a node of type $\tau_N$ can never have type $\tau_N$, since by definition, nodes of type $\tau_N$ are the ones whose children have type $\tau_S$ and/or $\tau_C$.
- Nodes of type $\tau_T$ can only have parent of type $\tau_T$ or $\tau$, 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.

$p = \tau_S, \tau_C, \ldots, \tau_C, \tau_T, \ldots, \tau_T, \tau$ (n $\geq 0$ and m $\geq 1$).

In this path, the only candidate of having type $\tau_N$ is the node whose type is $\tau_T$. This node will be renamed to $\tau_N$ if all of its children are of type $\tau_C$ and/or $\tau_S$. 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 $\tau_N$. Consequently, as we wanted to demonstrate, there is at most one node of type $\tau_N$ 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. 1. <conferencePapers>
2. {for $c in table("conference")
3. 4. <conference id="$c/confid/text()">}
5. 6. <![/$conf]
7. 8. $c/pub in table("paper")
9. 10. by $year in ($$/year)
11. 12. where $year/$confid=$c/confid
13. 14. return
15. 16. <papers year="$year/text()">}
17. 18. </conference>
19. 20. }
21. 22. </conferencePapers>
23. {for $c in table("conference")
24. 25. return
26. 27. {let $p := table("paper")
28. 29. where $year=distinct-values($$/year)
30. 31. <papers year="$year/text()">}
32. 33. return
34. 35. </papers>
36. 37. }
38. 39. };
40. 41. </conference>
42. 43. }
44. 45. </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:

```xml
<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" XVarName "in" TableExpr (",", XVarName "in" TableExpr)*
[33] ByClause ::= "by" XVarName "in" UniqueExpr (",", XVarName "in" UniqueExpr)*
[34] Header ::= "on" XName (XName "=" NestAttributeValue) |
| "on" XName ("(*)" XVarName "=" UniqueExpr) |
| "on" XName ("(*)" XVarName "=" UniqueExpr) "in" (UniqueExpression | "(" UniqueExpression "|
| UniqueExpression ")") "in" (UniqueExpression | 
| UniqueExpression )"

[35] UniqueExpression ::= 
| ("(*)" XVarName "=" UniqueExpression) |
| UniqueExpression "in" (UniqueExpression | UniqueExpression )"
[36] EltGroup ::= EltConstructor
[37] UnionExpr ::= "(" XVarName ")" XName ("(" union | "|")" XVarName ")" (XName = ")")
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 ::= ElmConstructor

The `ElmConstructor` production produces a node of type `τ`, which is the document root. By definition 1, `τ` = `E_0: \{E_1: \tau_1\}, \ldots, \{E_n: \tau_n\}, n \geq 1`. `E_0` is the name of the element constructed by `ElmConstructor`.

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

[3] ElmConstructor ::= "" QName AttList "/" | "" QName AttList? "/" ElmContent+ "" 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 `FVarExpr` or `Nest` productions.

Consequently, the only possible choice in production [3] at this point is "" QName AttList? "/" ElmContent+ "" QName S? "/" "". Here, `AttList` will be empty for the same reason explained above. The children of `E_0` are then determined by `ElmContent+`. Since this production is marked with "+", it is guaranteed that `E_0` will have at least one child, that is, the restriction (\(n \geq 1\)) 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_1\}, \ldots, \{E_n: \tau_n\}`, (\(n \geq 1\)). In this case, (\(1 \leq i \leq n\)).

The production that defines `ElmContent` is:

[4] ElmContent ::= ElmConstructor | 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 $\tau_T$. 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:


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: $\tau = E_0: \{E_1 : \tau_T\}, \ldots, \{E_n : \tau_T\}, (n \geq 1)$. Since each $E_i (1 \leq i \leq n)$ has type $\tau_T$, it is necessary to analyze the structure of this type and make sure that UXQuery produces it correctly.

According to definition 1, $\tau_T = [F_1 : \tau_1, \ldots, F_k : \tau_k, \{F_{k+1} : \tau_T\}, \ldots, \{F_{k+m} : \tau_T\}] (k \geq 1, m \geq 0)$ and $\tau_j (1 \leq j \leq k) = \tau_S$ or $\tau_C$.

As shown previously, node $E_i$ of type $\tau_T$ 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.


Attributes are atomic elements, so their type is $\tau_S$. 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 $\tau_S$. This conforms to the definition of $E_i$'s type ($\tau_T = [F_1 : \tau_1, \ldots, F_k : \tau_k, \{F_{k+1} : \tau_T\}, \ldots, \{F_{k+m} : \tau_T\}] (k \geq 1, m \geq 0)$ and $\tau_j (1 \leq j \leq k) = \tau_S$ or $\tau_C$). In this case, we ensure the restriction ($k \geq 1$), and $\tau_j$ is $\tau_S$. 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 $\tau_S$. However, since AttList is optional, we must analyze the structure produced by ElmtContent+. 

8
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 ≥ 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 ≥ 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, then F_j has type τ_S.

2(b) Header: Header is defined as:

[34] Header ::= "<" QName QName="NestAttValue">" ("{" 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 ≥ 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 ≥ 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.
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 represent 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:
Figure 5: Example of UXQuery that nests elements, and its corresponding auxiliary query tree

```sql
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.

```sql
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` 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` creates a new set of tuples, which should be mapped to distinct relational views. Information on levels above is considered to be common to
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 book AS b ON a.id = b.author) LEFT JOIN book AS b ON b.isbn = b.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 paper AS p ON a.id = p.author) LEFT JOIN paper AS p ON p.pid = p.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, Δ, ref>, where u is the type of operation (insert, delete, modify); Δ is the XML tree to be inserted, or (in case of a modification
an atomic value; and ref is a simple path expression in XPath [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).
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@uacm.com", 2000, "Books", 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 (6, "James Perez", "james@uacm.com", 2000, "Updating Relational Views", 999)
```

```
INSERT INTO VIEWPAPER (id, name, email, year, title, pid)
VALUES (6, "James Perez", "james@uacm.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
```

14
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 exist (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[0id="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 canceled.

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


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 ::= BeginConstructor
[3] BeginConstructor ::= <!Q Name Attlist? /> | <!Q Name Attlist? />
[5] Attr ::= $(Q Name $= $Q Name ? $)
[6] AttValue ::= "$" | "[Q Name Attlist?] $(Q Name Attlist?) $"
[7] AttValueContent ::= "$" | "$" | "$"
[8] PathExprAtt ::= $"$ QName $" / NodeTest
[9] QName ::= QName
[10] EnclosedExpr ::= "( PathExpr | PathExpr | Text )"
[12] DisExpr ::= AndExpr ( "or" AndExpr )
[13] AndExpr ::= Comparison Expr "and" Comparison Expr
[14] PathExpr ::= ![Q Name Name ? $ QName Name ? $ NodeTest]
[15] NodeTest ::= TextTest
[16] TextTest ::= "text" Text
[17] ForClause ::= ![Q Name Name ? $ QName Name ? $ TableExpr ( , , ) QName Name ? $ TableExpr]
[18] TableExpr ::= "table" ( "database" ? $ ) | "table" ( "database" QName Name ? $, )
[19] WhereClause ::= ![Q Name Name ? $ QName Name ? $] & $ QName Name ? $ & $ & $ & $ & $
[20] GeneralComp ::= ![Q Name Name ? $ QName Name ? $] & $ QName Name ? $ & $ QName Name ? $ & $
[21] OrderByClause ::= ![Q Name Name ? $ QName Name ? $ QName Name ? $]
[22] OrderSpec ::= ![Q Name Name ? $ QName Name ? $ QName Name ? $]
[23] OrderSpec ::= [Q Name Name ? $ QName Name ? $]
[24] OrderSpec ::= ![Q Name Name ? $ QName Name ? $ QName Name ? $]
[25] OrderSpec ::= ![Q Name Name ? $ QName Name ? $]
[26] OrderSpec ::= ![Q Name Name ? $ QName Name ? $]
[27] OrderSpec ::= ![Q Name Name ? $ QName Name ? $]
[28] OrderSpec ::= ![Q Name Name ? $ QName Name ? $]
[29] OrderSpec ::= ![Q Name Name ? $ QName Name ? $]
[30] OrderSpec ::= ![Q Name Name ? $ QName Name ? $]
[31] Where ::= ![Q Name Name ? $ QName Name ? $]
[32] Where ::= ![Q Name Name ? $ QName Name ? $]
[33] Where ::= ![Q Name Name ? $ QName Name ? $]
[34] Where ::= ![Q Name Name ? $ QName Name ? $]
[35] Where ::= ![Q Name Name ? $ QName Name ? $]
[36] Where ::= ![Q Name Name ? $ QName Name ? $]
[37] Where ::= ![Q Name Name ? $ QName Name ? $]
[38] Where ::= ![Q Name Name ? $ QName Name ? $]
[39] Where ::= ![Q Name Name ? $ QName Name ? $]
[40] Where ::= ![Q Name Name ? $ QName Name ? $]
[41] Where ::= ![Q Name Name ? $ QName Name ? $]
[42] Where ::= ![Q Name Name ? $ QName Name ? $]
[43] Where ::= ![Q Name Name ? $ QName Name ? $]
[44] Where ::= ![Q Name Name ? $ QName Name ? $]
[45] Where ::= ![Q Name Name ? $ QName Name ? $]
[46] Where ::= ![Q Name Name ? $ QName Name ? $]
[47] Where ::= ![Q Name Name ? $ QName Name ? $]
[48] Where ::= ![Q Name Name ? $ QName Name ? $]
[49] Where ::= ![Q Name Name ? $ QName Name ? $]
[50] Where ::= ![Q Name Name ? $ QName Name ? $]
[51] Where ::= ![Q Name Name ? $ QName Name ? $]
[52] Where ::= ![Q Name Name ? $ QName Name ? $]
[53] Where ::= ![Q Name Name ? $ QName Name ? $]
[54] Where ::= ![Q Name Name ? $ QName Name ? $]
[55] Where ::= ![Q Name Name ? $ QName Name ? $]
[56] Where ::= ![Q Name Name ? $ QName Name ? $]
[57] Where ::= ![Q Name Name ? $ QName Name ? $]

[18]
B Normalization process for the nest operator

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

```
[ nest Variable1 in TableExpr1, ..., Variablen in TableExprn.
  by NestVariable1 in (Variable1/QName11 | ... | Variablem/QName1m),
  ..., NestVariablen in (Variable1/QNamen1 | ... | Variables/QNamek) ]

where Expr

<ElName AttName1="{NestVariable1/text()}" ... AttNamek="{NestVariablek/text()}">
  <ElGroup1> ... <ElGroupm> </ElName>

==

let Variable1 := TableExpr1, ..., Varialec := TableExprm
for NestVariable1 in distinct-values(Variable1/QName11 | ... | Variable1m/QName1m),
  ..., NestVariablen in distinct-values(Variable1/QName11 | ... | Variables/QNamek)
return

<ElName AttName1="{NestVariable1/text()}" ... AttNamek="{NestVariablek/text()}">
  {for fs:SubVariable(1)}
  where fs:SubExpr(1) and (Variable1 = NestVariable1 and ... and Variablek = NestVariablek)
  return ElGroup1
  {for fs:SubVariable(m)}
  where fs:SubExpr(m) and (Variable1 = NestVariable1 and ... and Variablek = NestVariablek)
  return ElGroupm
</ElName>
```

This normalization process supposes that:

- \{Variable1, ..., Variablem, ..., Variablek1, ..., Variablekn\} \subseteq \{Variable1, ..., Variablen\}
- The auxiliary function `fs:SubVariable(i)` returns all variables Vx referenced in ElGroupi and also all variables Vy appearing in a condition of the form "Vx/QName1x cmp Vy/QName1y" in Expr in the where clause of the nest operator; cmp ∈ \{"=", ", <", ", >", ", ! =", ", ", <=", ", >="\}.
- 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 = "'0' + 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 cl is element then
    n.value = cl
  else
    n.value = cl. "/@text()"
  end if
end if
else
Let A denote all variable bindings below cl 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 below cl and above the next non-leaf element, ignoring conditions inside nests
for each w in W do
  annotate n(w)
end for
if cl is a header of a nest then
  Let N denote the nest expression of which cl 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
  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 sub-trees.

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

map(aux)

Let s be the number of sub-trees 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(n1) = clone(n)
      x.parent = n1
      delete-annotation(n1, x)
      r1 = n.parent
      while r1 <> root(aux) do
        Create node(c) = clone(r1)
        {clone r1’s children}
        let Y be the set of direct children of the node r1 (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 n1 with c}
      end if
      r1 = n.parent then

20
\[
n_i, \text{parent} \rightarrow c
\]
end if
\[
r_i = r_i, \text{parent}
\]
end while
Create node(c') = clone(r_i)
c.parent = c'
r_i = c

{Deal with nest attributes}

Let \text{Next} be the set of all leaf children of \text{n}
for each \text{nest} in \text{Next} do
Create node(z) = clone(nest)
z.parent = n_i
delete-value(z, n_i)
end for
end for
end if
if \ i = 0 then
create-relational-view(aux)
else
for j from 1 to i do
create-relational-view(r_j)
end for
end if

The algorithm map uses the sub-routines \text{delete-annotation}(n,x) and \text{delete-value}(n,x).

\text{delete-annotation}(n,x)

{deletes all the annotations in n that are not related with x}

Let \text{W} be the set of annotations of where clauses in \text{n}
Let \text{C} be the set of children of \text{n}
Let \text{V} be the variables referenced in the value of the nodes in \text{C}
Let \text{V'} = \text{V}
for each \text{w} in \text{W} do
if \text{w} has the form \text{V}_{\text{a}}/\text{QName}_{\text{w}}, \text{cmp} \text{V}_{\text{b}}/\text{QName}_{\text{w}}_
where \text{V}_{\text{a}}, \text{V}_{\text{b}} are variables, \text{V}_{\text{a}} \in \text{V},
\text{cmp} \in \{"=", "<", "",">", "<="", ",">="\}\ then
\text{V'} = \text{V'} + \text{V}
end if
end for
for each \text{w} that does reference a variable in \text{V'} do
\text{delete-where-annotation}(\text{w}, \text{W})
end for

The text of \text{W} be the set of where clauses in \text{n} (this set was modified by the previous loop)
for each \text{a} in \text{A} do
if \text{a} does not reference a variable in \text{W'} then
\text{delete}(\text{a})
end if
end for

The algorithm \text{delete-annotation} uses the algorithm \text{delete-where-annotation}.

\text{delete-where-annotation}(\text{w}, \text{W})

if \text{size}(\text{W}) > 1 then
if \text{w} is connected in \text{W} by an expression \text{e} of the form \text{e}_1 and \text{w} then
replace \text{e} by \text{e}_1
else
if \text{w} is connected in \text{W} by an expression \text{e} of the form \text{w} and \text{e}_1 then
replace \text{e} by \text{e}_1
else
if \text{w} is connected in \text{W} by an expression \text{e} of the form \text{w} or \text{e}_1 then
replace \text{e} by \text{e}_1
else
if \text{w} is connected in \text{W} by an expression \text{e} of the form \text{e}_1 or \text{w} then
replace \text{e} by \text{e}_1
end if
end if
end if

21
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 vn/QName
Let A be the set of variable bindings in x
for i from 1 to k do
  if vi \notin 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 next normalization process)

The algorithm generate-relational-view generates an SQL statement corresponding to a relational view. The parameter t is a sub-tree produced by 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 + " LEFT JOIN "
  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
      end if
      J = J + joinCond
    end while
  end if
end if

22
mark(v)
J = J + joinCond
end if
end while
nl = c
\[ c = \text{non-leaf child of } nl \]
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 unselected 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
\[ where = "WHERE " \]
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.

\[ \text{findJoinTable}(a, W, VC, joinCond) \]

\[ \text{joinCond} = "" \]
for each v in VC do
if there is a w in W of the form V_n/QName_n cmp V_v/QName_v,
(\{V_n, V_v are variables, \text{V}_n \in a, \text{V}_v \in v\} cmp \in \{"="", "<", ",", "\text{X}="", "\text{X}>=""\})
then
for each w in W that involves variables a and v do
joinCond = joinCond + w + ""
end for
end if
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 \text{ and } w_2"
replace; by and
else
if w_1 and w_2 are connected in W by an expression of the form "w_1 \text{ or } w_2"
replace; by or
else
if w_1 and w_2 are connected in W by an expression of the form "w_1 \text{ and } w_2"
replace; by or

\[ 23 \]
else
    if \( w_1 \) and \( w_2 \) are connected in \( W \) by an expression of the form \( "w_1 \text{ and } c \text{ or } w_2" \) then
        replace \( ; \) by \( \text{and} \)
    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 \textit{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
- \( \Delta \): the subtree to be inserted

The procedure supposes that the update specification was already checked for schema conformance. The subroutine \textit{type(n)} returns the type of a given node (\( \tau_N, \tau_T, \tau_C \text{ or } \tau_R \)). The subroutine \textit{view(n)} returns the name of the relational view associated with node \( n \), assuming that \( n \) is of type \( \tau_N \).

\textbf{translateInsert}(V, aux, ref, \Delta)

Insert \( \Delta \) in \( V \) using \( ref \) as insertion point. \( \Delta \) 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 \( \text{type}(m) = \tau_N \) then
        \textit{generateInsertSQL}(view(m), root(\( \Delta \)), \( n \), \( V \))
    else
        Let \( X \) be the set of nodes of type \( \tau_N \) in \( \Delta \)
        for each \( x \) in \( X \) do
            \textit{generateInsertSQL}(view(n), \( x \), \( n \), \( V \))
        end for
    end if
end for

The algorithm \textit{translateInsert} calls the subroutine \textit{generateInsertSQL}, which generates a single SQL insert statement. The parameters are:

- \textit{RelView}: the relational view where the insertion is to take place
- \( r \): the root of the subtree being inserted
- \textit{InsertionPoint}: the node in \( V \) under which the subtree is being inserted
- \( V \): the XML view

\textbf{generateInsertSQL}(RelView, \( r \), InsertionPoint, \( V \))

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

24
Let the node found be \( m \)

\[
sql = sql + getValue(m)
\]

end if

if \( i < \) getNumberOfAttributes[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 \( (\tau_N, \tau_T, \tau_C \text{ or } \tau_S) \). The subroutine \( view(n) \) returns the name of the relational view associated with node \( n \), assuming that \( n \) is of type \( \tau_N \).

\[
\text{translateDelete}(V, aux, ref)
\]

Let \( p \) be the unqualified portion of \( ref \)

Let \( m \) be the node resulting from the evaluation of \( p \) against \( aux \)

if \( type(m) = \tau_N \) then

\[
generateDeleteSQL(view(m), ref)
\]

else

Let \( X \) be the set of nodes of type \( \tau_N \) 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

25
• $\Delta$: the new value

The procedure supposes that the update specification was already checked for schema conformance. The subroutine \texttt{type(n)} returns the type of a given node ($\tau_N$, $\tau_T$, $\tau_C$, or $\tau_S$). The subroutine \texttt{view(n)} returns the name of the relational view associated with node \texttt{n}, assuming that \texttt{n} is of type $\tau_N$.

\texttt{translateModify(V, aux, ref, $\Delta$)}

Let $p$ be the unqualified portion of \texttt{ref}
Let $m$ be the node resulting from the evaluation of $p$ against \texttt{aux}
Let $r$ be the ancestor of $m$ whose type is $\tau_T$ or $\tau_N$
Let \texttt{N} be the set of nodes resulting from the evaluation of $\texttt{ref}$ in \texttt{V}
for each \texttt{n} in \texttt{N} do
  if \texttt{type(r)} = $\tau_N$ then
    \texttt{generateModifySQL(view(r), $\Delta$, ref)}
  else
    Let $X$ be the set of nodes of type $\tau_N$ under $r$
    for each \texttt{x} in \texttt{X} do
      \texttt{generateModifySQL(view(x), $\Delta$, ref)}
    end for
  end if
end for

The algorithm \texttt{translateModify} calls the subroutine \texttt{generateModifySQL}, which generates a single SQL update statement. The parameters are:

• \texttt{RelView}: the relational view where the insertion is to take place
• \texttt{$\Delta$}: the new value
• \texttt{ref}: the update path

\texttt{generateDeleteSQL(RelView, $\Delta$, ref)}

\texttt{sql} = "UPDATE " + \texttt{RelView} + " SET "
Let $t$ be the terminal node in \texttt{ref}
\texttt{sql} = \texttt{sql} + $t$ + "$=\Delta$" $\texttt{WHERE}$
for each filter $f$ in \texttt{ref} do
  if $f$ is the first filter in \texttt{ref} then
    \texttt{sql} = \texttt{sql} + $f$
  else
    \texttt{sql} = \texttt{sql} + "$\texttt{AND}$ $f$
  end if
end for