September 2006

PATAXÓ: A Framework to Allow Updates Through XML Views

Vanessa P. Braganholo
Universidade Federal do Rio de Janeiro

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

Carlos A. Heuser
Universidade Federal do Rio Grande do Sul

Postprint version. The definitive version was published in *ACM Transactions on Database Systems (TODS)*, Volume 31, Issue 3, September 2006, Pages: 839 - 886.
Publisher URL: http://doi.acm.org/10.1145/1166074.1166078

PATAXÓ: A Framework to Allow Updates Through XML Views

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 a lot of work has been done on querying relational databases through XML views, the problem of updating relational databases through XML views has not received much attention. In this work, we map XML views expressed using a subset of XQuery to a corresponding set of relational views. Thus, we transform the problem of updating relational databases through XML views into a classical problem of updating relational databases through relational views. We then show how updates on the XML view are mapped to updates on the corresponding relational views. Existing work on updating relational views can then be leveraged to determine whether or not the relational views are updatable with respect to the relational updates, and if so, to translate the updates to the underlying relational database.

Keywords
Relational databases, updates, XML views

PATAXÓ: a framework to allow updates through XML views

VANESSA P. BRAGANHOLO
COPPE, Universidade Federal do Rio de Janeiro, Brazil

SUSAN B. DAVIDSON
CIS, University of Pennsylvania

CARLOS A. HEUSER
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil

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 a lot of work has been done on querying relational databases through XML views, the problem of updating relational databases through XML views has not received much attention. In this work, we map XML views expressed using a subset of XQuery to a corresponding set of relational views. Thus, we transform the problem of updating relational databases through XML views into a classical problem of updating relational databases through relational views. We then show how updates on the XML view are mapped to updates on the corresponding relational views. Existing work on updating relational views can then be leveraged to determine whether or not the relational views are updatable with respect to the relational updates, and if so, to translate the updates to the underlying relational database.

1 Introduction

XML is frequently used as an interface to relational databases. In this scenario, XML documents (or views) are exported from relational databases and published, exchanged, or used as the internal representation in user applications. This fact has stimulated much research in exporting and querying relational data as XML views [29, 44, 43, 18]. However, the problem of updating a relational database through an XML view has not received as much attention: Given an update on an XML view of a relational database, how should it be translated to updates on the relational database?

There are many applications in which the need to update through an XML view of a relational database arises, from manufacturing to finance to general administration. As an example, Supplier Relationship Management (SRM) systems are used in companies which purchase large amounts of supplies from a variety of suppliers. Due to the size of the purchases, each supplier works almost exclusively with the company. The company can therefore specify how orders are to be managed; in particular, it typically requires suppliers to make their stock available in a specific XML format conformant with their SRM so that purchases can be planned in an automated manner. The XML file is a view of the supplier's underlying (relational) product management database. The company then notifies each supplier of the quantity of each product it intends to purchase next so that the supplier can begin production and be ready for immediate delivery when the order is placed. Currently, the notification and ordering is done outside of the supplier's XML stock view. However, discussions with developers in at least one such company have revealed the desire to provide notification and ordering by updating the stock XML view directly, since this would considerably simplify the current transaction process.

The approach we take to solving this problem is to take advantage of the well studied problem of updating through relational views, and present a solution by mapping from XML views to relational

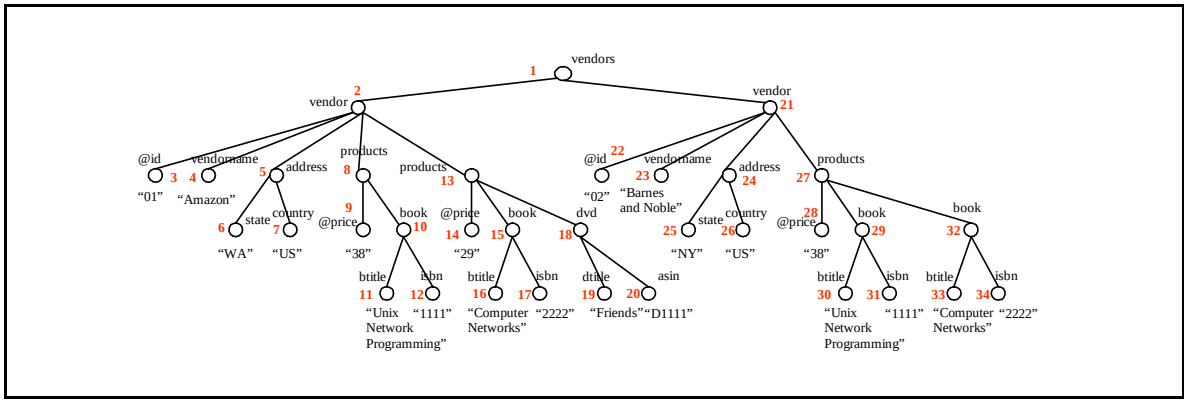


Figure 1: XML view containing vendors, books and dvds

views. Specifically, we (i) map an XML view query to a set of relational view queries; (ii) map updates over the XML view to updates over the corresponding relational views; and (iii) use existing work on updating through relational views to map the updates to the underlying relational database. The starting point of this mapping is a formalism that we call *query trees*. In the relational case, work on updating through views has focused on select-project-join views since they represent a common form of view that can be easily reasoned about using key and foreign key information. Similarly, query trees represent a common form of XML views that allow nesting, composed attributes, heterogeneous sets and repeated elements. However, query trees were conceived as an internal query representation and are not well-suited for end-users. To specify how an XML view is constructed from a relational source, we therefore use the UXQuery [11] view definition language. UXQuery is expressive enough to capture the XML views that we have encountered in practice yet is simple to understand and manipulate. It is a subset of XQuery [5], and equivalent in expressive power to DB2 DAD files [19]. Throughout the paper, we will use the term "XML view" to mean those produced by UXQuery.

In summary, in this paper we will focus on the interactions between UXQuery and query trees, as well as the architecture and algorithms of the system that implements them, PATAXÓ[1].

1.1 Running Example and Overview

An example of an XML view is shown in Figure 1. In this example, and in every example of this paper, we use the database shown in Figure 2. Its schema consists of six tables: *Vendors*, *Warehouse*, *Book*, *DVD*, *SellBook* and *SellDVD*. Table *SellBook* establishes a relationship between tables *Vendor* and *Book*, registering prices of books sold by a given vendor. The table *SellDVD* plays the same role for dvds and vendors. A vendor has several warehouses in which products are stored. In the XML view of Figure 1, data was extracted from tables *Vendor*, *Book*, *DVD*, *SellBook* and *SellDVD*. Note that the products for a vendor are grouped by price.

The strategy we adopt is to map an XML view query to a set of underlying relational view queries. Similarly, we map an update against the XML view to a set of updates against the underlying relational views. It is then possible to use any existing technique on updating through relational views to both translate the updates to the underlying relational database and to answer the question of whether or not the XML view is updatable with respect to the update.

In preliminary work [10], we used the nested relational algebra (NRA) as the view definition language. In this approach, each XML view query is mapped to a *single* relational view query. However, NRA views are not capable of handling heterogeneous sets, which arise frequently in practice. Thus,

[1]PATAXÓ is the name of a native Brazilian tribe (there are still a few living in Bahia) and stands for "Permitindo ATualizações Através de visões Xml em bancos de dados relaciOnais", which is loosely translated as permitting updates on relational databases through XML views.

Vendor

vendorId	vendorName	url	state	country
01	Amazon	www.amazon.com	WA	USA
02	Barnes and Noble	www.barnesandnoble.com	NY	USA

SellBook

vendorId	isbn	price
01	1111	38
01	2222	29
02	1111	38
02	2222	38

Warehouse

wId	vendorId	address	city	state	country
D1	01	1245, Bourbom Street	Seatle	WA	USA
D2	02	1478, 25th Avenue	New York	NY	USA
D3	01	4545, 15th Avenue	Seatle	WA	USA

Book

isbn	title	publisher	year
1111	Unix Network Programming	Prentice Hall	1998
2222	Computer Networks	Prentice Hall	1996

Dvd

asin	title	genre	nrDisks
D1111	Friends	Comedy	4

SellDvd

vendorId	asin	price
01	D1111	29

Constraints:

On table Vendor:
- primary key(vendorId)

On table Warehouse
- primary key(wId)
- foreign key(vendorId) references Vendor

On table Book
- primary key(isbn)

On table Dvd
- primary key(asin)

On table SellBook
- primary key(vendorId, isbn)
- foreign key(vendorId) references Vendor
- foreign key(isbn) references Book

On table SellDvd
- primary key(vendorId, asin)
- foreign key(vendorId) references Vendor
- foreign key(asin) references Dvd

Figure 2: Sample Database

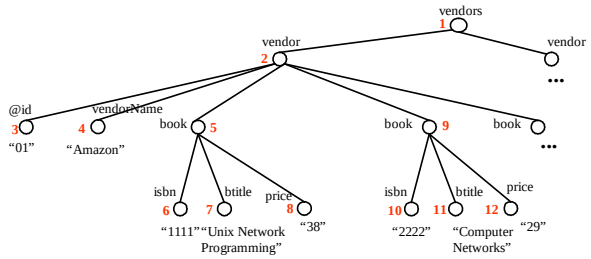


Figure 3: XML view showing books and vendors

the NRA is capable of representing the XML view of Figure 3 but not that of Figure 1. To make this clearer, in this context *heterogeneity* means distinct DTD types for repeating children. In the example of Figure 1, the node products has heterogeneous children – that is, it has repeating children of types *book* and *dvd*. In contrast, in Figure 3 there are no heterogeneous nodes.

Since views such as the one in Figure 1 are very common in practice, we have decided to adopt a more general view definition language – UXQuery. We then map a query in UXQuery to an *extended* query tree[2], and use the results of [12] to map the resulting XML view to a set of relational views. The extension to query trees presented here allows the grouping of tuples that agree on a given value. For example, the *products* in Figure 1 represents grouping by price.

As mentioned before, a single XML view produced by a query tree can be mapped to a *set* of relational views. The reason why there may be more than one underlying relational view for an XML view expressed by a query tree is the presence of heterogeneous sets. For example, the XML view of Figure 1 is mapped to two corresponding relational views: one for vendors and books (*ViewBook*), and another one for vendors and DVDs (*ViewDVD*). We must then identify to which relational views an XML update should be mapped.

[2]This paper extends the query trees of [12] to support grouping of tuples.

As a concrete example, suppose we wish to insert a new book

```
<book>
  <btitle>Birding in North America</btitle>
  <isbn>5555</isbn>
</book>
```

at the point in the XML view of Figure 1 specified by the following update path expression: `/vendors/vendor[@id="01"]/products[@price="29"]` (node 13). Using the techniques of this paper, this update would be mapped to the following SQL insert statement over *ViewBook*:

```
INSERT INTO VIEWBOOK (id, vendorname, state, country, price, isbn, btitle)
VALUES ("01", "Amazon", "WA", "US", 29, "5555", "Birding in North America");
```

Using existing relational techniques (in particular, that of [24]), we would then map the update on the relational view to a set of updates against the underlying relations.

We also address the problem of checking if an update respects the schema of the XML view. As an example, the DTD of the XML view of Figure 3 is shown below:[3]

```
<!ELEMENT vendors (vendor*)>                        <!ELEMENT vendorname (#PCDATA)>
<!ELEMENT vendor (book*)>                           <!ELEMENT isbn (#PCDATA)>
<!ATTLIST vendor id CDATA #REQUIRED>                <!ELEMENT title (#PCDATA)>
<!ELEMENT book (vendorname, isbn, title, price)>    <!ELEMENT price (#PCDATA)>
```

If a user tries to insert the XML tree `<bike>Giant</bike>` using the update path `/vendors/vendor[@id="01"]`, we would determine that the update is not correct with respect to the schema of the XML view, and would not attempt to map it to the underlying relational database.

1.2 Contributions

In summary, the main contributions of this paper are:

- A notion of query trees which captures a common form of XML views that allows nesting, composed attributes, heterogeneous sets and repeated elements.
- A subset of XQuery, to define XML views over relational databases.
- An architecture of a system that implements the ideas of this paper.
- Algorithms to map an XML view query to a set of corresponding relational views queries, and to map updates over the XML view to updates over the relational views.

The chief contribution of this paper is a complete and novel treatment of updating relational databases through XML views, a problem that has not been addressed in previous literature.

1.3 Organization of the text

The outline of this paper is as follows: Section 2 presents related work. Section 3 shows the architecture of the PATAXÓ System and discusses its main modules. The view definition language (UXQuery) and query trees are presented in Section 4. Section 5 presents a simple language for updating XML views and shows how to detect whether or not an update is correct with respect to the XML view DTD. An algorithm for mapping an XML view to a set of underlying relational views is given in Section 6, along with an algorithm for mapping insertions, modifications and deletions on XML views to updates on the underlying relational views. Section 7 presents an evaluation of our view definition algorithm. We conclude in Section 8 with a summary of the paper's main contributions and a discussion of future work.

[3] We use DTDs because their syntax is succinct. Our implementation actually uses XSD.

2 Related Work

The closest work on updates through XML views is that of [51, 52]. In [51], the XML views considered are XML documents stored in relational databases and reconstructed using XQuery. For this class of views, it is proven that it is always possible to correctly translate the updates back to the database. However, they do not give details of how such translations are made. This approach differs from ours since we deal with XML views constructed over legacy databases. The approach in [52] presents an extension to the notion of clean source of Dayal and Bernstein [24]. They use this extended notion to study the updatability of XQuery views published over relational databases. Their results are analogous to the results of our previous work [10]. Neither of these papers discuss how updates are translated to the underlying relational database.

Work on data provenance (or data lineage) is also relevant, since one of the concerns is knowing where a piece of data that is in a given view came from. In the literature, this is called *where provenance* [15]. The solution adopted in [15] is to syntactically analyze the view definition query. *Why provenance*, on the other hand, deals with knowing *why* a piece of data is in the view, i.e. what tuples were used to generate a given piece of information in the view [23]. In our work, we also need to know where data came from, so we can map updates over this data to the database. As in [15], our approach uses view definition analysis to solve this problem.

Another area related to our work is data mapping, which is studied in data integration. The main goal in this area is to integrate autonomous data sources so they can be viewed as an integrated repository where queries can be posed. Most of the work in this area [30, 16, 4, 41, 36] use a mediated architecture that defines a global view of the data sources and maps each local source to the global view. While the XML files we consider are views of a (single) relational database, we are primarily concerned with updating through the view rather than the simpler problem of querying the view. Furthermore, systems that allow updates over the global view [16, 36] do not specify how to map an XML update to the underlying systems.

2.1 Building XML views over Relational Databases

Several papers explore the subject of building and querying XML views over relational databases [29, 43, 6, 18, 44]. Most 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 [29, 43, 6, 18].

Each of the existing approaches uses a different technique to construct the XML view using a relational engine to retrieve data. Some transform the XML view definition into extended SQL [44, 43, 18]; others use internal representations to map the XML view to several SQL queries [29, 6]. In our approach, views are built using an extension of XQuery, but the goal is to update the resulting views rather than to query them.

Other approaches focus on extracting XML documents from relational databases; querying the resulting document is not the goal. Some of these approaches apply a default mapping over an SQL query specified by the user [49]. Others allow the user to specify an XML template, together with SQL instructions that will be used to generate the resulting XML document [31].

Commercial relational databases also offer support for extracting XML views over relational databases. IBM DB2 XML Extender [19] 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 [28]. SQL Server extends SQL with a directive called `FOR XML` [22]. It also provides an alternative way of expressing XML views, which can be done using an annotated XML Schema. The schema reflects the view structure that the user wants to construct, and is augmented by annotations that tell where the element must come from (i.e. database table and column name). The XML instance is not generated from the schema. The user must specify an XPath query over the view in order to get the instance (or portions of it).

As we can see, most commercial databases have their own way of dealing with XML, which makes it difficult to use them for accessing and updating legacy databases. DB2, which allows the creation

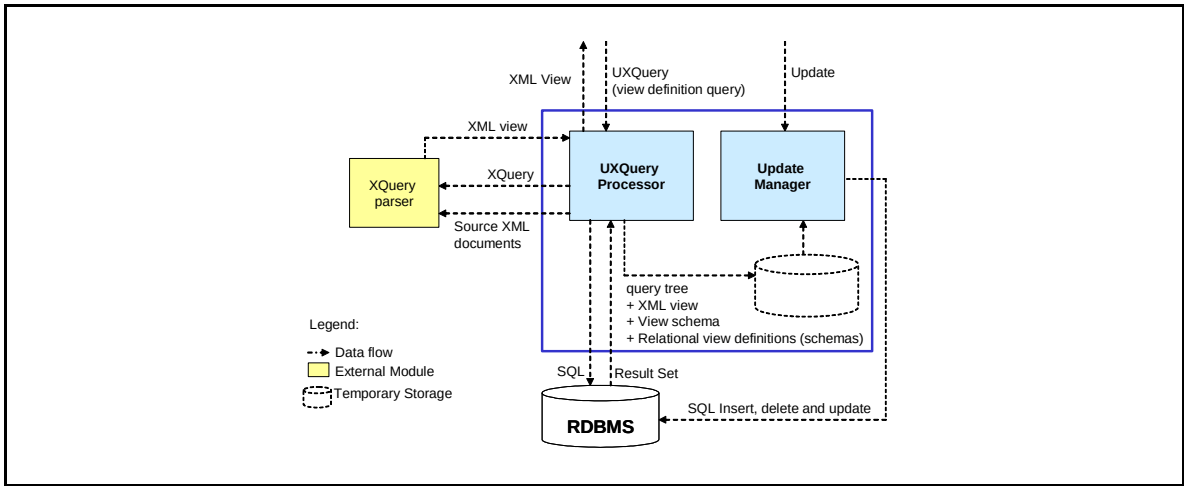


Figure 4: PATAXÓ System architecture

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 [21]. 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.

Native XML databases like XIndice [2], Timber [32] and Tamino [45, 42] also support updates. However, the goal of all these systems differs from ours since they do not update through views, nor is the source data relational.

3 Architecture

In our approach, the user is presented with an XML document which is an UXQuery view of an underlying relational database. The user performs updates directly on this document; the system then determines how to translate the updates to the underlying relational database so that the updated view remains correct. Thus there is no need for the XML document to be regenerated by re-executing the view query over the relational database.

PATAXÓ implements an UXQuery processor and maps each UXQuery view definition to a set of SQL view definitions over the underlying relational database. To do this, an UXQuery view definition query is transformed into an internal representation called a *query tree* [12], which is then manipulated by our system and mapped to a set of corresponding relational view definitions. When an update is performed on the UXQuery view document, it is mapped to a set of updates over the relational views.

The overall architecture of PATAXÓ is shown in Figure 4, and consists of two main modules: the *UXQuery Processor* and the *Update Manager*. The *UXQuery Processor* is responsible for processing the UXQuery view definition and generating the XML view instance (document). The *Update Manager* receives updates over this document from users and maps them to updates over the corresponding relational views. A sub-module called *Relational View Updater* then translates the relational view updates to the underlying relational database. We do not cover details of this module in this paper.

We now present each of these modules in detail.

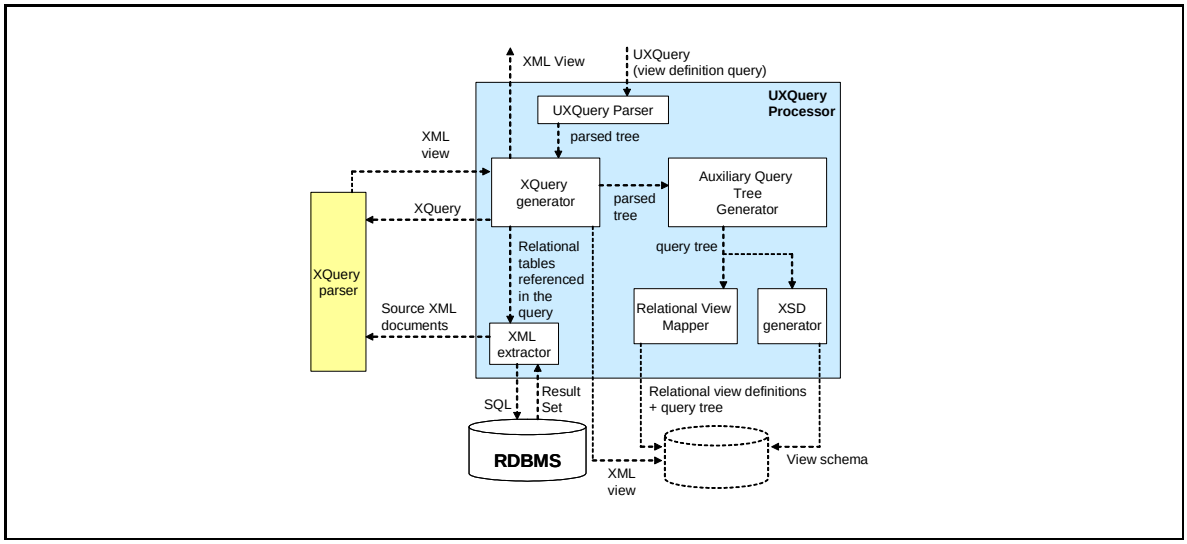


Figure 5: UXQuery Processor

3.1 UXQuery Processor

The UXQuery processor (Figure 5) is the module responsible for processing a view definition query expressed in UXQuery and producing the corresponding XML view instance. To do this, it translates a query in UXQuery to a query using pure XQuery syntax (see Section 4.1.1 for details on this translation).

From the parsed UXQuery query, the UXQuery Processor generates the XQuery query (which is executed by an external XQuery processor), the query tree and the XML schema of the XML view. (See Section 4.1.1 for the transformation rules for translating UXQuery queries into XQuery, and Section 4.3 for the translation of UXQuery queries to query trees.)

The generated query tree is used by the *Relational View Mapper* to generate the relational view queries that correspond to the XML view query (Section 6.1). Notice that we do not create the relational views in the underlying relational database. We just store the SQL view definition queries in PATAXÓ and use them in the Relational View Updater Module. The query tree is also used by the *XSD Generator* to generate the schema of the XML view (Section 4.2.4).

Relevant portions of the relational source data are translated to XML by a submodule called *XML Extractor*. The XML Extractor encodes a relational table in XML using an element `row` as a tuple delimiter. For example, the *Vendor* table of Figure 2 is represented in XML as:

```
<vendor>
  <row>
    <vendorid>01</vendorid>
    <vendorname>Amazon</vendorname>
    <url>www.amazon.com</url>
    <state>WA</state>
    <country>USA</country>
  </row>
  <row>
    ...
</vendor>
```

Since SQL does not distinguish between lower- and uppercase, we have adopted the convention that element names in the extracted XML documents are all in lowercase (notice that our example of Figure 2 uses mixed case). In this way, XML elements representing tables/attributes in UXQuery queries must also be referenced in lowercase.

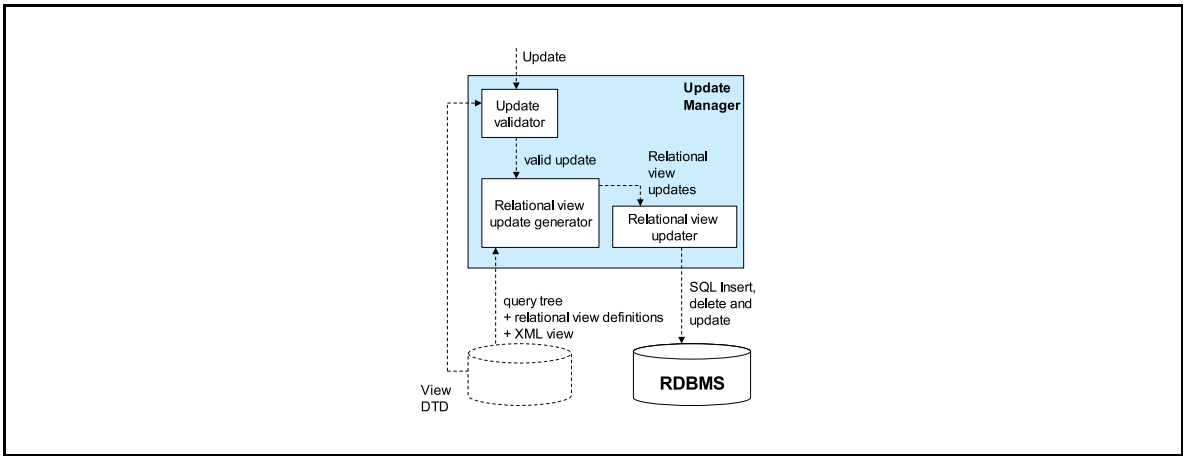


Figure 6: Update Manager

Rather than extracting the entire table, the XML Extractor uses selection conditions specified in the UXQuery (`where` conditions) to eliminate unnecessary tuples, and projects only the columns specified in the query. In this way, we avoid extracting data that would be discarded by the XQuery processor when processing the query. Notice that the extracted XML documents are used as input to process the UXQuery query.

After extracting the XML files that represent relevant portions of the underlying relational tables (*XML Extractor*) and producing the XQuery query (*XQuery Generator*), an external XQuery processor (Saxon [33]) is used to process the query. The result of this processing is the XML view document over which updates are then made by users.

3.2 Update Manager

The Update Manager (Figure 6) is the module responsible for receiving update requests and mapping the updates to the underlying relational database. In order to do so, it first checks whether or not the update conforms to the view schema and rejects updates that do not conform (Section 5.1).

Using the query tree, the *Relational View Update Generator* takes the requested update and translates it to updates over the corresponding relational views (as specified in Section 6.2). The *Relational View Updater* then uses existing techniques for mapping updates over the relational views to updates over the base tables. Our current version of PATAXÓ uses the translation algorithm of [24] to produce updates over the base tables; updates that cause side-effects are rejected (for details please see [12]). However, this module could be replaced by many other options [35, 3, 38], many of which allow side-effects that would then have to be propagated back to the view. Details are beyond the scope of this paper.

4 View Definition Language

4.1 UXQuery

Due to its wide-spread acceptance, we would ideally adopt XQuery [5] as the view definition language. However, it contains a number of features that are not consistent with the underlying relational nature of the data, as well as others that create new values that do not appear in the underlying relational database. In particular, order related operators affect only the layout of the resulting XML view document rather than the contents of the underlying relational database, in which tuples are inherently unordered. Thus while ordering is allowed in UXQuery, it is not considered in the translation of updates

to the relational database. Furthermore, aggregate operators create ambiguity when mapping a given view tuple to the underlying relational database. We therefore outlaw aggregate operators. This means that the use of `let` in our subset of XQuery must be very carefully controlled, and for this reason we will allow it only as expanded by a new macro called `xnest`. To evaluate the effect of these restrictions, we have analyzed the use of `let` in the queries of the XQuery Use Cases (Relational) [17], and have concluded that the only places `let` cannot be replaced by a `for` is when aggregate operations or function applications are used. We therefore feel that these restrictions are reasonable and do not overly limit the expressiveness of our language.

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

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

It is important to notice that UXQuery is a *view definition* language rather than an *update* language. Our update language will be introduced in Section 5.

In this section we assume the reader is familiar with XQuery, its syntax and semantics, and will not get into details that UXQuery "inherits" from XQuery. For further details on XQuery, please refer to [5]. The XQuery use cases are also an easy way to understand XQuery [17].

As a first example, we show a very simple UXQuery view definition query which retrieves vendors and their warehouses. The query is shown in Figure 7 (left hand side). The only difference between this query and one in XQuery is the `table` input function (lines 2 and 6), which takes as input the name of the relational table and produces a set of tuples. The XML view resulting from this query is also shown in Figure 7 (right hand side).

The EBNF of UXQuery is shown in Figure 8. It is based on the EBNF of the XQuery Core [5], and has been simplified to remove operators not allowed by UXQuery. We have also added the `xnest` operator and the `table` input function. In the EBNF we use a set of grammar definitions available in the XML documentation. The basic tokens `Letter` and `Digit` are defined in [14]. The identifier `QName` is defined in [13]. Literals and numbers are defined in [5].

The formal semantics of UXQuery matches the semantics of XQuery [27] with the exception of the new input function `table` and the macro `xnest`, which we discuss next.

Semantics of `table()`. XQuery has two input functions: `collection` and `doc` [39]. 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 of the following form:

```
<row> <!-- tuple attributes -->
   <attribute-1> value of attribute 1</attribute-1>
   ...
   <attribute-n> value of attribute n</attribute-n>
</row>
<row>
   ...
</row>
...
```

Following SQLX [28], we translate this input function to pure XQuery as follows.

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

```
 1. <vendors>
 2.  {for $v in table('Vendor')
 3.   return
 4.   <vendor id='{$v/vendorid/text()}'>
 5.    {$v/vendorname}
 6.    {for $w in table('Warehouse')
 7.     where $v/vendorid=$w/vendorid
 8.     return
 9.     <warehouse>
10.       <idWarehouse>{$w/wid/text()}</idWarehouse>
11.       <address>
12.         <street>{$w/address/text()}</street>
13.         {$w/city}
14.         {$w/state}
15.         {$w/country}
16.       </address>
17.     </warehouse>
18.    }
19.   </vendor>
20.  }
21. </vendors>
```

```
<vendors>
  <vendor id="01">
    <vendorname>Amazon</vendorname>
    <warehouse>
      <idWarehouse>D1</idWarehouse>
      <address>
        <street>1245, Bourbom Street</street>
        <city>Seatle</city>
        <state>WA</state>
        <country>USA</country>
      </address>
    </warehouse>
    <warehouse>
      <idWarehouse>D3</idWarehouse>
      <address>
        <street>4545, 15th Avenue</street>
        <city>Seatle</city>
        <state>WA</state>
        <country>USA</country>
      </address>
    </warehouse>
  </vendor>
  <vendor id="02">
    <vendorname>Barnes and Noble</vendorname>
    <warehouse>
      <idWarehouse>D2</idWarehouse>
      <address>
        <street>1478, 25th Avenue</street>
        <city>New York</city>
        <state>NY</state>
        <country>USA</country>
      </address>
    </warehouse>
  </vendor>
</vendors>
```

Figure 7: Example of a simple query that retrieves vendors and warehouses and its result

For this input function to work, the relational table used as the parameter in the function call must be represented in XML. As an example, the function call shown in line 2 of Figure 7 assumes that table *Vendor* is available in a file named `vendor.xml` which has the following structure:

```
<vendor>
  <row>
    <vendorid>01</vendorid>
    <vendorname>Amazon</vendorname>
    <url>www.amazon.com</url>
    <state>WA</state>
    <country>USA</country>
  </row>
  <row>
    ...
</vendor>
```

The extraction of relational tables to XML is done by the *XML Extractor* in PATAXÓ (see Section 3.1).

Semantics of `xnest`. The `xnest` operator is used to specify a possibly heterogeneous set of nested tuples that agree on the value of one or more attributes. The tuples are grouped according to the value of these attributes, which we call *nesting attributes*. A simple (non-heterogeneous) example of such a query is shown in Figure 9 (lines 1-23). The query specifies a join of tables *Vendor*, *Book* and *SellBook*. For each vendor, it shows the vendor name, the vendor Id, and the books sold by that vendor grouped by price. The `xnest` operator is shown in lines 6-20. It is responsible for grouping books by price. The nesting attribute in this case is *price* (line 8).

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

Figure 8: EBNF of UXQuery

The `xnest` operator consists of four parts:

1. The nesting attribute (line 8). The variable bound to this attribute is called the *nesting variable* (`$price` in the example);
2. The *source* tables, which contains the data that will be returned by the `xnest` operator. In the example, the source tables are *Book* and *SellBook* (lines 6-7).
3. The *header* element, which is the element that encloses the XML fragment returned by the `xnest` operator. In this example, the *header* element is *books* (line 12). The nesting attribute must appear either as an attribute of the header element or as a subelement of it.
4. One or more element groups (*ElGroup*), which are XML fragments that will be grouped according to the nesting attribute. The code in lines 13-18 of Figure 9 is an element group. This element group specifies books that will be returned according to their prices.

The XML view resulting from this example query is as follows:

```
...
<vendor id="2">
...
  <books price="38">
```

```
 1.<vendors>
 2.  {for $v in table("Vendor")
 3.   return
 4.   <vendor id="{$v/vendorid/text()}">
 5.      {$v/vendorname}
 6.      {xnest $b in table("Book"),
 7.             $sb in table("SellBook")
 8.       by $price in ($sb/price)
 9.       where $v/vendorid=$sb/vendorid
10.             and $sb/isbn=$b/isbn
11.       return
12.       <books price="{$price/text()}">
13.          {
14.          <book>
15.             {$b/isbn}
16.             {$b/title}
17.          </book>
18.          }
19.       </books>
20.      }
21.   </vendor>
22.  }
23.</vendors>
```

```
24.<vendors>
25.  {for $v in table("Vendor")
26.   return
27.   <vendor id="{$v/vendorid/text()}">
28.     {$v/vendorname}
29.     {let $b' := table("Book"),
30.          $sb' := table("SellBook")
31.      for $price in distinct-values($sb'/price)
32.      return
33.      <books price="{$price/text()}">
34.        {for $b in table("Book"),
35.             $sb in table("SellBook")
36.         where $v/vendorid=$sb/vendorid
37.               and $sb/isbn=$b/isbn
38.               and $sb/price=$price
39.         return
40.         <book>
41.           {$b/isbn}
42.           {$b/title}
43.         </book>
44.        }
45.      </books>
46.     }
47.  </vendor>
48.  }
49.</vendors>
```

Figure 9: Example of a query that uses the xnest operator (lines 1-23) and its translation to regular XQuery syntax (lines 24-49)

```
  <book>
    <isbn>1111</isbn>
    <title>Unix Network Programming</title>
  </book>
  <book>
    <isbn>2222</isbn>
    <title>Computer Networks</title>
  </book>
</books>
...
```

The need for a way of specifying groups in XQuery has been extensively discussed over the past few years. Following our `xnest` proposal in 2003 [11], the `groupby` operator was proposed in 2004 [25, 26]. They propose an algorithm that translates XQuery queries using pure XQuery syntax into equivalent queries that use `groupby`. Notice that they are going in the opposite direction from us: they start with a general XQuery query and produce another with `groupby` with the goal of query amelioration. Since our goal is to simplify query specification for the user and limit "bad" uses of `let`, we start with queries using `xnest` and produce pure XQuery queries. The approach proposed by [26] also requires a special XQuery processor which understands `groupby`. In 2004, BEA Systems proposed an extension to XQuery which allows grouping through a `group by` operator [8]. The motivation is the same as ours: to facilitate query specification. The difference between `group by` and `xnest` is that `xnest` allows the specification of heterogeneous groups (see Figure 10). With `group by`, groups are homogeneous. There was also a submission to W3C of a grouping extension to XQuery, authored by well known DB researchers; unfortunately, this submission is not yet available online. However, the Working Draft of XSLT 2.0 [34] now has a `for-each-group` operator which is also similar in purpose to `xnest`. The difference is that in XSLT, when grouping by a set of values, the same item may appear in more than one group. In `xnest`, each item will belong to exactly one group, thus avoiding update problems. XSLT 2.0 also allows grouping by patterns, which `xnest` does not. Due to the differences discussed above, we believe `xnest` allows the types of grouping that are useful in XML views (i.e. heterogeneous sets) while avoiding potential problems when updating.

4.1.1 Normalization to XQuery

A query containing `xnest` can be normalized to one using pure XQuery syntax. The normalized query corresponding to the query in Figure 9 (lines 1-23) is shown in Figure 9 (lines 24-49). The normalization process ensures that the nesting variable (in the example, *$price*) appears in the *Header* element (in the example, *books*) as an attribute or a sub-element. Notice that in the normalized query, we still use the input function `table`.

Continuing with the example, the `xnest` operation (lines 6-20) is normalized to the expression shown in lines 29-46. The expression consists of a `let/for` (lines 29-31) and an additional `for` (lines 34-44) for each element group (*ElGroup*) (lines 13-18) specified in the query.[4] In the normalization process, we introduce new variables in the `let` clause. These variables are primed ('), and correspond to the variables bound to source tables in the `xnest` operator. There will be one primed variable in the `let` clause for each source variable specified in the `xnest` operator[5].

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 38) we added a condition requiring that the book is sold at the price specified by *$price*.

Note that this example shows a nesting over a single attribute, but that it is possible to specify nests over more than one attribute. As an example, we could group books over price and year as follows:

```
...
{xnest $b in table("Book"), $sb in table("SellBook")
 by $price in ($sb/price), $year in ($b/year)
 where $v/vendorid=$sb/vendorid and $sb/isbn=$b/isbn
 return
 <books price="{$price/text()}" year="{$year/text()}">
 {
   <book>
     {$b/isbn}
     {$b/title}
   </book>
 }
 </books>
}
...
```

The query of Figure 9 has a single element group (*ElGroup*) (lines 13-18). In this example, it is not necessary to separate the conditions and variable bindings that appear in the `xnest` operator over the corresponding `for`s in the normalized query. We now show an example of where this is necessary.

Figure 10 shows a query that has two element groups (lines 21-26 and 27-32). In this case, the normalized query will have two `for`s, one for each of the element groups (lines 54-64 and 65-75). The variable bindings and where conditions must then be carefully analyzed in order to identify which of the `for`s they belong to. This is done by functions *fs:SubVariable(i)* and *fs:SubExpr(i)* in the normalization process shown below.

The normalization process described through the above examples can be formally stated as:

$$
\begin{array}{l}
[\text{xnest Variable}_1 \text{ in TableExpr}_1, \ldots, \text{Variable}_n \text{ in TableExpr}_n \\
\text{by NestVariable}_1 \text{ in } (\text{Variable}_{1_1}/\text{QName}_{1_1} \mid \ldots \mid \text{Variable}_{1_m}/\text{QName}_{1_m}), \\
\ldots, \text{NestVariable}_k \text{ in } (\text{Variable}_{k_1}/\text{QName}_{k_1} \mid \ldots \mid \text{Variable}_{k_m}/\text{QName}_{k_m}) \\
\text{where Expr return} \\
<\text{ElName AttName}_1=\text{"}\{\text{NestVariable}_1/\text{text()}\}\text{"} \ldots \text{AttName}_k=\text{"}\{\text{NestVariable}_k/\text{text()}\}\text{"}> \\
\{\text{ElGroup}_1\} \ldots \{\text{ElGroup}_m\} </\text{ElName}>]_{xnest} \\
== \\
\text{let Variable}'_1 := \text{TableExpr}_1, \ldots, \text{Variable}'_n := \text{TableExpr}_n
\end{array}
$$

[4] See Figure 8 for the definition of *ElGroup*.

[5] XQuery does not accept variable names with (') . However, we use them here for ease of explanation.

```
 1. <vendors>
 2.   {for $v in table("Vendor")
 3.    return
 4.    <vendor id="{$v/vendorid/text()}">
 5.       {$v/vendorname}
 6.       <address>
 7.          {$v/state}
 8.          {$v/country}
 9.       </address>
10.       {xnest $b in table("Book"),
11.              $sb in table("SellBook"),
12.              $d in table("DVD"),
13.              $sd in table("SellDVD")
14.        by $price in
              ($sb/price | $sd/price)
15.        where $v/vendorid=$sb/vendorid
16.              and $v/vendorid=$sd/vendorid
17.              and $sb/isbn=$b/isbn
18.              and $sd/asin=$d/asin
19.        return
20.        <products price="{$price/text()}">
21.           {
22.           <book>
23.              {$b/isbn}
24.              {$b/btitle}
25.           </book>
26.           }
27.           {
28.           <dvd>
29.              {$d/asin}
30.              {$d/dtitle}
31.           </dvd>
32.           }
33.        </products>
34.       }
35.    </vendor>
36.   }
37. </vendors>
```

```
38. <vendors>
39.   {for $v in table("Vendor")
40.    return
41.    <vendor id="{$v/vendorid/text()}">
42.      {$v/vendorname}
43.      <address>
44.         {$v/state}
45.         {$v/country}
46.      </address>
47.      {let $b' := table("Book"),
48.           $sb' := table("SellBook"),
49.           $d' := table("DVD"),
50.           $sd' := table("SellDVD")
51.       for $price in
      distinct-values($sb'/price|$sd'/price)
52.       return
53.       <products price="{$price/text()}">
54.         {for $b in table("Book"),
55.              $sb in table("SellBook")
56.          where $v/vendorid=$sb/vendorid
57.                and $sb/isbn=$b/isbn
58.                and $sb/price=$price
59.          return
60.          <book>
61.             {$b/isbn}
62.             {$b/btitle}
63.          </book>
64.         }
65.         {for $d in table("DVD"),
66.              $sd in table("SellDVD")
67.          where $v/vendorid=$sd/vendorid
68.                and $sd/asin=$d/asin
69.                and $sd/price=$price
70.          return
71.          <dvd>
72.             {$d/asin}
73.             {$d/dtitle}
74.          </dvd>
75.         }
76.       </products>
77.      }
78.    </vendor>
79.   }
80. </vendors>
```

Figure 10: Example of a query with two element groups (lines 1-37) and its translation to regular XQuery syntax (lines 38-80)

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), fs:IsolatedTables()
where fs:SubExpr(1) and (Variable$_{1_1}$ = NestVariable$_1$ and ... and Variable$_{k_1}$ = NestVariable$_k$)
return ElGroup$_1$ }
...
{for fs:SubVariable(m), fs:IsolatedTables()
where fs:SubExpr(m) and (Variable$_{1_m}$ = NestVariable$_1$ and ... and Variable$_{k_m}$ = NestVariable$_k$)
return ElGroup$_m$ }
</ElName>

The notation for the normalization process is the same as that in [27]. The process assumes that:

- {Variable$_{1_1}$, ..., Variable$_{1_m}$, ..., Variable$_{k_1}$, ..., Variable$_{k_m}$} $\subseteq$ {Variable$_1$, ..., Variable$_n$}

- The auxiliary function *fs:SubVariable(i)* returns all variables V_x referenced in $ElGroup_i$ and also all variables V_y appearing in a condition of the form:
"$V_x/QName_x$ **cmp** $V_y/QName_y$" or "$V_y/QName_y$ **cmp** $V_x/QName_x$" in *Expr* in the `where` clause of the `xnest` operator (**cmp** $\in \{=, <, >, !=, <=, >=\}$).

- The auxiliary function *fs:IsolatedTables()* returns all variables V_x not referenced in any *ElGroup* of the `xnest` operation, or in any `where` condition. Such variables reference what we call *isolated tables*, and they must be added to all `for`s of the normalized query (excluding the first `for`, which defines the nesting variables). An example of this situation follows the function definitions.

- The auxiliary function *fs:SubExpr(i)* returns every expression specified in *Expr* in the `where` clause of the `xnest` operator that references a variable returned by the function *fs:SubVariable(i)*.

Returning to the example of Figure 10, the first element group (*ElGroup*) (lines 21-26) references variable `$b`. Additionally, there is a where condition that uses `$b` and references `$sb` (`$sb/isbn=$b/isbn`, line 17). Hence the function *fs:SubVariable(1)* returns `$b` and `$sb`. These variables are used in the `for` clause corresponding to this element group (lines 54-55). The *where* conditions for this element group are found by function *fs:SubExpr(1)*, which analyzes the *where* condition of the `xnest` expression and takes all such conditions that references variables `$b` and `$sb`. A condition requiring that each book is sold by the price specified by `$price` is also added. The resulting *where* condition is shown in lines 56-58. The same process is done with the second element group (the one that builds the `dvd` element – lines 27-32).

When there are isolated tables in the `xnest`, i.e. tables which are not referenced in an element group and for which there are no where conditions, the tables are added in each `for` produced by the normalization process. As an example, suppose that there is an additional binding using a new variable `$w` to the Warehouse table in line 13a of Figure 10, and that no `where` condition is specified for `$w`. In this case, this instance of Warehouse is an isolated table, and the normalization process would add `$w` to the two `for`s under the `products` element (lines 55a and 66a). The semantics in this case is a cartesian product of table Warehouse with the joined tables Book and SellBook, and also with the joined tables DVD and SellDVD.

The normalized query is the one used by PATAXÓ to produce the XML view, as mentioned in Section 3.1. The XML view resulting from the sample query of Figure 10 is shown in Figure 1.

4.1.2 Expressive Power

UXQuery can express everything in the XQueryCore [5] except for: queries that refer to element order; recursive functions; is/is not operators; if/then/else expressions; sequences of expressions; disjunctions; function applications; and arithmetic and set operations. The restriction on sequences of expressions is due to the fact that the result of a query must be a single XML document, and the restriction on disjunctions relates to restrictions imposed by the algorithm we use to translate the view updates to the relational database [24]. Input functions are also limited to single relations, whereas in XQuery variables can be bound to the results of expressions.

Even with such limitations, UXQuery is capable of expressing most real world views we encountered in practice [12], and its expressive power is equivalent to that of DB2 DAD files [19].

4.2 Query Trees

Query trees are used as an internal representation of the XML view extraction query. This abstract representation enables reasoning about updates and the updatability of an XML view, using the structure of the XML view and the source of each XML element/attribute. These are syntax independent features, which allow us to work on a syntax independent level. Other systems in the literature, such as SilkRoute [29], XPERANTO (XQGM) [43] and Rainbow [52], also use internal structures (*view*

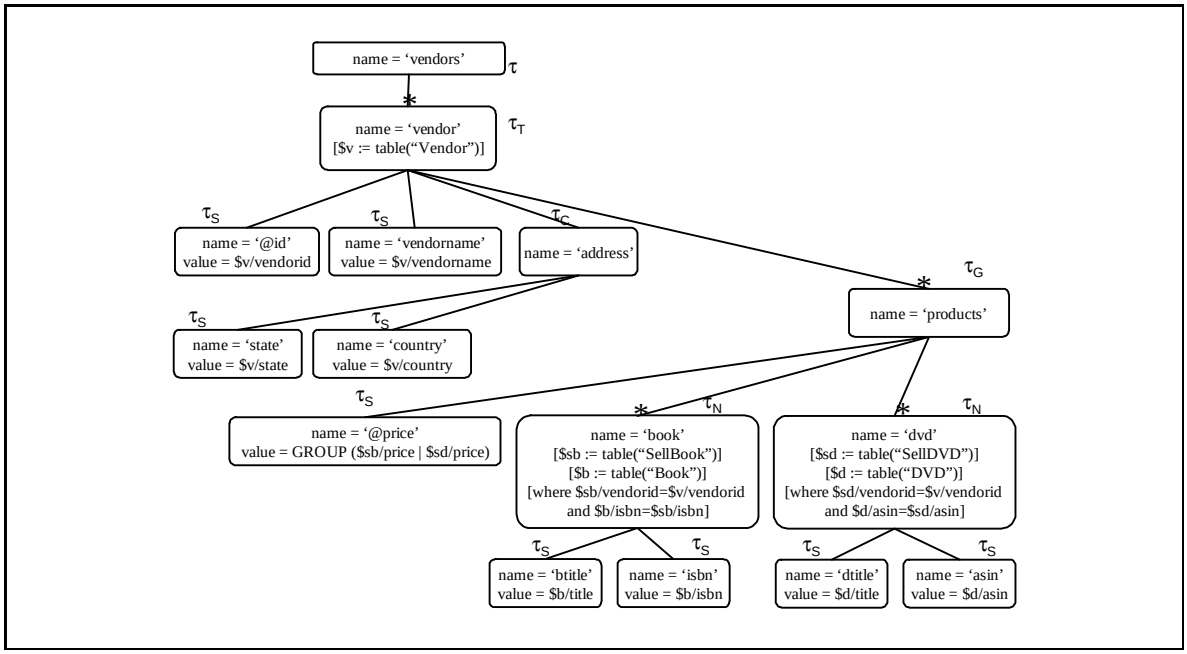


Figure 11: Query tree with grouped values

forests, XQGM and *view relationship graph*, respectively) which are easier to manipulate than a user level language.

Query trees were first introduced in [12]. Here, we present an extension which adds a new type of node called a *group node*. These nodes are used to represent nodes whose children are grouped according to a given value, and correspond to nodes returned by the UXQuery `xnest` operator.

After defining query trees, we introduce a notion which will be used to describe the mapping to relational queries: the abstract type of a query tree node. We use this notion of typing to define the semantics of query trees, and then present their result type DTD.

4.2.1 Query Trees Defined

An example of a query tree can be found in Figure 11 (ignore for now the types τ associated with nodes). This query tree retrieves *vendors*, and for each *vendor*, its *@id*, *vendorname*, *address* and a set of *book*s and *dvd*s grouped by *price* within *products*. The query tree resembles the structure of the resulting XML view. The root of the tree corresponds to the root element of the result. Leaf nodes correspond to attributes of relational tables, and interior nodes whose incoming edges are starred capture repeating elements (an *incoming* edge of a node n is an edge that connects n to its parent). The UXQuery corresponding to this query tree is shown in Figure 10, and the resulting XML instance is shown in Figure 1.

Query trees are very similar to the *view forests* of [29] and *schema-tree queries* presented in [6]. The difference is that, instead of annotating all nodes with the relational queries that are used to build the content model of a given node, we annotate interior nodes in the tree using only the selection criteria (not the entire relational query).

An annotation can be a *source* annotation or a *where* annotation. Source annotations bind variables to relational tables, and *where* annotations impose restrictions on the relational tables making use of the variables that were bound to the tables.

In the definitions that follow, we assume that $\mathcal{D}$ is a relational database over which the XML view is being defined. $\mathbb{T}$ is the set of table names of $\mathcal{D}$. $\mathbb{A}_T$ is the set of attributes of a given table $T \in \mathbb{T}$.

DEFINITION 4.1 *A query tree defined over a database $\mathcal{D}$ is a tree with a set of nodes $\mathbb{N}$ and a set of edges $\mathbb{E}$ in which:* ***Edges*** *are simple or starred ("*-edge"). An edge is* simple *if, in the corresponding XML instance, the child node appears exactly once in the context of the parent node (regardless of the database content), and* starred *otherwise.* ***Nodes*** *are as follows:*

1. *All nodes have a name that represents the tag name of the XML element associated with this node in the resulting XML view.*
2. *Leaf nodes have a value, which is either* projected *or* grouped. *Names of leaf nodes that start with "@" are considered to be XML attributes.*
3. Starred nodes *(nodes whose incoming edge is starred) may have one or more source annotations and zero or more where annotations. An exception is made for starred nodes with* group *children, which must have no source annotation.*
4. *A* Group *node (one that has a grouped value) must have siblings that are starred nodes or group nodes of a restricted form (see Definition 4.4).*

Since we map XML view queries to relational view queries, nodes with the same name in the query tree may cause ambiguities in the mapping (a relation cannot have two attributes with the same name [50]). For simplicity, in this paper we will ignore this problem and use unique names for nodes in the query trees, and as a consequence, in element constructors in UXQuery. In [9], we present a very simple solution to this problem. When we build the query tree, we append a numeric suffix (generated according to the Global Order Encoding [47]) to each node name. This way, the relational views are generated with unique attribute names. This numbering schema is used only internally, and the user is not aware of it.

Returning to the example in Figure 11, there is a *-edge from the root (named *vendors*) to its child named *vendor*, indicating that in the corresponding XML instance there may be several *vendor* subelements of *vendors*. There is a simple edge from the node named *vendor* to the node named *vendorname*, indicating that there is a single *vendorname* subelement of *vendor*. The node named *@id* will be mapped to an XML attribute instead of an element.

Before giving an example of how values are associated with nodes, we define *source* and *where* annotations on nodes of a query tree.

DEFINITION 4.2 *A* source annotation *s within a starred node n is of the form $[\$x := \text{table}(T)]$, where $\$x$ denotes a variable and $T \in \mathbb{T}$ is a relational table. We say that $\$x$ is bound to T by s.*

DEFINITION 4.3 *A* where annotation *on a starred node n is of the form* [where $\$x_1/A_1$ op Z_1 AND ... AND $\$x_k/A_k$ op Z_k], *$k \geqslant 1$, where $A_i \in \mathbb{A}_{T_i}$ and $\$x_i$ is bound to T_i by a source annotation on n or some ancestor of n. The operator* op *is a comparison operator $\{=, \neq, >, <, \leqslant, \geqslant\}$. Z_i is either a literal (integer, string, etc.) or an expression of the form $\$y/B$, where $B \in \mathbb{A}_T$ and $\$y$ is bound to T by a source annotation on n or some ancestor of n.*

Continuing with the example of Figure 11, the *vendor* node is annotated with a binding for *$v* (to table Vendor), and has several children at the end of simple edges (*@id*, *vendorname*, and *address*). The value of its *id* attribute is specified by the path *$v/vendorid*, indicating that the content of the XML view attribute *id* will be generated using column *vendorid* of the table *Vendor*. The value of *vendorname* is specified by the path *$v/vendorname*. The node *address* is more complex, and is composed of *state* and *country* subelements.

The node *products* has two *-edge children, *book* and *dvd*, and a group child, *@price*. Source annotations on the *book* node include bindings for *$b* (Book) and *$sb* (SellBook), and its where annotations connect tuples in SellBook to tuples in Book, and tuples in SellBook with tuples in Vendor (join conditions). Node *dvd* has source annotations for *$d* (DVD) and *$sd* (SellDVD). Its where annotation connects tuples in SellDVD to tuples in DVD and tuples in SellDVD with tuples in Vendor.

DEFINITION 4.4 *The value of a node n can be* projected *or* grouped.
A projected value *is of form $\$x/A$, where $A \in \mathbb{A}_T$ and $\$x$ is bound to table T by a source annotation on n or some ancestor of n.*

A grouped value *is of form* GROUP($\$x_1/A_1$ | ... | $\$x_m/A_m$)*, where* $m \geqslant 1$ *and* $A_i \in \mathbb{A}_{T_i}$ *and* $\$x_i$ *is bound to* T_i *by a source annotation on a* sibling *node of* n*. The domains of* $A_1, ...A_m$ *in* $\mathcal{D}$ *must be the same. Group nodes with the same parent must be defined over the same set of variables* x_1*, ...,* x_m*, and must have* m *siblings* $b_1, ...b_m$ *whose incoming edges are starred*[6]*. Furthermore, the parent of node* n *must be starred, and it must have no source annotations.*

Still in the example of Figure 11, the *@price* node has a grouped value which references variables *$sb* (declared on its sibling node *book*) and *$sd* (declared on its sibling node *dvd*). The XML instance resulting from this query tree will take values from both tables *SellBook* and *SellDVD*, grouping by their price.

From now on, we assume UXQuery queries are non-empty, and consequently, their corresponding query trees are non-empty.

4.2.2 Abstract Types

In our mapping strategy, it will be important to recognize nodes that play certain roles in a query tree. In particular, we identify six abstract types of nodes: τ, τ_T, τ_N, τ_C, τ_S and τ_G. We call them *abstract types* to distinguish them from the type or DTD of the XML view elements.

Nodes in the query tree are assigned abstract types as follows:

1. The root has abstract type τ.
2. Each leaf node has abstract type τ_S (Simple).
3. Each non-leaf node with an incoming simple edge has abstract type τ_C (Complex).
4. Each starred node which is either a leaf node or whose subtree has only simple edges has an abstract type of τ_N (Nested).
5. Each starred node with one or more children with a *grouped* value has an abstract type τ_G (with Grouped children).
6. All other starred nodes have abstract type τ_T (Tree).

Note that each node has exactly one type unless it is a starred leaf node, in which case it has types τ_S and τ_N.

As an example of this abstract typing, consider the query tree in Figure 11, which shows the type of each of its nodes. Since *book* and *dvd* are repeating nodes whose descendants are non-repeating nodes, their types are τ_N rather than τ_T. Also, since *products* has a child *@price* with *grouped value*, its abstract type is τ_G.

The motivation behind abstract types is as follows. To map updates in the XML view to updates in the underlying relational database, we must be able to identify a mapping from the column of a tuple in the relational database to an element or attribute in the XML view. Ideally, this mapping is 1:1, i.e. each attribute of a tuple occurs at most once in the XML view and can therefore be updated without introducing side-effects into the view. In general, however, it may be a 1:n mapping. The class of views allowed by our query trees and its associated abstract type views captures this mapping intrinsically.

Specifically:

- $\tau_T/\tau_N/\tau_G$ identifies potential tuples in the underlying relational database. Nodes of type $\tau_T/\tau_N/\tau_G$ are mapped to tuples, and the node itself serves as a tuple delimiter. A node of type τ_T may have children of type τ_T, i.e. nesting is allowed.

[6]Notice that we do not require that ($\$x_1/A_1$ | ... | $\$x_m/A_m$) in the group operation be in the same order as $b_1, ...b_m$.

- τ_S identifies relational attributes (columns). A node of type τ_S must have a node of type τ_T, τ_N or τ_G as its ancestor. Starred leaf nodes are an exception to this rule: they need not to have such ancestor.

- τ_C identifies complex XML elements. Since they do not carry a value, this type of node is not mapped to anything in the relational model. Nodes of type τ_C are present in our model to allow more flexible XML views, but are not important in the mapping process.

XML views produced by query trees and their associated abstract types can be easily mapped to a set of corresponding relational views, as we will show in Section 6. However, before turning to the mapping we prove two facts about query trees that will be used throughout the paper.

PROPOSITION 4.5 *There is at least one τ_N node in the abstract type of a query tree qt.*

Proof: Since query trees are assumed to be non-empty, *qt* must have at least one leaf. This means that *qt* must have at least one starred node n, since the leaf node has a value which involves at least one variable which must be defined in some source annotation attached to a starred node. Since the tree is finite, at least one of these starred nodes is either a leaf node or has a subtree of simple edges, i.e. the starred node is a τ_N node.

PROPOSITION 4.6 *There is at most one τ_N node along any path from a leaf with projected value to the root in the abstract type of a query tree qt.*

Proof: Suppose there are two τ_N nodes, n_1 and n_2, along the path from some leaf with projected value to the root of *qt*. Without loss of generality, assume that n_1 is the ancestor of n_2. By definition of τ_N, n_2 must be a starred node. Therefore n_1 has a *-edge in its subtree, a contradiction.

We will refer 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 *dvd* in Figure 11 is referred to as *τ_N(dvd)*, and its type (DTD) is *<!ELEMENT dvd (dtitle, asin)>*.

4.2.3 Semantics of Query Trees

The semantics of a query tree follows the abstract type of its nodes, and can be found in Algorithm 1. The algorithm constructs the XML view resulting from a query tree *qt* recursively, and starts with n being the root of the query tree. The basic idea is that the source and where annotations in each starred node n are evaluated in the database instance d, producing a set of tuples. The algorithm then iterates over these tuples, generating one element corresponding to n in the output for each of these tuples and evaluating the children of n once for each tuple.

The *bindings*{} hash array keeps the values of variables taken from the underlying relational database. We assume that values in *bindings*{} are represented as *$x/A = 1, $x/B = 2*, where $$x$ is a variable bound to a relational table T, A and B are the attributes of T and 1 and 2 are the values of attributes A and B in the current tuple of T.

4.2.4 View Schema

Query tree views defined over a relational database have a well-defined schema that is easily derived from the tree. Given a query tree, its DTD is generated as follows:

1. For each attribute leaf node named @A with parent named E, create an attribute declaration <!ATTLIST E @A CDATA #REQUIRED>

2. For each non-attribute leaf node named E, create an element declaration <!ELEMENT E (#PCDATA)>

3. For each non-leaf node named E, create an element declaration <!ELEMENT E (E_1, ..., E_k)>, where $E_1, ..., E_k$ are non-attribute child nodes of E connected by a simple or starred edge. In case E_i is connected to E by a starred edge, add a "*" after E_i. In case $k = 0$, then create an element declaration <!ELEMENT E EMPTY>

As an example, the DTD of the view produced by the query tree shown in Figure 11 is:

```
<!ELEMENT vendors (vendor*)>
<!ELEMENT vendor (vendorname, address, products)>
<!ATTLIST vendor id CDATA #REQUIRED>
<!ELEMENT vendorname (#PCDATA)>
<!ELEMENT address (state, country)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT products (book*,dvd*)>
<!ATTLIST products
          price CDATA #REQUIRED>
<!ELEMENT book (btitle, isbn)>
<!ELEMENT btitle (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT dvd (dtitle, asin)>
<!ELEMENT asin (#PCDATA)>
<!ELEMENT dtitle (#PCDATA)>
```

Note that all (#PCDATA) elements are required. When the value of a relational attribute is *null*, we produce an element with a distinguished *null* value. This makes it easier for the user to distinguish between a value which is not known (*null*) and a value which is known to be the empty string.

We could also have chosen to omit the element tag when the value of that element is *null*. However, using a distinguished *null* value has several advantages. First, it facilitates modifying a *null* value to some other value: if tag t is omitted from the view, the user must know whether or not an element with tag t can be added at a particular point in the XML view. Second, it makes our update translation easier: If modifying a *null* value required inserting a new tag in the view, then this insertion in the view would translate to a modification in the underlying relational database. Similarly, changing from some known value to *null* would require a deletion in the view but would be mapped to a modification in the underlying relational database. Our strategy is to map an update (e.g. insertion, deletion or modification) in the XML view to the same type of update in the underlying relational database.

In our implementation, we use XML Schema instead of DTDs since it supports data types, thus making the schema checking more accurate. The generation of the schema is analogous to the DTD generation shown above. The data types are taken from the database metadata. We use a type conversion table that maps SQL types to XML Schema simple types (string, integer, float, etc.). We use DTDs in this paper for ease of explanation.

4.3 From UXQuery to Query Trees

As mentioned before, query trees are used as an intermediate representation of the view definition query. We must therefore define how a view definition query expressed in UXQuery is translated to its corresponding query tree. To illustrate the mapping process, we start with the query of Figure 7. For clarity, the query is presented again in Figure 12 together with its query tree.

Each XML element specified in the query is represented by a node in the query tree. Each node in the query tree needs a name, and possibly a value (if it is a leaf node). Since XML elements and attributes can be constructed in three distinct ways in UXQuery, we analyze each case separately:

- The leaf element is generated by an expression `{$x/A}`: in this case, the corresponding node in the query tree has name A and value $\$x/A$.
- The leaf element is constructed by an expression `<tagName> {$x/A/text()} </tagName>`: in this case, the corresponding node in the query tree has name *tagName* and value $\$x/A$.
- The leaf is an attribute constructed by an expression `attName="{$x/A/text()}"`: in this case, the node in the query tree has name *@attName* and value $\$x/A$.

```
 1. <vendors>
 2.  {for $v in table('Vendor')
 3.   return
 4.   <vendor id='{$v/vendorid/text()}'>
 5.    {$v/vendorname}
 6.    {for $w in table('Warehouse')
 7.     where $v/vendorid=$w/vendorid
 8.     return
 9.     <warehouse>
10.       <idWarehouse>
            {$w/depid/text()}
          </idWarehouse>
11.       <address>
12.         <street>{$w/address/text()}</street>
13.         {$w/city}
14.         {$w/state}
15.         {$w/country}
16.       </address>
17.     </warehouse>
18.    }
19.   </vendor>
20.  }
21. </vendors>
```

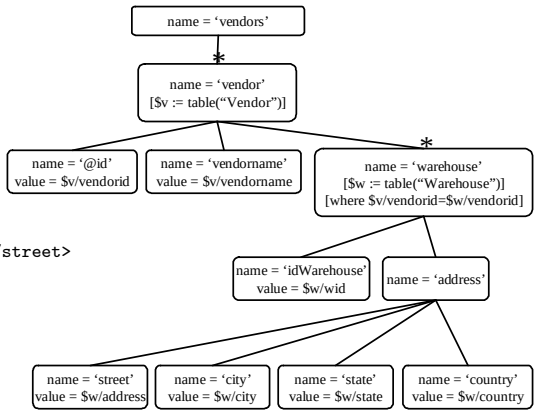


Figure 12: Example of an UXQuery query that joins two relations, and its query tree

As an example, the expression `$v/vendorname` in the query of Figure 12 is mapped to a node named *vendorname* in the query tree. As an example of mapping of an attribute, see node *@id*.

An exception to the above rules is an element or attribute which uses a nesting variable to specify its content. For example, attribute `price` in the query of Figure 10 is constructed using variable `$price` as its content (line 20). The variable `$price` was specified as `$price in ($sb/price | $sd/price)` (line 14). In this case, the rules for the node name are the same as above (in this example, the node will be named *price*), but its value is *GROUP($sb/price | $sd/price)*. The rule for this case can be specified as:

- The leaf element `tagName` is specified by a nesting variable `$y`, and is constructed as `<tagName>{$y}</tagName>`. Variable `$y` is in turn specified as `$y in ($x`$_1$`/A`$_1$ `| ... | $x`$_n$`/A`$_n$`)`. The corresponding node will be named *tagName* and its value will be $GROUP(\$x_1/A_1 \mid ... \mid \$x_n/A_n)$.

- The attribute `attName` is specified by a nesting variable `$y`, and is constructed as `attName ="{$y/text()}"`. Variable `$y` is in turn specified as `$y in ($x`$_1$`/A`$_1$ `| ... | $x`$_n$`/A`$_n$`)`. The corresponding node will be named *@attName* and its value will be $GROUP(\$x_1/A_1 \mid ... \mid \$x_n/A_n)$.

Non-leaf elements can only be constructed with an expression of type `<tagName>` {`content`} `</tagName>`, where `content` are other element constructors, `for`s and/or `xnest`s. In this case, the corresponding node in the query tree will have name *tagName*, but no value. As an example, the XML element `address` in the query of Figure 12 is a non-leaf element whose content is four element constructors. Its corresponding node in the query tree is named *address*, and it has no value.

Nodes in the query tree are connected to represent the parent/child relationship of XML elements in the view definition query. As an example, the node *address* is connected to nodes *street*, *city*, *state* and *country* in the query tree of Figure 12. In the view definition query, elements `street`, `city`, `state` and `country` are children of `address`. We will explain how starred edges are identified later.

Source and where annotations are identified as follows. Each `for` expression in the view definition query has variable bindings, optional where conditions and a return clause followed by an element constructor. Suppose this element is named `e`. The variable bindings are placed as source annotations in the node e that represents element `e` in the query tree. A variable binding of type `$x in table("X")` becomes a source annotation of type [*$x := table("X")*]. The where conditions (if any) are placed in node e as where annotations (`where x` becomes [*where x*]). After this, we change the edge that

connects e to its parent to a *-edge. As an example, the query of Figure 12 has a `for` expression at line 2. The expression has an element constructor after the `return` clause that constructs the element `vendor` (line 4). As a consequence, the node *vendor* in the query tree is a starred node, and it has a source annotation [*$v := table("Vendor")*].

When a query has an `xnest` operation, the source and where annotations are identified using the functions *fs:SubVariable(i)* and *fs:SubExpr(i)*, shown in Section 4.1.1. The *Header* element is mapped to a node that has a *-edge, but no source annotation. In the query of Figure 10, the *Header* element is `products`, and the corresponding node is shown in the query tree of Figure 11. After this query tree is typed, this node will receive a type τ_G.

The root element of each element group in the query receives a *-edge. The source annotations are selected using the functions *fs:SubVariable(i)* and *fs:IsolatedTables()* to identify the relevant variables for that node. In the same way, the function *fs:SubExpr(i)* is used to identify the where annotations for the node. As an example, node *book* in Figure 11 has source annotations [*$b := table("Book")*] and [*$sb := table("SellBook")*]. Similarly, its where annotation is [*where $v/vendorid=$sb/vendorid AND $b/isbn=$sb/isbn*].

The `order by` clause of UXQuery does not have a corresponding construction in query trees. This is not a problem, since the purpose of query trees is to drive the mapping to relational views and from there to the underlying relational database, which does not have a concept of order.

5 Update Language

In this section, we present a simple update language for XML views, and describe how we check for schema conformance after updates.

Although no standard has been established for an XML update language, several proposals have appeared [1, 46, 7, 37]. The language described below is much simpler than any of these proposals and in some sense can be thought of as an internal form for one of these richer languages (assuming a static translation of updates [7]). The simplicity of the language allows us to focus on the key problem we are addressing.

Updates are specified using path expressions to point to a set of target nodes in the XML tree at which the update is to be performed. For insertions and modifications, the update must also specify a Δ containing the new values.

DEFINITION 5.1 *An update operation u is a triple <t, Δ,* ref*>, where t 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 [20] which indicates where the update is to occur.*

DEFINITION 5.2 *An update path* ref *is of the form $p_1/p_2/.../p_n$ where p_i is either a label l_i or a qualified label $l_i[c_1$ and c_2 and ... $c_m]$. Each p_i is called a* step *of P. Each c_i is a qualification of the form $A = x$, where A is a label and x is an atomic value (string, integer, etc).*

The path expression *ref* is evaluated from the root of the tree and may yield a set of nodes which we call *update points*. In the case of modify, it must evaluate to a set of leaf nodes. We restrict the filters used in *ref* to conjunctions of comparisons of attributes or leaf elements with atomic values, and call the expression resulting from removing filters in *ref* the *unqualified portion* of *ref*. For example, the unqualified portion of `/vendors/vendor[@id="01"]` is `/vendors/vendor`.

DEFINITION 5.3 *An update path* ref *is* valid *with respect to a query tree qt iff the unqualified portion of* ref *is non-empty when evaluated on qt.*

For example, `/vendors/vendor[@id="01"]/vendorname` is a valid path expression with respect to the query tree of Figure 11, since the unqualified path `/vendors/vendor/vendorname` is non-empty when evaluated on that query tree.

The semantics of insert is that Δ is inserted as a child of the nodes indicated by *ref*; the semantics of modify is that the atomic value Δ overwrites the values of the leaf nodes indicated by *ref*; and the semantics of a delete is that the subtrees rooted at nodes indicated by *ref* are deleted.

The following examples refer to Figure 1:

EXAMPLE 5.1 *To insert a new book with title "New Book" and isbn "9999" selling for $38 under the vendor with id="01" we specify:*
t = insert,
ref = `/vendors/vendor[@id="01"]/products[@price="38"]`,

```
Δ = {<book>
        <btitle>New Book</btitle><isbn>9999</isbn>
     </book>}.
```

EXAMPLE 5.2 *To change the* vendorname *of the vendor with id = "01" to* Amazon.com *we specify:*
t = modify,
ref = `/vendors/vendor[@id="01"]/vendorName`,
Δ = {*Amazon.com*}.

EXAMPLE 5.3 *To delete all vendors of state WA we specify:*
t= delete,
ref = `/vendors/vendor/[state="WA"]`.

5.1 Schema conformance

Note that not all insertions and deletions make sense since the resulting XML view may not conform to the schema of the query tree (for details, see Section 4.2.4). For example, the deletion specified by the path `/vendors/vendor/vendorname` would not conform to the DTD of the query of Figure 11 since *vendorname* is a required subelement of *vendor*. We must also check that Δ's inserted and subtrees deleted are correct.

DEFINITION 5.4 *An update <t,*Δ*,*ref*> against an XML view specified by a query tree qt is* correct *iff*

- ref *is valid with respect to qt, according to Definition 5.3;*
- *if t is a modification, then the unqualified portion of* ref *evaluated on qt arrives at a node whose abstract type is* τ_S*;*
- *if t is an insertion, then the unqualified portion of* ref *extended with the root of* Δ *evaluated on qt arrives at a node whose incoming edge is starred (equivalently, its abstract type is* τ_T*,* τ_G *or* τ_N*);*
- *if t is a deletion, then the unqualified portion of* ref *evaluated on qt arrives at a node whose incoming edge is starred;*
- *if nonempty, then* Δ *conforms to the DTD of the element arrived at by* ref.

For example, the deletion of Example 5.3 is correct since *vendor* is a starred subelement of *vendors*. However, the deletion specified by the update path `/vendors/vendor/vendorname` is not correct since *vendorname* is of abstract type τ_S. The deletion specified by the invalid update path `/vendors/vendor/dvd` is also incorrect.

As another example, the insertion of Example 5.1 is correct since *book* (arrived at by `/vendors/vendor/products`) is a starred subelement of *products*, the DTD for *book* is <!ELEMENT book (btitle, isbn)>, and Δ conforms to this DTD. However, the following insertion would not be correct for the update path `/vendors/vendor[@id="01"]/products[@bprice="38"]` and Δ = `<book><rating>Children</rating></book>`, since the *isbn* and *btitle* subelements are missing, and *book* does not have a *rating* subelement.

6 Mapping

In our approach, updates over an XML view are translated to SQL update statements on a set of corresponding relational view expressions. Existing techniques such as [24, 35, 38, 3, 48] can then be used to accept, reject or modify the proposed SQL updates.

In order to do so, it is first necessary to map an XML view query to a relational view query. As we will show later in this section, there are cases where a single XML view query must be mapped to a *set* of relational view queries.

The proofs for the theorems presented in this section are available at the Appendix.

6.1 Mapping to Relational Views

In this section, we discuss how an XML view constructed by a query tree is mapped to a set of corresponding relational view expressions. There are two main steps in the mapping process: *map* and *split*. The *map* process maps a query tree with a single τ_N node to a relational view query, and the *split* process deals with query trees that have more than one node of type τ_N. It splits the query tree into several *split trees*, so that each of them has a single node of type τ_N. Then, the *map* process can be applied.

We start by showing the *map* process, and then we discuss the *split* process in detail.

6.1.1 Map

Given a query tree qt with only one τ_N node, the corresponding SQL view statement is generated as follows:

- Join together all tables found in source annotations (called *source tables*) in a given node n in qt, using the where annotations that correspond to joins on source tables in n as inner join conditions. If no such join condition is found then use "true" (e.g. 1=1) as the join condition, resulting in a cartesian product. Call these expressions *source join expressions*.

- Use the hierarchy implied by the query tree to left outer join source join expressions in an ancestor-descendant direction, so that ancestors with no children still appear in the view. The conditions for the outer joins are captured as follows: If node a is an ancestor of n and a where annotation in n specifies a join condition on a table in n with a table in a, then use this annotation as the join condition for the outer join. As with inner joins, if no condition for the outer join is found, then use "true" as the join condition so that if the inner relation is empty, the tuples of the outer will still appear.

- Use the remaining where annotations (the ones that were not used as inner or outer join conditions) in an SQL where-clause and project the values of leaf nodes. The resulting SQL view statement represents an unnested version of the XML view.

According to the above procedure, *source join expressions* are as follows:

```
<source table> AS <source variable> INNER JOIN
<source table> AS <source variable> INNER JOIN ...
ON <inner joincond>
```

The complete SQL expression resulting from the mapping process is:

```
SELECT <leaf value> AS <leaf name>, ...,
       <leaf value> AS <leaf name>
FROM (<source join expression> LEFT JOIN
  <source join expression> ON <outer joincond>) LEFT JOIN ...
WHERE <remaining "where" annotation>  AND ...
     AND <remaining "where" annotation>
```

For example, the relational view query corresponding to the query tree in Figure 12 is:

```
SELECT v.vendorid AS id, v.vendorname AS vendorname, w.depid AS idWarehouse,
       w.address AS street, w.city AS city, w.state AS state, w.country AS country
FROM (Vendor AS v INNER JOIN Warehouse AS w ON v.vendorid=w.vendorid)
```

The mapping algorithm is shown in Algorithm 2. The auxiliary functions used in this algorithm have obvious meanings. The one that is not so obvious is function *variable(n)*, which returns the variable used in the value of a leaf node (without the $ symbol). For example, if the value of node n is $\$x/A$, then *variable(n)* returns x. When the parameter is a source annotation s, then the function returns the variable referenced in this source annotation without the $ (e.g. with $s = \$x$ *in table("X")*, function *variable(s)* returns x). Function *attribute(n)* returns the relational attribute that was used to specify the value of a leaf node. Using the example of value of leaf node n above, *attribute(n)* returns A.

6.1.2 Split

For a query tree with more than one τ_N node, the process shown above is incorrect. As an example, consider the query tree of Figure 11 which has two τ_N nodes (*book* and *dvd*). If we follow the mapping process described above, the tables DVD and Book will be joined, resulting in a cartesian product. In this expression, a book is repeated for each DVD, violating the semantics of the UXQuery query that corresponds to this query tree (Figure 10 (lines 1–33)). We must therefore split a query tree into sub-query trees containing exactly one τ_N node each before generating the corresponding relational view queries. After the splitting process, each sub-query tree produced is mapped to a relational view query as explained above.

The splitting process consists in isolating a node n of type τ_N in the query tree qt, and taking its subtree as well as its ancestors and their non-repeating descendants (types τ_C and τ_S) to form a new tree qt_i. Recall that qt must have at least one τ_N node by Proposition 4.5.

The first step to generate qt_i is to copy qt to qt_i. Then, delete from qt_i all subtrees rooted at nodes of type τ_N, except for the subtree rooted at n. Observe that deleting a subtree r may change the abstract type of the ancestors of r. Specifically, if r has an ancestor a with type τ_T, and r is a's only starred descendant, then the type of a becomes τ_N after the deletion of r. Continue to delete subtrees rooted at nodes of type τ_N in qt_i and retype ancestors until n is the only node of type τ_N in qt_i. The process is repeated for every node of type τ_N in qt and results in exactly one τ_N node per split tree.

Also, it is necessary to remove parts of the value of group nodes so that variable references are correct in each split tree. As an example, the query tree of Figure 11 has a group node *price* whose value is *GROUP($sb/price | $sd/price)*. The two split trees generated by the split algorithm will have a node *price* referencing just one of the variables each ($*sb* or $*sd*).

Formally, the *split* algorithm (Algorithm 3) splits a query tree qt producing one split tree qt_i for each node of type τ_N in qt.

The result of this process for the query tree of Figure 11 is shown in the Electronic Appendix. Using these split trees, the corresponding relational view queries *ViewBook* and *ViewDVD* are:

```
CREATE VIEW VIEWBOOK AS
SELECT v.vendorId AS id, v.vendorname AS vendorname, v.state AS state,
       v.country AS country, sb.price AS price, b.isbn AS isbn, b.title AS btitle
FROM (Vendor AS v LEFT JOIN (SellBook AS sb INNER JOIN
                             Book AS B ON b.isbn=sb.isbn) ON v.vendorId=sb.vendorId);

CREATE VIEW VIEWDVD AS
SELECT v.vendorId AS id, v.vendorname AS vendorname, v.state AS state,
       v.country AS country, sd.price AS price, d.asin AS asin, d.title AS dtitle
FROM (Vendor AS v LEFT JOIN (SellDVD AS sd INNER JOIN
                             DVD AS d ON d.asin=sd.asin) ON v.vendorId=sd.vendorId)
```

As described above, *split* takes as input the original query tree qt and produces as output a set of query trees $\{qt_1, ..., qt_n\}$, each of which has one τ_N node; *map* takes $\{qt_1, ..., qt_n\}$ as input and produces

	id	vendorname	state	country	price	btitle	isbn	dtitle	asin
t_1	1	Amazon	WA	US	38	Unix Network Programming	1111	NULL	NULL
t_2	1	Amazon	WA	US	29	Computer Networks	2222	NULL	NULL
t_3	1	Amazon	WA	US	29	NULL	NULL	Friends	D1111
t_4	2	Barnes and Noble	NY	US	38	Unix Network Programming	1111	NULL	NULL
t_5	2	Barnes and Noble	NY	US	38	Computer Networks	2222	NULL	NULL

Figure 13: Tuples resulting from *evalRel(eval(qt, d))* for the query tree of Figure 11

a set of relational view expressions $\{V_1, ..., V_n\}$, where each V_i is produced from qt_i as described above. It follows directly from these algorithms that:

PROPOSITION 6.1 *The number of relational view expressions in* map(split(qt)) *is the number of* τ_N *nodes in* qt.

6.1.3 Correctness

The correctness of the set of relational view expressions resulting from *map* and *split* can be understood in the following sense: Each tuple in the bindings relations (generated during the execution of the eval algorithm (Algorithm 1)) for the XML view is in one or more instances of the corresponding relational views. Of course, this bindings relation is not materialized during the execution of *eval*, so we now show how to capture it using the query tree and its resulting XML view as source to materialize a relation that we call *evalRel*, which in some sense corresponds to the bindings relation mentioned above. To be more precise, we define the following:

DEFINITION 6.2 *The* evaluation schema S *of a query tree* qt *is the set of all names of leaf nodes in* qt.

As an example, the *evaluation schema* of the query tree of Figure 11 is S = (*id*, *vendorname*, *state*, *country*, *price*, *btitle*, *isbn*, *dtitle*, *asin*).

DEFINITION 6.3 *Let* x *be an XML instance of a query tree* qt *with evaluation schema* S*, in which the instance nodes are annotated by the query tree type from which they were generated. Let* $\{n_1, ..., n_k\}$ *be the set of all deepest* τ_N *or* τ_T *instance nodes for some root to leaf path in* x*. Let* p_i *be the set of nodes in the path from* n_i *to the root of* x*. An* evaluation tuple *of* x *is created from each* n_i *by associating the value of each leaf node* l *that is a descendant of* n_i *or of some node in* p_i *with the attribute in* S *corresponding to the name of* l*, and leaving the value of all other attributes in* S *null.*

The multi-set[7] *of all evaluation tuples of* x *is called its* evaluation relation *and is denoted* evalRel(x).

For example, Figure 13 shows the result of *evalRel(x)* for the query tree qt of Figure 11 and the XML view x of Figure 1. Recall that x is actually the evaluation of query tree qt over the database instance d using algorithm *eval* (Algorithm 1), which produces the XML view x as a result. Thus, *evalRel(eval(qt, d)) = evalRel(x)*.

The evaluation relation *evalRel* carries all data that is in the leaves of the XML view. To prove the correctness of our approach, we must show that this data corresponds to data in the relational views generated by our mapping process (*map* and *split*). Since a single XML view can be mapped to more than one relational view, we first collect the relational views together using outer union and call the resulting relation *relOuterUnion*.

DEFINITION 6.4 *Let* $\{V_1, ..., V_n\}$ *be defined over a relational schema* $\mathcal{D}$*, and* d *be an instance of* $\mathcal{D}$*. Let* evalV(V,d) *denote the instantiation of view definition* V *over the database instance* d*. Then* relOuterUnion($\{V_1, ..., V_n\}, d$) *denotes the set of relational instances that result from taking the outer*

[7]Note that SQL queries may return repeated tuples, and therefore we can have repeated evaluation tuples in *evalRel*. Thus, *evalRel* is a multi-set instead of a set.

	id	vendorname	state	country	price	btitle	isbn	dtitle	asin
t_1	1	Amazon	WA	US	38	Unix Network Programming	1111	NULL	NULL
t_2	1	Amazon	WA	US	29	Computer Networks	2222	NULL	NULL
t_3	2	Barnes and Noble	NY	US	38	Unix Network Programming	1111	NULL	NULL
t_4	2	Barnes and Noble	NY	US	38	Computer Networks	2222	NULL	NULL
t_5	1	Amazon	WA	US	29	NULL	NULL	Friends	D1111
t_6	2	Barnes and Noble	NY	US	NULL	NULL	NULL	NULL	NULL

Figure 14: Tuples resulting from *relOuterUnion({ViewBook,ViewDVD}, d)*

	id	vendorname	state	country	price	btitle	isbn
t_1	1	Amazon	WA	US	38	Unix Network Programming	1111
t_2	1	Amazon	WA	US	29	Computer Networks	2222
t_3	2	Barnes and Noble	NY	US	38	Unix Network Programming	1111
t_4	2	Barnes and Noble	NY	US	38	Computer Networks	2222

Figure 15: Tuples on *ViewBook*

union of the evaluation of each V_i over d: relOuterUnion($\{V_1, ..., V_n\}$, d) = evalV(V_1, d) $\bigcup$ ... $\bigcup$ evalV(V_n, d), *where* $\bigcup$ *denotes outer union.*

For example, *relOuterUnion*({*ViewBook*, *ViewDVD*}, *d*) is the result of the outer union of *evalV* (*ViewBook*, *d*) and *evalV* (*ViewDVD*, *d*), which is shown on Figure 14. The evaluations *evalV* (*ViewBook*, *d*) and *evalV* (*ViewDVD*, *d*) are shown in Figures 15 and 16, respectively.

It is now possible to compare the data in the XML view (*evalRel*) with data in the relational views generated by the mapping process (*relOuterUnion*). The correctness of the set of relational views resulting from *map* and *split* can now be understood in the following sense:

THEOREM 6.5 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then:* evalRel(eval(*qt*, *d*)) $\subseteq$ relOuterUnion(map(split(*qt*)), *d*).

Note that the set of tuples in Figures 13 and 14 are not exactly the same (and that Theorem 6.5 uses "$\subseteq$" instead of "="). For instance, tuple t_6 of *relOuterUnion* (Figure 14) is not in *evalRel* (Figure 13). This is because the XML instance of Figure 1 does not have any dvd sold by vendor *Barnes and Noble*, thus there is a tuple [*2, Barnes and Noble, NY, US, null, null, null*] in *ViewDVD* which was added by the LEFT join. This is correct, since *vendor* is in a common part of the view query, so its information appears both in *ViewBook* and *ViewDVD*. However, t_6 is not in Figure 13, since when the entire view is evaluated, this vendor joins with a book. We call t_6 a *stub*.

Tuples in *relOuterUnion* that are not in *evalRel* are stubs. Stubbed tuples represent starred nodes with an empty evaluation. This situation is denoted by
relOuterUnion(map(split(qt)), d) − evalRel(eval(qt, d)). More precisely:

DEFINITION 6.6 *Let x be an XML instance of a query tree qt with evaluation schema S, and n be a τ_N or τ_T instance node in x. A* stubbed tuple *of x is created from n by associating the value of each leaf node l that is an ancestor of n with the attribute in S corresponding to the name of l, and leaving the value of all other attributes in S null. The set of all stubbed tuples of x is denoted* stubs(x).

The set *stubs(x)* for the XML view of Figure 1 is shown in Figure 17.

THEOREM 6.7 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then every tuple t in* relOuterUnion(map(split(*qt*)), *d*) − evalRel(eval (*qt*, *d*)) $\subseteq$ stubs(x).

Note that the statement of correctness is *not* that the XML view can be constructed from instances of the underlying relational views. The reason is that we do not know whether or not keys of relations along the path from τ_N nodes to the root are preserved, and therefore do not have enough information

	id	vendorname	state	country	price	dtitle	asin
t_1	1	Amazon	WA	US	29	Friends	D1111
t_2	2	Barnes and Noble	NY	US	NULL	NULL	NULL

Figure 16: Tuples on *ViewDVD*

	id	vendorname	state	country	price	btitle	isbn	dtitle	asin
t_1	1	Amazon	WA	US	NULL	NULL	NULL	NULL	NULL
t_2	2	Barnes and Noble	NY	US	NULL	NULL	NULL	NULL	NULL

Figure 17: The *stubs(x)* relation for the XML view x of Figure 1

to group tuples from different relational view instances together to reconstruct the XML view. When keys at all levels *are* preserved, then the query tree can be modified to a form in which the variables iterate over the underlying relational views instead of base tables (see [9]).

6.2 Mapping Updates over XML views to updates over Relational Views

We now discuss how correct updates to an XML view are translated to SQL updates on the corresponding relational views produced in the previous section.

Throughout this section, we will use the XML view of Figure 1, produced by the query tree of Figure 11, as an example. The relational views *ViewBook* and *ViewDVD* corresponding to this XML view were presented in Section 6.1.2.

The translation algorithm for insertions, deletions and modifications, *translateUpdate*, is given in Algorithm 4. What it does is to check the type of update operation and call the corresponding algorithm to translate the update. All the three algorithms (*translateInsert*, *translateDelete* and *translateModify*) assume that the update specification u was already checked for schema conformance (see Section 5.1 for details on how update operations are checked against the view schema).

6.2.1 Insertions

To translate an insert operation on the XML view to the underlying relational views we do the following: First, the unqualified portion of the update path *ref* is used to locate the node in the query tree under which the insertion is to take place. Together with Δ, this will be used to determine which underlying relational views are affected. Second, *ref* is used to query the XML instance and identify the update points. Third, SQL insert statements are generated for each underlying relational view affected using information in Δ as well as information about the labels and values in subtrees rooted along the path from each update point to the root of the XML instance.

Observe that by Proposition 4.6 there is at most one node of type τ_N along the path from any node to the root of the query tree and that insertions can never occur below a τ_N node, since all nodes below a τ_N node are of type τ_S or τ_C by definition.

For example, to translate the insertion of Example 5.1, we use the unqualified update path `/vendors/vendor/products` on the query tree of Figure 11, and find that the type of the update point is $\tau_C(products)$. Continuing from $\tau_C(products)$ using the structure of Δ, we discover that the only τ_N node in Δ is its root, which is of type $\tau_N(book)$. The underlying view affected will therefore be *ViewBook*. We then use the update path *ref*= `/vendors/vendor[@id="01"]/products[@price="38"]` to identify update points in the XML document. In this case, there is one node (8). Therefore, a single SQL insert statement against view *ViewBook* will be generated.

To generate the SQL insert statement, we must find values for all attributes in the view. Some of these attribute-value pairs are found in Δ, and others must be taken from the XML instance by traversing the path from each update point to the root and collecting attribute-value pairs from the leaves of trees rooted along this path. In Example 5.1, Δ specifies *btitle= "New Book"* and *isbn= "9999"*. Along the path from the node 8 to the root in the XML instance of Figure 1, we find *id="01"*,

vendorname="Amazon", *state="WA"*, *country="US"*, and *price="38"*. Combining this information, we generate the following SQL insert statement:

```
INSERT INTO VIEWBOOK (id, vendorname, state, country, price, isbn, btitle)
VALUES ("01","Amazon","WA","US",38,"9999","New Book")
```

As another example, consider the following insertion against the view of Figure 1: t = *insert*, *ref* = `/vendors`,

```
Δ={<vendor id="03">
     <vendorname>New Vendor</vendorname>
     <address>
        <state>PA</state>
        <country>US</country>
     </address>
       <products price="30">
        <book>
         <btitle>Book 1</btitle><isbn>9111</isbn></book>
        <book>
         <btitle>Book 2</btitle><isbn>9222</isbn></book>
        <dvd>
         <dtitle>DVD 1</dtitle><asin>D9333</asin></dvd>
       </products>
    </vendor>}.
```

The unqualified update path *ref* evaluated against the query tree of Figure 11 yields a node τ*(vendors)*, which is the root. Continuing from here using labels in Δ, we discover two nodes of type τ_N: τ_N*(book)* and τ_N*(dvd)*. We will therefore generate SQL insert statements to *ViewBook* as well as *ViewDVD*.

Evaluating *ref* against the XML instance of Figure 1 yields one update point, node 1. Traversing the path from this update point to the root yields no label-value pairs (since the update point is the root itself). We then identify each node of type τ_N in Δ, and generate one insertion for each of them. As an example, traversing the path from the first τ_N(*book*) node in Δ yields label-value pairs *btitle="Book 1"*, and *isbn="9111"*. Going up to the root of Δ, we have *id="03"*, *vendorname="New Vendor"*, *state="PA"*, *country="US"* and *price="30"*. This information is therefore combined to generate the following SQL insert statement:

```
INSERT INTO VIEWBOOK (id, vendorname, state, country, price, isbn, btitle)
VALUES ("03","New Vendor","PA","US",30,"9111","Book 1");
```

In a similar way, information is collected from the remaining two τ_N nodes in Δ to generate:

```
INSERT INTO VIEWBOOK (id, vendorname, state, country, price, isbn, btitle)
VALUES ("03","New Vendor","PA","US",30,"9222","Book 2");
INSERT INTO VIEWDVD (id, vendorname, state, country, price, asin, dtitle)
VALUES ("03","New Vendor","PA","US",30,"D9333","DVD 1");
```

Notice that in the above example, information about vendors is redundantly collected since there are multiple relational views to which the update is mapped. To improve efficiency, in our implementation of the translation algorithm the collected values and values in Δ are cached and reused when specifying the INSERT SQL statements. These cached values may be flushed before processing the next update point.

The algorithm *translateInsert* is presented in the Appendix.

6.2.2 Modifications

By definition, modifications can only occur at leaf nodes. To process a modification, we do the following: First, we use the unqualified *ref* against the query tree to determine which relational views are to be updated. This is done by looking at the first ancestor of the node specified by *ref* which has

type τ_T or τ_N, and finding all nodes of type τ_N in its subtree. (At least one τ_N node must exist, by definition.) If the leaf node that is being modified is of type τ_N itself, then it is guaranteed that the update will be mapped only to the relational view corresponding to this node.

Second, we generate the SQL modify statements. The qualifications in *ref* are combined with the terminal label of *ref* and value specified by Δ to generate an SQL update statement against the view. The corresponding algorithm is presented in the Appendix.

For example, consider the update in Example 5.2. The unqualified *ref* is `/vendors/vendor/vendorname`. The τ_N nodes in the subtree rooted at *vendor* (the first τ_T or τ_N ancestor of *vendorname*) are τ_N(*book*) and τ_N(*dvd*), and we will therefore generate SQL update statements for both *ViewBook* and *ViewDVD*. We then use the qualification *id = "01"* from *ref* = `/vendors/vendor[@id="01"]/vendorname` together with the new value in Δ, to yield the following SQL modify statements:

```
UPDATE VIEWBOOK SET vendorname="Amazon.com" WHERE id="01";
UPDATE VIEWDVD  SET vendorname="Amazon.com" WHERE id="01"
```

6.2.3 Deletions

Deletions are very simple to process. First, the unqualified portion of the update path *ref* is used to locate the node in the query tree at which the deletion is to be performed. This is then used to determine which underlying relational views are affected by finding all τ_N nodes in its subtree. Second, SQL delete statements are generated for each underlying relational view affected using the qualifications in *ref*. The corresponding algorithm is presented in the Appendix.

As an example, consider the deletion in Example 5.3. The unqualified update path expression is `/vendors/vendor`. The τ_N nodes in the subtree indicated by this path in the query tree are τ_N*(book)* and τ_N*(dvd)*. This means that the deletion will be performed in both *ViewBook* and *ViewDVD*. Examining the update path `/vendors/vendor[state="WA"]` yields the label-value pair *state="WA"*. Thus the deletion on the XML view is translated to SQL delete statements as:

```
DELETE FROM VIEWBOOK WHERE state="WA"
DELETE FROM VIEWBOOK WHERE state="WA"
```

It is important to notice that if a tuple t in one relation "owns" a set of tuples in another relation via a foreign key constraint (e.g. a vendor "owns" a set of books), then deletions must cascade in the underlying relational schema in order for the deletion of t specified through the XML view to be allowed by the underlying relational system.

6.2.4 Correctness

Since we are not focusing on how updates over relational views are mapped to the underlying relational database, our notion of correctness of the update mappings is their effect on each relational view *treated as a base table*.

Let x = *eval(qt, d)* be the initial XML instance, u be the update as specified in Definition 5.1, and *apply(x, u)* be the updated XML instance resulting from applying u to x. The function *translateUpdate(x, qt, u)* translates u to a set of SQL update statements $\{U_{11}, ..., U_{1m_1}, ..., U_{n1}, ..., U_{nm_n}\}$, where each U_{ij} is an update on the underlying view instance v_i = *evalV(V_i,d)* generated by *map(split(qt))*.

We use the notation v_i' = *applyR(v_i, $\{U_{i1}, ..., U_{im_i}\}$)* to denote the application of $\{U_{i1}, ..., U_{im_i}\}$ to v_i, resulting in the updated view v_i'. If the set of updates for a given v_i is empty, then $v_i' = v_i$.

THEOREM 6.8 *Given a query tree qt defined over database $\mathcal{D}$, then for any instance d of $\mathcal{D}$ and correct update u over qt,* evalRel(apply(x, u)) $\subseteq v_1' \bigcup ... \bigcup v_n'$, *where* $\bigcup$ *denotes outer union.*

THEOREM 6.9 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then* $v_1' \bigcup ... \bigcup v_n'$ − evalRel(apply(x, u)) $\subseteq$ stubs(apply(x, u)).

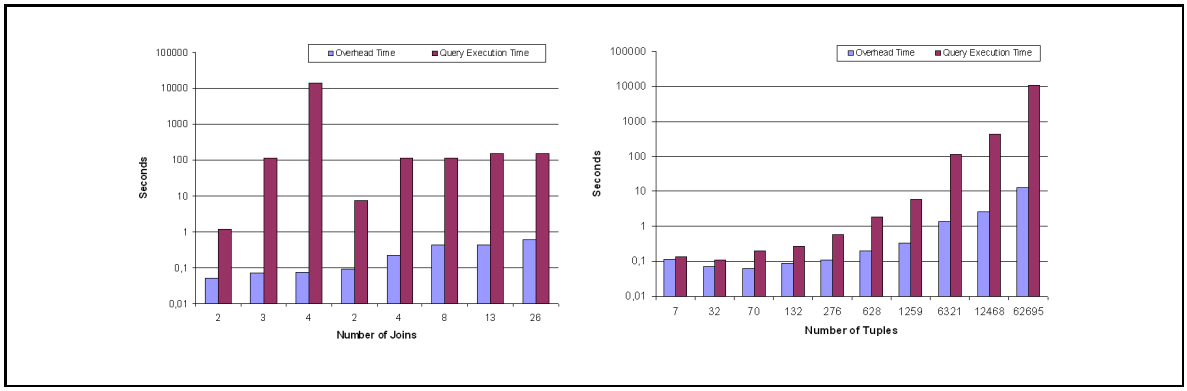


Figure 18: Experimental Results for Mondial (left-hand side) and TPC-H (right-hand side) Databases

Note that a correctness definition like *apply(eval(qt,d), u)* $\equiv$ *eval(qt, d′)*, where d' is the updated relational database state resulting from the application of the translated view updates $\{U_{11}, ..., U_{1m_1}, ..., U_{n1}, ..., U_{nm_n}\}$ to updates on d, does not make sense due to the fact that we do not control the translation of view updates to the underlying relational database. Therefore we cannot claim that they are side-effect free. In [9, 12] we present a scenario where this claim can be made.

7 Experimental Evaluation

In our approach, we have adopted a naive solution for constructing the XML views. We have opted to use an existing XQuery engine (SAXON), and extract the relational tuples in XML format to use as input to SAXON. We have measured the performance of our solution, and, as expected, the results could be improved by leveraging more efficient XML query processing techniques such as those in [29, 44, 43, 18]. In future work, we plan to adapt these ideas in our architecture.

However, since the focus of our work is translating updates to relational updates, the total cost of performing the update is less relevant than the overhead of our solution. The *Overhead Time* of our solution is measured as: time to parse the UXQuery + time to extract the relations as XML files + time to transform the UXQuery into XQuery. We have compared the Overhead Time with the time to execute the corresponding XQuery using SAXON (the *Query Execution Time*) and find that it is not large.

We have evaluated the overhead of our view construction solution on two different databases: Mondial [40] and TPC-H. The evaluation results are shown in Figure 18, where the left-hand side of the figure shows the results for the Mondial Database, and the right-hand side shows the results for the TPC-H database.

In the Mondial database experiment, we varied the number of joins (source tables) in the view from 2 to 26 (shown on the X axis). Some of the views have the same number of joins; for instance, we had 2 views with 2 joins each. What varies, in this case, is the number of tuples in the resulting view. Thus the graph in Figure 18 is ordered according to the number of tuples involved in the view construction. As an example, the first view has 2 joins and 390 tuples, and the last view has 26 joins and 15384 tuples involved in the view construction (the number of tuples is not shown in the graph). The number of tuples was obtained by counting the number of tuples that were extracted by the *XML Extractor*. The Y axis uses logarithmic scale and shows the time to construct the view – the overhead time and the query execution time, as explained above. With this data set, our overhead as a percentage of total execution time was always low, no matter how many joins were in the view. The largest overhead was 4.27% of the total time (first view), and the smallest was 0.0005% of the total time (third view). It is also important to state that the views in Mondial have varying depths.

The TPC-H database was used to measure the data volume our approach is capable of handling. We used the same view definition query in each experiment, varying the number of tuples that were involved in each view. The view contains data about Orders and LineItems, and we varied the number of orders in each view. The first view has Orders with OrderKey = 1 (that is, it has a single order and its line items). The second view has Orders with OrderKey < 10. We increased this progressively to 50, 100, 200, 500, 1000, 5000, 10000 and 50000; after this point, the performance became unnaceptable. For the TPC-H graph, we show on the X axis the number of tuples involved in each view, and on the Y axis, the time in seconds (logarithmic scale). Our tests with the TPC-H views show that when very few tuples are involved the overhead as a percentage of total cost is large. For instance, in the first view, we take 0.13s executing the corresponding XQuery, and the overhead of our solution takes an additional 0.11s. The total execution time in this case is 0.24s, and the overhead of our solution represents 46.43% of this time. However, since the times are small, this overhead is not a performance problem. When the number of tuples involved increases, and consequently the query execution time increases, the overhead as a percentage of total cost diminishes drastically. In the last view (the one with 62695 tuples), our overhead represents only 0.11% of the total execution time.

These results show that the overhead of our solution is low, but that the overall performance of the system could be improved by using an XQuery engine that takes advantage of the underlying relational DBMS engine. We plan to address this issue in future work.

8 Remarks and Future Work

In this paper, we have presented a solution to the problem of updates through XML views over relational databases. The proposed solution takes advantage of existing work on updates through relational views. The XML views are constructed using UXQuery, which allow nesting as well as heterogeneous sets of tuples, and can be used to capture most of the features we encountered in real views.

One of the main contributions of this paper are algorithms to map XML views to a set of underlying relational views, and to map updates on an XML view instance to a set of updates on the underlying relational views. By providing these mappings, the XML update problem is reduced to the relational view update problem and existing techniques on updates through relational views [24, 35, 3, 38] can be leveraged. As an example, in [9] we show how to use the approach of [24] to produce side-effect free updates on the underlying relational database.

Another benefit of our approach is that query trees are agnostic with respect to a query language. Query trees represent an intermediate query form, and any (subset of an) XML query language that can be mapped to this form could be used as the top level language. In particular, we have implemented our approach in a system called PATAXÓ that uses a subset of XQuery to build the XML views and translates XQuery expressions into query trees as an intermediate representation.

Similarly, our update language represents an intermediate form that could be mapped into from a number of high-level XML update languages. In our implementation, we use a graphical user interface which allows users to click on the update point or (in the case of a set oriented update) specify the path in a separate window and see what portions of the tree are affected.

The contributions of this paper are:

A notion of query trees which supports grouping of tuples We add a new type of node (τ_G) into the query trees of [12] to group tuples that agree on a given value. Query trees can be used as an intermediate representation of a top-level query language, making our approach syntax independent. Any language that can be mapped to query trees can be used to specify the XML views. In [9], we evaluate the expressive power of query trees and show that query trees are expressive enough to be used in practice.

Mapping from XML views to relational views Given an XML view specified by a query tree, we provide algorithms to map it to a set of corresponding relational view expressions. We also

provide algorithms to translate updates over the XML view to updates over the corresponding relational views. We thus transform an open problem – that of updating relational databases through XML views – into an existing problem – that of updating relational databases through relational views.

A subset of XQuery to specify XML views over relational databases We have proposed and implemented a subset of XQuery which is capable of constructing XML views over relational databases [11]. UXQuery uses query trees as an intermediate representation to map the resulting XML view to relational views.

PATAXÓ We have implemented our ideas in the PATAXÓ system to show the feasibility of our approach. PATAXÓ uses UXQuery as the view definition language and the approach of [24] to translate updates from the relational views to the underlying relational database.

In this paper, we do not deal with the correctness of the translation of updates to the underlying relational database, since we do not control how they are performed. In our implementation, we use the algorithms of [24] to translate updates to the relational views to the relational database; updates which can potentially cause side-effects are rejected, thus the definition of correctness is "side-effect free". In [12], we use these algorithms to present an updatability study of query trees without *group* nodes. The extensions of query trees we make in this paper, however, require some changes to the updatability study as shown in [9]. As an example of the side-effects that may be caused by group nodes, suppose we specify a modification over the view in Figure 1 by *ref* = `/vendor/vendor[@id=‘‘1"]/products[@price=‘‘38"]/@price` and $\Delta = \{29\}$. The evaluation of *ref* yields node 9. Although it seems fine to modify the value of this node, the reconstructed XML view would collapse the subtree rooted at node 13 with the subtree rooted at node 8. This happens because we are changing the value of node 9 to a value that was already in the view, and the semantics of GROUP requires that nodes that agree in the value of price should be collected together. As a consequence, the XML view modified by the user will be different from the reconstructed view – a side-effect.

As another example, consider a deletion over the view in Figure 1 with update path *ref* = `/vendor/vendor[@id=‘‘1"]/products[@price=‘‘38"]/book`, which evaluates to node 10. The deletion of this book will also make the subtree rooted at node 8 (*products*) to disappear. This is because node 10 was the only book being sold by this price under this vendor.

Our implementation of PATAXÓ prevents these situations by adding two more restrictions on correct updates (Definition 5.4). The first one prohibits modifications on group nodes, and the second one prohibits deletions of starred children of τ_G nodes. Other options for dealing with the problem of updating group nodes include: (1) performing instance analysis to catch exactly those cases that produce side- effects; or (2) allowing side-effects in these special cases, or re-defining side-effects to exclude empty groups or groups which collapse. We leave this for future research.

In future work, we also plan to improve the feedback given to users when an update is rejected. Currently, there is a mismatch between what a user sees (and how he understands the system – XML views), and how updates are being managed (translated to relational views). As an example, suppose the user wants to delete a subtree t in the XML view. This update is translated to a deletion over the corresponding relational view V. Suppose that this update fails because the translation procedure detects that there would be a side-effect. The side-effect was detected using the relational views and its constraints, which the user is not aware of. The question is: how can we explain to the user why the update was rejected? This becomes even more complicated in our scenario, since any translation process of updates through relational views can be used. Consequently, different notions of correctness can be adopted, and different error messages can happen.

Another important issue regards the view generation process. As shown in our evaluation (Section 7), our view generation process is not efficient. We plan to adapt one of the existing proposals in literature [29, 44, 43, 18] to PATAXÓ, so views are constructed more efficiently, taking advantage of the underlying DBMS query engine.

We also plan to extend the language to include other features such as aggregates, and to extend the model to include order.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library. The appendix contains the proofs of the theorems of Section 6 as well as algorithms and the partitioned query trees corresponding to the application of algorithm *split*.

ACKNOWLEDGEMENTS

Research supported in part by CNPq, Capes (BEX 1123-02/5) and FAPERJ Brazil, the Fulbright Program, as well as NSF IIS 0415810. This research was conducted while Vanessa was at UFRGS.

References

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] Apache Software Foundation. Apache Xindice, 2002. 2002. Available at: `<http://xml.apache.org/xindice>`.

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems, TODS*, 6(4):557–575, December 1981.

[4] Chaitan Baru, Amarnath Gupta, Bertram Ludaesher, Richard Marciano, Yannis Papakonstantinou, Velikhov Pavel, and Vincent Chu. Xml-based information mediation with mix. In *SIGMOD*, pages 597–599, Philadelphia, PA, USA, 1999.

[5] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language, September 2005. 2005. Available at: `<www.w3.org>`. W3C Working Draft.

[6] P. Bohannon, S. Ganguly, H.F. Korth, P.P.S. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *VLDB*, Hong Kong, China, August 2002.

[7] Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *ICDE*, San Jose, California, February 2002. IEEE Computer Society.

[8] Vinayak Borkar and Michael Carey. Extending XQuery for grouping, duplicate elimination, and outer joins. In *XML 2004 Conference and Exibition*, pages 1–11, Washington, D.C., U.S.A., November 2004.

[9] Vanessa Braganholo. *From XML to Relational View Updates: applying old solutions to solve a new problem.* PhD thesis, UFRGS, Porto Alegre, RS, Brazil, November 2004.

[10] Vanessa Braganholo, Susan B. Davidson, and Carlos A. Heuser. On the updatability of XML views over relational databases. In *WebDB*, San Diego, CA, June 2003.

[11] Vanessa Braganholo, Susan B. Davidson, and Carlos A. Heuser. UXQuery: building updatable XML views over relational databases. In *Simpósio Brasileiro de Banco de Dados, SBBD*, pages 26–40, Manaus, AM, Brasil, 2003. Belo Horizonte: Departamento de Ciência da Computação/UFMG.

[12] Vanessa Braganholo, Susan B. Davidson, and Carlos A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, Toronto, Canada, September 2004.

[13] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML, January 1999. 1999. Available at: `<www.w3.org>`. W3C Recommendation.

[14] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (third edition), February 2004. 2004. Available at: `<www.w3.org>`. W3C Recommendation.

[15] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330, London, UK, 2001. Springer.

[16] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen W. Luniewski, Wayne Niblack, Dragutin Petkovic, John Thomas, John Williams, and Edward L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach. In *RIDE: Distributed Object Management*, pages 124–131, Taipei, Taiwan, March 1995.

[17] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML query use cases, September 2005. 2005. Available at: `<www.w3.org>`. W3C Working Draft.

[18] Surajit Chaudhuri, Raghav Kaushik, and Jeffrey Naughton. On relational support for XML publishing: Beyond sorting and tagging. In *SIGMOD*, San Diego, CA, June 2003. ACM.

[19] J. Cheng and J. Xu. XML and DB2. In *ICDE*, San Diego, CA, 2000. IEEE Computer Society.

[20] James Clark and Steve DeRose. XML path language (XPath) version 1.0, November 1999. Available at: `<www.w3.org>`. W3C Recomendation.

[21] Andrew Conrad. Interactive microsoft SQL Server & XML online tutorial, 2001. 2001. Available at: `<www.topxml.com/tutorials/main.asp?id=sqlxml>`.

[22] Andrew Conrad. A survey of Microsoft SQL Server 2000 XML features, 2001. MSDN Library. July 2001. Available at: `<http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp>`.

[23] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, San Diego, CA, USA, 2000.

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

[25] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. Minimization and group-by detection for nested XQueries. In *ICDE*, page 839, Boston, USA, March 2004. IEEE Computer Society.

[26] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT framework for logical XQuery optimization. In *VLDB*, pages 168–179, Toronto, Canada, September 2004.

[27] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme SimÉon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics, September 2005. 2005. Available at `<www.w3.org>`. W3C Working Draft.

[28] A. Eisenberg and J. Melton. SQL/XML is making good progress. *SIGMOD Record*, 31(2):101–108, 2002.

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

[30] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[31] Intelligent System Research. Odbc2xml: Merging ODBC data into xml documents, 2001. 2001. Available at: `<www.intsysr.com/odbc2xml.htm>`.

[32] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V.S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[33] Michael Kay. Saxon XSLT and XQuery processor, 2001. 2001. Available at: `<http://sourceforge.net/projects/saxon>`.

[34] Michael Kay. XSL Transformations (XSLT) Version 2.0, September 2005. 2005. Available at: `<www.w3.org>`. W3C Working Draft.

[35] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, Portland, Oregon, March 1985. New York: ACM.

[36] Sasivimol Kittivoravitkul and Peter McBrien. Integrating unnormalised semi-structured data sources. In *CAiSE*, pages 460–474, Porto, Portugal, 2005.

[37] Andreas Laux and Lars Martin. XUpdate WD, September 2000. Sept. 2000. Available at: `<http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html>`. XML:DB Working Draft.

[38] Jens Lechtenbörger. The impact of the constant complement approach towards view updating. In *PODS*, pages 49–55, San Diego, CA, June 2003. ACM.

[39] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 functions and operators, September 2005. 2005. Available at: `<www.w3.org>`. W3C Working Draft.

[40] Wolfgang May. Information extraction and integration with florid: The Mondial case study. Technical Report Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available at: `<http://dbis.informatik.uni-goettingen.de/Mondial/>`.

[41] Ronaldo Santos Mello and Carlos Heuser. BInXS: A process for integration of XML Schemata. In *CAiSE*, pages 151–166, Porto, Portugal, 2005.

[42] Harald Schöning. Tamino – a DBMS designed for XML. In *ICDE*, pages 149–154, Germany, April 2001.

[43] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene Shekita, Catalina Fan, and John Funderburk. Querying XML views of relational data. In *VLDB*, Roma, Italy, September 2001.

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

[45] Software AG. Tamino XML server, 2002. 2002. Available at: `<www.softwareag.com/tamino/details.htm>`.

[46] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *SIGMOD*, Santa Barbara, CA, May 2001. ACM.

[47] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, and E. Shekita. Storing and querying ordered XML using a relational database system. In *SIGMOD*, Madison, Wisconsin, June 2002.

[48] Luiz Tucherman, Antonio L. Furtado, and Marco A. Casanova. A pragmatic approach to structured database design. In *VLDB*, pages 219–231, Florence, Italy, October 1983.

[49] Volker Turau. Db2xml 1.4: Transforming relational databases into XML documents, October 2001. Oct., 2001. Available at: `<www.informatik.fh-wiesbaden.de/~turau/DB2XML>`.

[50] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice Hall, 1997.

[51] Ling Wang, Mukesh Mulchandani, and Elke A. Rundensteiner. Updating XQuery views published over relational data: A round-trip case study. In *XML Database Symposium*, Berlin, Germany, September 2003.

[52] Ling Wang and Elke A. Rundensteiner. On the updatability of XML Views Published over Relational Data. In *ER*, Shanghai, China, November 2004.

A Theorem Proofs

In this section, we present the proofs of theorems 6.5, 6.7, 6.8 and 6.9.

THEOREM 6.5 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then:* evalRel(eval(*qt*, *d*)) $\subseteq$ relOuterUnion(map(split(*qt*)), *d*).

Proof: The $\subseteq$ operation needs the two multi-sets being compared to be union compatible. By definition, the schema of *evalRel* is the *evaluation schema* S, which is composed of all leaf node names in *qt*. The execution of *map(split(qt), d)* results in a set of relational views $\{V_1, ..., V_n\}$. Each view V_i is a schema composed of names of leaf nodes in qt_i (which is produced by *split(qt)*). By definition of *split*, each split tree qt_i contains a single τ_N node n_i: the subtrees rooted at τ_N nodes different from n_i are deleted from qt_i. However, nodes deleted in qt_i are preserved in qt_j, so that each node n in *qt* is in at least one of the $qt_1, ...qt_n$. Consequently, the schema of $V_1 \bigcup ... \bigcup V_n$ equals S.

Assume t is in *evalRel(eval(qt, d))*, but not in *relOuterUnion(map(split(qt)), d)*. Let x be the XML view resulting from *eval(qt, d)*. Since t is in *evalRel(eval(qt, d))*, it was constructed by taking values from the leaf nodes in a given path p. The path p starts in a node n which is the deepest node of type τ_N or τ_T in a given subtree and goes up to the root of x. If n is of type τ_N, and V_i is the view corresponding to n, then t is in *evalV(V_i,d)*, and consequently, t is in *relOuterUnion(map(split(qt)), d)*, a contradiction. If n is of type τ_T, then the node that originated n in the query tree has at least one node of type τ_N in its subtree. Assume $V_j, ..., V_k$ are the relational views corresponding to those τ_N nodes. Consequently, t is in $V_j \bigcup ... \bigcup V_k$, and thus in *relOuterUnion(map(split(qt)), d)*, a contradiction.

THEOREM 6.7 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then every tuple t in* relOuterUnion(map(split(*qt*)), *d*) $-$ evalRel(eval (*qt*, *d*)) $\subseteq$ stubs(*x*).

Proof: Tuples in *relOuterUnion(map(split(qt)), d)* that are not in *evalRel(eval(qt, d))* are those resulting from left outer joins with no match in a given relational view $V_i \in$ *map(split(qt), d)*. Since *stubs(x)* contains tuples that has *null*s in attributes related to descendant nodes, and a LEFT JOIN always keeps information of the ancestor, then:
relOuterUnion(map(split(qt)), d) $-$ evalRel(eval(qt, d)) $\subseteq$ stubs(x).

THEOREM 6.8 *Given a query tree qt defined over database $\mathcal{D}$, then for any instance d of $\mathcal{D}$ and correct update u over qt,* evalRel(apply(x, u)) $\subseteq v'_1 \bigcup ... \bigcup v'_n$, *where* $\bigcup$ *denotes outer union.*

Proof: Since the update u does not change the view schema, and the application of an update U_{ij} over view v_i also does not change v_i's schema, by Theorem 6.5 we have that *evalRel(apply(x, u))* and $v'_1 \bigcup ... \bigcup v'_n$ have the same schema (are union compatible).

Insertions, Suppose t is a tuple in *evalRel(apply(x, u))*, resulting from a insertion of a subtree in x. Assume t is not in $v'_1 \bigcup ... \bigcup v'_n$, and that update U_{ij} is the translation of u.

Consider a tuple t' which was inserted by update U_{ij} in v_i. Since U_{ij} is the translation of u, t' has the values of one of the subtrees that were inserted in x by u, and also the values of x that were above the update point ref of u. As a consequence, $t = t'$ and t is in $v'_1 \bigcup ... \bigcup v'_n$, a contradiction.

The same applies for the insertion of a more complex subtree. It will generate several tuples t_1, ..., t_n to appear in *evalRel(apply(x, u))*. Each of these tuples will be inserted in the relational views by a set of updates $U_{ij}, ..., U_{kl}$. So *evalRel(apply(x, u))* $\subseteq v'_1 \bigcup ... \bigcup v'_n$ holds for insertions.

Modifications. Suppose t is a tuple in *evalRel(apply(x, u))*, resulting from a modification of a leaf value in x. Assume t is not in $v'_1 \bigcup ... \bigcup v'_n$, and that update U_{ij} is the translation of u.

Consider a tuple t' which was modified by update U_{ij} in v_i. Since U_{ij} is the translation of u, t' had a single attribute modified - the one that was updated in x. As a consequence, $t = t'$ and t is in $v'_1 \bigcup ... \bigcup v'_n$, a contradiction.

The same applies for modifications that affect more than one leaf in x, that is, when ref in u evaluates to more than one update point. For every node affected by the modification, will be generated one modification U_{ij}. Since by Theorem 6.5 all tuples in *evalRel(x)* are in $v_1 \bigcup ... \bigcup v_n$, then *evalRel(apply(x, u))* $\subseteq v'_1 \bigcup ... \bigcup v'_n$.

Deletions. Following the inverse reasoning for insertions, every subtree deleted from x makes a tuple disappear from *evalRel(apply(x, u))*s. Analogously, the translation U_{ij} of u will make that tuple disappear from $v'_1 \bigcup ... \bigcup v'_n$, so *evalRel(apply(x, u))* $\subseteq v'_1 \bigcup ... \bigcup v'_n$ holds.

THEOREM 6.9 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then* $v'_1 \bigcup ... \bigcup v'_n$ − evalRel(apply(x, u)) $\subseteq$ stubs(apply(x, u)).

Proof: Insertions of incomplete subtrees or deletions of incomplete subtrees may cause tuples to be filled in with nulls because of the LEFT JOINS in some v'_i. These tuples, however, will be in *stubs*. The reasoning is the same as in proof of Theorem 6.7.

B Algorithms to translate updates

This section presents algorithms to translate insertions, deletions and modifications from the XML view to updates over the corresponding relational views. Such algorithms are mentioned in Section 6.2.

B.1 Insertions

```
translateInsert(V, qt, ref, Δ)
//Inserts Δ in the XML view V using ref as insertion point. Δ must be inserted under every node
  resulting from the evaluation of ref in V. qt is the query tree.
//Assumes that view(n) returns the name of the rel. view associated with node n

Let p be the unqualified portion of ref concatenated with the root of Δ
Let m be the node resulting from the evaluation of p against qt
Let N be the set of nodes resulting from the evaluation of ref in V
for each n in N
   if abstract_type(m) = τ_N
      generateInsertSQL(view(m), root(Δ), n, V)
   else Let X be the set of nodes of abstract type τ_N in Δ
```

```
     for each x in X
       generateInsertSQL(view(x), x, n, V)
     end for
   end if
end for

generateInsertSQL(RelView, r, InsertionPoint, V)
//Inserts the subtree rooted at r into RelView
sql = "INSERT INTO" + RelView + getAttribuelView)
sql = sql + " VALUES ("
for i = 0 to getTotalNumberAttributes(RelView) - 1
   att = getAttribute(RelView, i)
   if att is a child n of r
      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
      sql = sql + ", "
   else sql = sql + ")"
   end if
enf for
```

B.2 Modifications

```
translateModify(V, qt,ref,Δ)

Let p be the unqualified portion of ref
Let m be the node resulting from the evaluation of p against qt
if abstract_type(m) = τ_N
   r = m
else Let r be the ancestor of m whose abstract type is τ_T, τ_G or τ_N
end if
if abstract_type(r) = τ_N
   generateModifySQL(view(r), Δ, ref)
else Let X be the set of nodes with abstract type τ_N under r
   for each x in X
      generateModifySQL(view(x), Δ, ref)
   end for
end if

generateModifySQL(RelView, Δ, ref)

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

B.3 Deletions

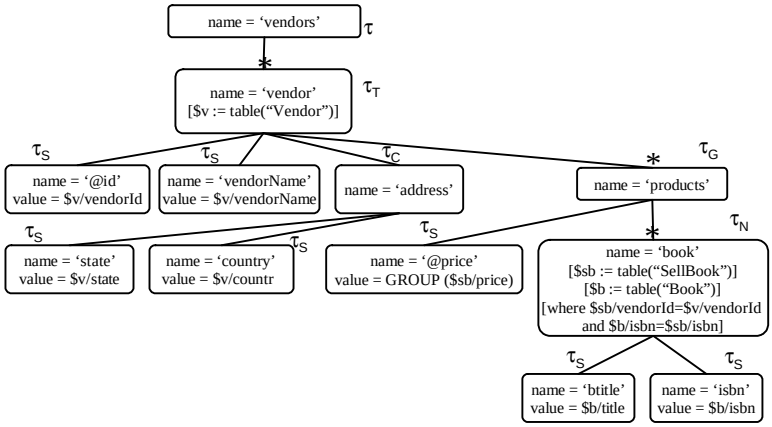
```
translateDelete(V, qt,ref)
//Deletes the subtree rooted at ref from V

Let p be the unqualified portion of ref concatenated with the root of Δ
Let m be the node resulting from the evaluation of p against qt
if abstract_type(m) = τ_N
   generateDeleteSQL(view(m), ref)
else Let X be the set of nodes of abstract type τ_N under m
   for each x in X
      generateDeleteSQL(view(x), ref)
   end for
end if

generateDeleteSQL(RelView, ref)

sql = "DELETE FROM " + RelView
for each filter f in ref
```

```
   if f is the first filter in ref
      sql = sql + + " WHERE " + f
   else sql = sql + " AND " + f
   end if
end for
```

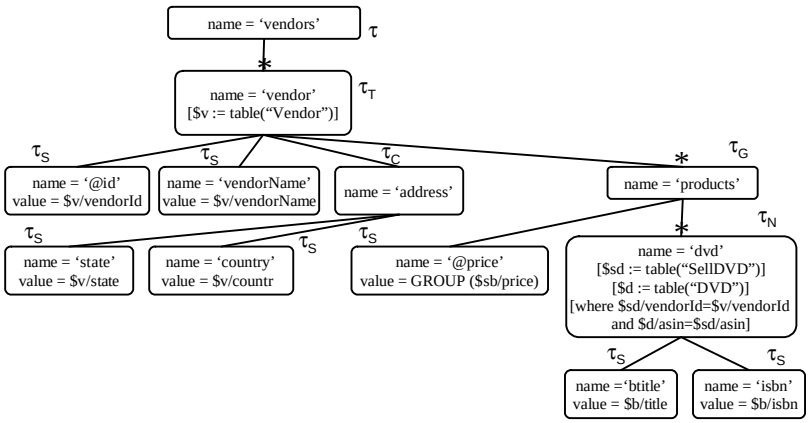
C Partitioned Query Trees

In this section, we present the partitioned query trees corresponding to the application of algorithm *split* to the query tree of Figure 11.

Partitioned query tree for $\tau_N(book)$:



Partitioned query tree for $\tau_N(dvd)$:



```
eval(qt, d)
  evaluate(root(qt, d))

evaluate(n ,d)
  let bindings{} be a hash array of bindings of variable attributes to values, initially empty.
  case abstract_type(n)
     τ | τ_C: buildElement(n)
     τ_T | τ_N: table(n)
     τ_G: group(n)
     τ_S: print "<name(n)>value(n)</name(n)>"
  end case

buildElement(n)
  let tag = "name(n)"
  for each attribute c in children(n)
     add "name(c) = value(c)" to tag;
  print "<tag>"
  for each non-attribute c in children(n)
     eval(c);
  print "</name(n)>"

table(n)
  let w be a list of conditions in sources(n)
  for each w[i]
     if w[i] involves a variable v in bindings{}
        substitute the value binding{v} for v;
  calculate the set B of all bindings for variables in sources(n) that makes the
      conjunction of the modified w[i]'s true
  for each b in B
     add b to bindings{}
     buildElement(n)
     remove b from bindings{}
  end for

group(n)
  let g_1, ..., g_s be the GROUP children of n
  let w be a list of conditions in sources(m), for all starred nodes m that are children of n
  for each w[i]
     if w[i] involves a variable v in bindings{}
        substitute the value binding{v} for v;
  calculate the set B of all bindings for variables in sources(m) (for all starred nodes m
       that are children of n) that makes the conjunction of the modified w[i]'s true
  let V_1 = ⋃_i values of i'th group term in g_1, taken from B
  let ...
  let V_s = ⋃_i values of i'th group term in g_s, taken from B
  for each v_1 in V_1
     add variable bindings x_i/p=value(g_1) for each group variable x_i to bindings{}
     for each v_s in V_s
        add variable bindings x_i/p=value(g_s) for each group variable x_i to bindings{}
        buildElement(n)
        remove variable bindings x_i/p=value(g_s) for each group variable x_i in bindings{}
     end for
     remove variable bindings x_i/p=value(g_1) for each group variable x_i in bindings{}
  end for
```

Alg. 1: Algorithm *eval*

```
map(qt[])
Let sql[] be an empty array of strings; Let numberqt be the number of split trees in qt[]
for k from 1 to numberqt
{ Let n be the node of type τN in qt[k]
   sql[k] = "CREATE VIEW " + name(n) + " AS "; sql[k] = sql[k] + "SELECT "
   Let N be the list of leaf nodes in qt[k]
   for i from 1 to size(N)
   { get next n in N
      if i > 1
         sql[k] = sql[k] + "," + variable(n) + "." + attribute(n) + " AS " + name(n)
      else sql[k] = sql[k] + variable(n) + "." + attribute(n) + " AS " + name(n);
      i = i + 1 }
   sql[k] = sql[k] + " FROM "; Let from = ""; Let N be the set of starred nodes in qt[k]
   for each n in N
   { Let join = "";
      Let S be the list of source annotations in n; Let W be the list of where annotations in n
      for i = 1 to size(S)
      { get next s in S
         join = join + table(s) + " AS " + variable(s)
         if i < size(S)
            join = join + " INNER JOIN ";
         i = i + 1 }
      Let count = 0
      for i = 1 to size(W)
      { get next w in W
         if (w has the form $x/A op $y/B AND $x is bound to table X by a source annotation s ∈ S
             AND $y is bound to table Y by a source annotation s' ∈ S)
            if count = 0
               join = join + " ON " + x.A op y.B
            else join = join + " AND " + x.A op y.B;
            i = i + 1; count = count + 1 }
      if count = 0
         join = join + " ON (1=1) ";
      if size(S) > 1
         join = "(" + join + ")";
      Let A be the set of starred ancestors of n; Let count = 0
      if n has a starred ancestor
         join = " LEFT JOIN " + join
         for i = 1 to size(W)
         { get next w in W
            if (w is of the form $x/A op $y/B AND
              (($x is bound to table X on node n AND $y is bound to table Y on a node a in A)
              OR ($x is bound to table X on a node a in A AND $y is bound to table Y on node n)))
               if count = 0
                  join = join + " ON " + x.A op y.B
               else join = join + " AND " + x.A op y.B;
            i = i + 1; count = count + 1 }
         if count = 0
            join = join + " ON (1=1) ";
         from = "(" + from + join + ")"
      else if abstract_type(n) != τG
               from = from + join ;
   }
   sql[k] = sql[k] + from; Let W' be the set of all where annotations on nodes of qt[k].
   Let count = 0
   for each w' in W'
   { if w' is of the form $x/A op Z AND Z is an atomic value
         if count = 0
            sql[k] = sql[k] + " WHERE " + x.A op Z
         else sql[k] = sql[k] + " AND " + x.A op Z;
   }
}
return sql[]
```

Alg. 2: The *map* algorithm

```
split(qt)
Let t[] be an array of query trees, initially empty
Let i = 0
Let N be the set of nodes of type τ_N in qt
for each node n in N
   inc i
   //initialize t[i] with qt
   t[i] = qt
   repeat
      delete from t[i] all subtrees rooted at a node z of type τ_N, where z ≠ n
      retype the ancestors of the deleted nodes
   until n is the only node of type τ_N in t[i]
   for each group node g in t[i]
      delete from g all the variable references not declared as source annotations
         in its starred sibling
   end for
end for
return t[]
```

Alg. 3: The *split* algorithm

```
translateUpdate(x, qt, u)
case u.t
    insert: translateInsert(x, qt, u.ref, u.Δ)
    delete: translateDelete(x, qt, u.ref)
    modify: translateModify(x, qt, u.ref, u.Δ)
end case
```

Alg. 4: The *translateUpdate* algorithm