From XML view updates to relational view updates: old solutions to a new problem *

Vanessa P. Braganholo[1]
vanessa@inf.ufrgs.br

Susan B. Davidson[2]
susan@cis.upenn.edu

Carlos A. Heuser[1]
heuser@inf.ufrgs.br

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul - UFRGS
Brazil

[2]Department of Computer and Information Science
University of Pennsylvania, USA &
INRIA-FUTURS, France

Abstract

This paper addresses the question of updating relational databases through XML views. Using *query trees* to capture the notions of selection, projection, nesting, grouping, and heterogeneous sets found throughout most XML query languages, we show how XML views expressed using query trees can be mapped to a set of corresponding 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 [15, 23, 22, 8]. 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? Since the problem of updates through relational views has been studied for more than 20 years by the database community, it would be good to use all that work to solve the new problem of updates though XML views. Specifically, is there a way to leverage existing work on updating through relational views to map view updates to the underlying relational database?

In the relational case, attention has focused on updates through 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, we focus on a common form of XML views that allows nesting, composed attributes, heterogeneous sets and repeated elements. An example of such a view is shown in figure 2, which was defined over the database of figure 1. In this XML view, *books* are nested under the *products* node, and the *address* node composes attributes in a nested record format. The *products* node is composed of tuples of two different types, *book* and *dvd*.

We represent XML view expressions as *query trees*. Query trees can be thought of as the intermediate represen-

Vendor(vendorId, vendorName, url, state, country)
Book(isbn, title, publisher, year)
DVD(asin, title, genre, nrDisks)
Sell-Book(vendorId, isbn, price)
– foreign key(vendorId) references Vendor
– foreign key(isbn) references Book
Sell-DVD(vendorId, asin, price)
– foreign key(vendorId) references Vendor
– foreign key(asin) references DVD

Figure 1: Sample database

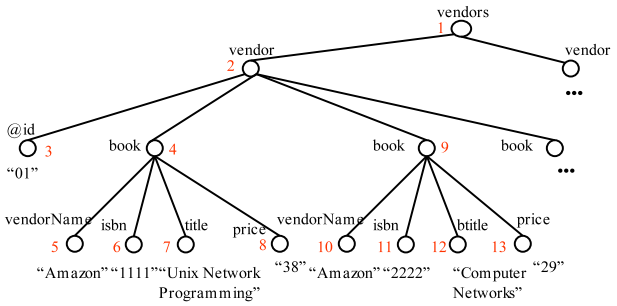

Figure 3: View 1: books and vendors

*Research partially supported by CNPq and Capes (Brazil) as well as NSF DBI-9975206 (USA).

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

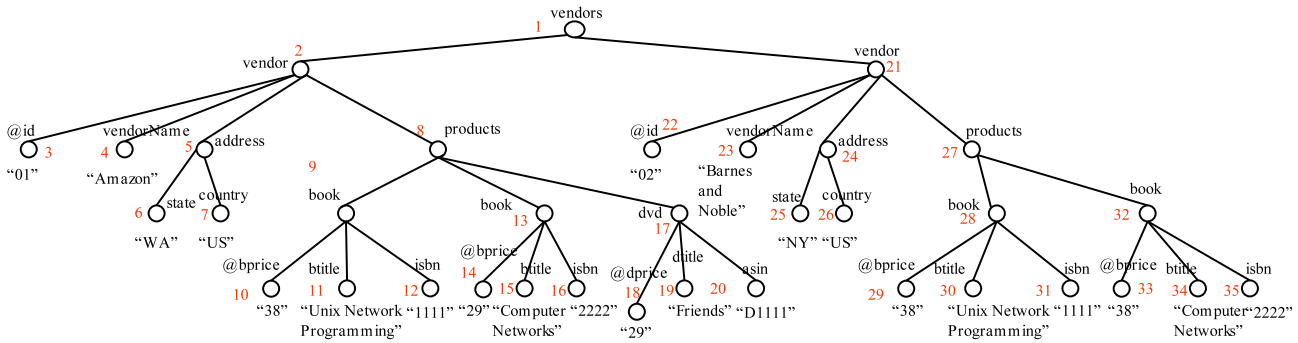

Figure 2: View 2: vendors, books and dvds

tation of a query expressed by some high-level XML query language, and provide a language independent framework in which to study how to map updates to an underlying relational database. They are expressive enough to capture the XML views that we have encountered in practice, yet are simple to understand and manipulate. Their expressive power is equivalent to that of DB2 DAD files [9]. Throughout the paper, we will use the term "XML view" to mean those produced by query trees.

The strategy we adopt is to map an XML view to a set of underlying relational views. 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 updates 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.

This strategy is similar to that adopted in [5] for XML views constructed using the nested relational algebra (NRA), however, our view and update language are far more general. In particular, nested relations cannot handle heterogeneity. Thus, the NRA is capable of representing the XML view of Figure 3 but not that of Figure 2, and maps an XML view to exactly one underlying relational view.

The outline and contributions of this paper are as follows:

- Section 2 defines query trees, their abstract types, and the resulting XML view DTD.
- Section 3 presents the algorithm for mapping an XML view to a set of underlying relational views, and proves its correctness.
- Section 4.1 defines a simple XML update language and algorithms to detect whether or not an update is correct with respect to the XML view DTD.
- Section 4.2 gives an algorithm for mapping insertions, modifications and deletions on XML views to updates on the underlying relational views, and Section 4.3 proves its correctness.
- Section 4.4 illustrates our approach by showing how to use the techniques of [13] to detect if an XML view is updatable with respect to a given update.
- Section 5 discusses the expressive power of our language, and evaluates our technique with respect to existing proposals on extracting XML views of relational databases.

Related work can be found in Section 6. We conclude in Section 7 with a discussion of future work.

2 Query Trees

Query trees are used as a representation of the XML view extraction query. We use this abstract representation rather than an XML query language syntax for several reasons: First, reasoning about updates and the updatability of an XML view is performed at this level. Second, they are easy to understand yet expressive enough to capture several important aspects of XQuery such as nesting, composed attributes, and heterogeneous sets.[1] They can therefore be thought of as the intermediate processing form for a subset of many different XML query languages. For example, we have developed an implementation of our technique which uses a subset of XQuery as the top-level language [6].

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.

2.1 Query Trees Defined

An example of a query tree can be found in Figure 4, which retrieves books that are sold for prices greater than $30. 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. The result of this query is also presented in Figure 4.

Query trees are very similar to the *view forests* of [15] and *schema-tree queries* presented in [3]. 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.

[1] They can also capture grouping, but for simplicity we omit that [7].

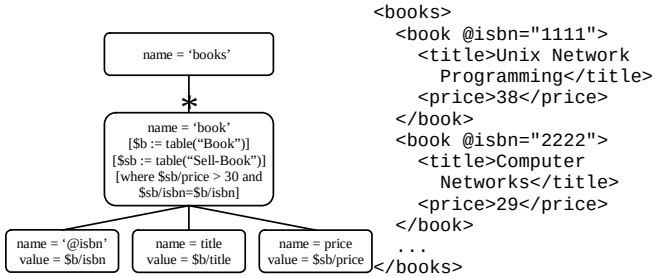

Figure 4: Example of query tree and its resulting XML view

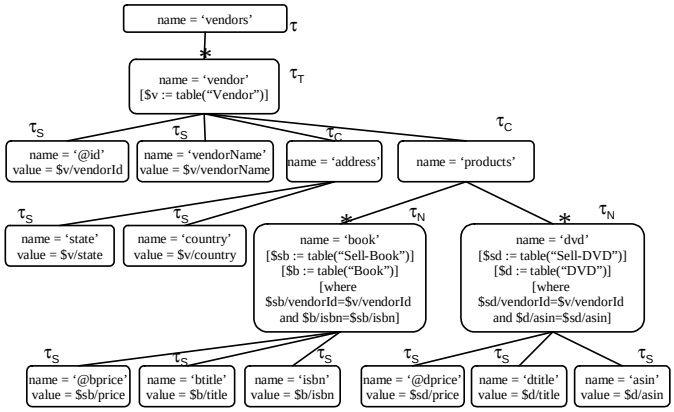

Figure 5: Query tree for View 2

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 2.1 *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, 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 (to be defined). 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 (to be defined).*

Since we map XML views to relational views, nodes with the same name in the query tree may cause ambiguities in the mapping. This problem can easily be solved by associating with each node name a number corresponding to its position in the query tree, and using it internally in the mapping. For simplicity, in this paper we will ignore this problem and use unique names for nodes in the query trees.

Returning to the example in Figure 4, there is a *-edge from the root (named *books*) to its child named *book*, indicating that in the corresponding XML instance there may be several *book* subelements of *books*. There is a simple edge from the node named *book* to the node named *title*, indicating that there is a single *title* subelement of *book*. The node named *@isbn* 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 2.2 *A* source annotation *s within a starred node n is of the form* [$\$x$:= 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 2.3 *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.*

Definition 2.4 *The value of a node n 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.*

In Figure 4, the node *book* has source annotations and where annotations. The source annotations bind variable *$b* to the relational table *Book*, and variable $\$sb$ to the relational table *Sell-Book*. The where annotations restrict the books that appear in the view to those with price greater than $30, and specify the join condition of tables *Book* and *Sell-Book*. The value of the node *@isbn* is specified as *$b/isbn*, indicating that the content of the XML view attribute *isbn* will be generated using attribute *isbn* of the table *Book*.

A more complex example of a query tree can be found in Figure 5 (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 within *products*. The root *vendors* has a set of *vendor* child nodes (*-edge). 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*, and that 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*. Source annotations of the *book* node include bindings for *$b* (Book) and *$sb* (Sell-Book) and its where annotations connect tuples in Sell-Book to tuples in Book, and tuples in Sell-Book with tuples in Vendor (join conditions). Node *dvd* has source annotations for *$d* (DVD) and *$sd* (Sell-DVD). Its where annotation connects tuples in Sell-DVD to tuples in DVD and tuples in Sell-DVD with tuples in Vendor. The result of this query tree is View 2, shown in Figure 2.

From now on, we assume that a query tree is *non-empty*, i.e. that it consists of more than a root node.

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 five abstract types of nodes: τ, τ_T, τ_N, τ_C and τ_S. 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 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. 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 5, 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.

We call the XML views produced by query trees and their associated abstract types *well-behaved* because, as we will show in the next section, they can be easily mapped to a set of corresponding relational views. However, before turning to the mapping we prove two facts about query trees that will be used throughout the paper.

Proposition 2.1 *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 2.2 *There is at most one τ_N node along any path from a leaf 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 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 5 is referred to as *τ_N(dvd)*, and its type (DTD) is *<!ELEMENT dvd (dtitle, asin)>*.

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

```
eval(qt, d) {qt is the root of the query tree, d the database instance}
{qt is the query tree and d is the database instance}
evaluate(root(qt), d)

evaluate(n ,d)
{Assume a node type has functions abstract_type(n), name(n), value(n),
children(n), sources(n), and where(n) (with the obvious meanings).}
Let bindings{} be a hash array of bindings of variable attributes to values, initially
empty.
case abstract_type(n)
   τ|τC: buildElement(n)
   τT|τN: table(n)
   τS: print "<name(n)>value(n)</name(n)>"
end case

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

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

Algorithm 1: Eval algorithm

constructs the XML view resulting from a query tree 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 stores the current 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.

2.4 DTD of a Query Tree

Query tree views defined over a relational database have a well-defined schema (DTD) 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, E_{k+1}*, ..., E_n*)>, where $E_1, ..., E_k$ are non-attribute child nodes

of E connected by a simple edge, and E_{k+1}*, ..., E_n* are child nodes of E connected by a *-edge. In case $n = 0$, then create an element declaration <!ELEMENT E EMPTY>

As an example, the DTD of the view produced by the query tree shown in Figure 5 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*)>
<!ELEMENT book (btitle, isbn)>
<!ATTLIST book bprice CDATA #REQUIRED>
<!ELEMENT btitle (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT dvd (dtitle, asin)>
<!ATTLIST dvd dprice CDATA #REQUIRED>
<!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.

3 Mapping to Relational Views

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 [13, 17, 20, 2, 27] can then be used to accept, reject or modify the proposed SQL updates. In this section, we discuss how an XML view constructed by a query tree is mapped to a set of corresponding relational view expressions.

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 nodes 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. Similar to 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.

For example, the relational view corresponding to the query tree in Figure 4 is:

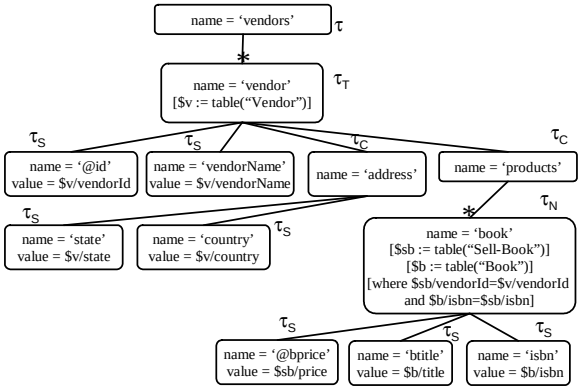

Figure 6: Partitioned query tree for $\tau_N(book)$

```
SELECT b.isbn AS isbn, b.title AS title, sb.price AS price
FROM (Book AS b INNER JOIN Sell-Book AS sb
      ON sb.isbn=b.isbn) WHERE sb.price > 30
```

Split. For a query tree with more than one τ_N node, this process is incorrect. As an example, consider the query tree of Figure 5 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 query tree. We must therefore split a query tree into sub-query trees containing exactly one τ_N node each before generating the corresponding relational views. After the splitting process, each sub-query tree produced is mapped to a relational view 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 2.1.

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, 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 (algorithms *map* and *split* are available in [7]).

The result of this process for the query tree of Figure 5 is shown in Figures 6 and 7. Using these split trees, the corresponding relational views *ViewBook* and *ViewDVD* are (we name these views so we can refer to them in the examples of Section 4):

```
CREATE VIEW VIEWBOOK AS
SELECT v.vendorId AS id, v.vendorName AS vendorName,
 v.state AS state, v.country AS country,
 sb.price AS bprice, b.isbn AS isbn, b.title AS btitle
FROM (Vendor AS v LEFT JOIN (Sell-Book 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 dprice, d.asin AS asin, d.title AS dtitle
FROM (Vendor AS v LEFT JOIN (Sell-DVD AS sd INNER JOIN
DVD AS d ON d.asin=sd.asin) ON v.vendorId=sd.vendorId)
```

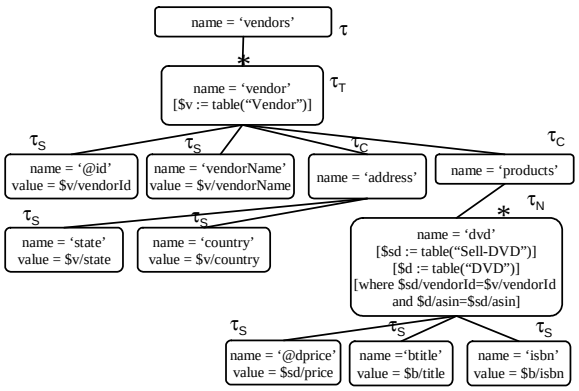

Figure 7: Partitioned query tree for $\tau_N(dvd)$

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

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 for the XML view is in one or more instances of the corresponding relational views. To be more precise, we define the following:

Definition 3.1 *The* evaluation schema S *of a query tree qt is the set of all names of leaf nodes in qt.*

Definition 3.2 *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 be the deepest τ_N or τ_T instance nodes for some root to leaf path in x. Let p be the set of nodes in the path from n to the root of x. An* evaluation tuple *of x is created from n by associating the value of each leaf node l that is a descendant of n or of some node in p 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 of all evaluation tuples of x is called its evaluation relation *and is denoted* evalRel(x).

For example, Table 1 shows the result of *evalRel(x)* for the query tree of Figure 5.

Definition 3.3 *Let $\{V_1, ..., V_n\}$ be defined over a relational schema $\mathcal{D}$, and d be an instance of $\mathcal{D}$. Then* relOuterUnion($\{V_1, ..., V_n\}, d$) *denotes the set of relational instances that result from taking the outer 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, and* evalV(V,d) *instantiates V over d.*

For example, *relOuterUnion({ViewBook, ViewDVD}, d)* is the outer union of *evalV(ViewBook, d)* and *evalV(ViewDVD, d)*, whose result is shown in Table 2.

The correctness of the set of relational views resulting from *map* and *split* can now be understood in the following sense:

Theorem 3.1 *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).

(Proofs for all the theorems of this papers are available in [7].)

Furthermore, the tuples in *relOuterUnion(map(split(qt)), d) − evalRel(eval(qt, d))* represent starred nodes with an empty evaluation (which we call "stubbed" nodes). More precisely:

Definition 3.4 *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).

As an illustration of a stubbed tuple, consider tuple t_6 in table 2. Since the XML instance of Figure 2 does not have any dvd sold by vendor *Barnes and Noble*, 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, so its information appears both in *ViewBook* and *ViewDVD*. However, t_6 is not in table 1, since when the entire view is evaluated, this vendor joins with a book.

Theorem 3.2 *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 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, and used to reconstruct the XML view. Details of this algorithm (*replace*) can be found in [7].

4 Updates

Given an update against a well-behaved view, we translate it to a set of SQL update statements against the corresponding relational view expressions, so existing work on updates through relational views can be used to translate the updates to the underlying relational database. In this section, we start by defining XML updates and then describe the translation. We also summarize how to determine whether or not an update is side-effect free.

Although no standard has been established for an XML update language, several proposals have appeared [1, 25, 4,

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

Table 1: Tuples resulting from *evalRel(eval(qt, d))* for the query tree of Figure 5

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

Table 2: Tuples resulting from *relOuterUnion({ViewBook,ViewDVD}, d)*

19]. 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 [4]). The simplicity of the language allows us to focus on the key problem we are addressing.

4.1 Update language

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 4.1 *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 [10] which indicates where the update is to occur.*

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 child 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 4.2 *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 5, since the 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 2:

Example 4.1 *To insert a new book selling for $38 under the vendor with id="01" we specify:* t = insert, ref = /vendors/vendor[@id="01"]/ products,

```
Δ = {<book bprice = "38">
        <btitle>New Book</btitle><isbn>9999</isbn>
     </book>}.
```

Example 4.2 *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 4.3 *To delete all books with title "Computer Networks" we specify:* t= delete, ref = /vendors/vendor/products/book[btitle="Computer Networks"].

Note that not all insertions and deletions make sense since the resulting XML view may not conform to the DTD of the query tree (see Section 2.4). For example, the deletion specified by the path */vendors/vendor/vendorName* would not conform to the DTD of Figure 5 since *vendorName* is a required subelement of *vendor*. We must also check that Δ's inserted and subtrees deleted are correct.

Definition 4.3 *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;*
- *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 (deletion), then the unqualified portion of* ref + *the root of Δ (*ref*) evaluated on qt arrives at a node whose incoming edge is starred (equivalently, its abstract type is τ_T or τ_N);*
- *if nonempty, then Δ conforms to the DTD of the element arrived at by* ref.

For example, the deletion of example 4.3 is correct since *book* is a starred subelement of *products*. However, the deletion specified by the update path */vendors/vendor/vendorName* is not correct since *vendorName* is of abstract type τ_S, as is the deletion specified by the invalid update path */vendors/vendor/dvd*.

4.2 Mapping XML updates to 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 2 of Figure 2 as an example. The relational views *ViewBook* and

ViewDVD corresponding to this XML view were presented in Section 3.

The translation algorithm for insertions, deletions and modifications, *translateUpdate*, is given in [7].

4.2.1 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 2.2, 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 4.1, we use the unqualified update path */vendors/vendor/products* on the query tree of Figure 5, 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* 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 4.1, Δ specifies *bprice="38"*, *btitle="New Book"* and *isbn="9999"*. Along the path from the node 8 to the root in the XML instance of Figure 2, we find *id="01"*, *vendorName="Amazon"*, *state="WA"* and *country="US"*. Combining this information, we generate the following SQL insert statement:

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

As another example, consider the following insertion against the view 2: t = *insert*, *ref = /vendors*,

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

The unqualified update path *ref* evaluated against the query tree of Figure 5 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* and as well as *ViewDVD*.

Evaluating *ref* against the XML instance of Figure 2 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 $\tau_N(book)$ node in Δ yields label-value pairs *bprice = "30"*, *btitle = "Book 1"*, and *isbn = "9111"*. Going up to the root of Δ, we have *id = "03"*, *vendorName = "New Vendor"*, *state = "PA"* and *country = "US"*. This information is therefore combined to generate the following SQL insert statement:

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

4.2.2 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.

For example, consider the update in example 4.2. 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 $\tau_N(book)$ and $\tau_N(dvd)$, 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"
```

4.2.3 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*.

As an example, consider the deletion in example 4.3. The unqualified update path is */vendors/vendor/products/book*. The only τ_N node in the subtree indicated by this path in the query tree is τ_N*(book)*. This means that the deletion will be performed in *ViewBook*. Examining the update path */vendors/vendor/products/book[btitle="Computer Networks"]* yields the label-value pair *btitle="Computer Networks"*. Thus the deletion on the XML view is translated to an SQL delete statement as:

```
DELETE FROM VIEWBOOK WHERE btitle="Computer Networks"
```

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.

4.3 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 4.1, and *apply(*x*,* u*)* be the updated XML instance resulting from applying u to x. The function *translateUpdate(*x*,* qt*,* u*)* (shown in [7] and summarized in Section 4.2) 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 4.1 *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 4.2 *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))

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. Therefore we cannot claim that they are side-effect free.

In the next subsection, we discuss a scenario in which this claim can be made.

4.4 Updatability

There are several choices of techniques that could be used to translate from updates on relational views to updates on the underlying relational database. Some consider a translation to be correct if it does not affect any part of the database that is outside the view [2, 20]. Others consider a translation to be correct as long as it corresponds exactly to the specified update, and does not affect anything else in the view [13]. Still others use additional information to build specific translators for each view [18, 21, 27]. Here, we choose [13] to illustrate how reasoning about *side-effect free* relational view updates can be extended to XML views.

In [5], we define conditions under which XML views constructed by "nest-last" nested relational algebra (NRA) expressions are updatable. Since nest-last NRA expressions perform nests over a relational algebra expression, our results are based on the ability to unnest the NRA expression to obtain a (single) corresponding relational view, and then build on the results of [13] to detect updatability. Since query trees also express nesting and are mapped to a *set* of corresponding relational views, we can use these results to reason about the updatability of XML views constructed by query trees. We assume the underlying relational database is in BCNF (as required by [13]), and impose three restrictions on the query tree and update: (1) each table must be bound to at most one variable; (2) each value in a leaf node must be unique, that is, if the value of n is specified as $\$x/A$, then this value specification does not appear on any other node in the query tree; (3) comparisons in the filters of *ref* must be equalities. These restrictions are imposed so that the resulting relational views do not include joins of the same tables and projections of the same attribute (as required by [13]). The restriction to equalities in conditions is also required by [13].

Theorem 4.3 *A correct update* u *to an XML view defined by a query tree* qt *is side-effect free if for all (*U_i*,* V_i*), where* V_i *is the corresponding relational view of* qt_i *and* U_i *is the translation of* u *over* V_i*,* U_i *is side-effect free in* V_i*.*

Based on Theorem 4.3, we can now answer a more general question: Is there a class of query tree views for which all possible updates are side-effect free? To answer this question, we summarize the results of [5] and [13] for conditions under which NRA views are updatable, and generalize them for XML views constructed by query trees.

Insertions. An insertion over an NRA view is side-effect free when the corresponding relational view V is a select-project-join view, the primary and foreign keys of the source relations of V are in the view and joins are made only through foreign keys. In terms of query trees, this means that the primary keys of the source relations of qt_i must appear as values in leaf nodes of qt_i and the *where* annotations in qt_i specifies joins using foreign keys, for all split trees qt_i corresponding to a query tree qt.

Deletions and modifications. Deletions and modifications over an NRA view V are side-effect free when the above conditions for insertions are met and V is *well-nested* [5]. By *well-nested*, we mean that the source relations in V must be nested according to key-foreign key constraints in the underlying relations. We rephrase this condition in terms of query trees as follows:

Definition 4.4 *A query tree qt is* well-nested *if for any two source relations R and S in qt, if S is related to R by a foreign key constraint then the source annotation for R occurs in an ancestor of the node s containing the source annotation for S. Additionally, attributes of R must not appear as values in the descendants of s.*

The results above identify three classes of updatable XML views: one that is updatable for all possible insertions; one that is updatable for all possible insertions, deletions and modifications; and a general one whose updatability with respect to a given update can be reasoned about using Theorem 4.3. Furthermore, we can now prove the following:

Theorem 4.4 *Given a query tree qt with the restrictions mentioned above and defined over a BCNF database $\mathcal{D}$, then for any instance d of $\mathcal{D}$ and correct update u over qt:* 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 $\{U_{11}, ..., U_{1m_1}, ..., U_{n1}, ..., U_{nm_n}\}$ using the techniques of [13].*

We leave the study of updatability using other existing relational techniques for future work.

5 Evaluation

For purposes of presentation, the query tree language presented in this paper was kept simple to highlight how the mapping of the query tree and updates are performed.

Query trees can be extended in a number of ways, for example to deal with grouping, aggregates, function applications and so on. As an example of such extension, in [7] we allow *grouped values* which allow tuples that agree on a given value to be clustered together, as well as leaf nodes with attributes. With such an extension, *book*s and *dvd*s that agree on a given price could be grouped under a common *products* ancestor. In this case, the node *products* would be a starred node with a child *@price*. The node *products* would repeat for every distinct value of *price* on tables Sell-Book and Sell-DVD. This extension affects the mapping algorithm only superficially and does not affect the results of this paper.

However, another consideration that must be kept in mind when extending the language is whether or not the relational views resulting from the XML view are updatable. The language presented in this paper, with suitable restrictions on the way in which joins and nesting are performed with respect to keys and foreign keys in the underlying relational database, presents a subset of XQuery in which *side-effect free* updates can be defined as discussed in the previous section. While grouped values and leaf nodes with attributes do not affect these results, the addition of functions and aggregates would. Analogous to work on updating views in relational databases which restricts views to select-project-join queries, we have therefore initially decided against considering a richer language (although we plan to do so in future work).

The EBNF for the subset of XQuery corresponding to our language (with grouped values) can be found in [7].

To evaluate our language, we first discuss the restrictions in our form of queries, and what query trees can or cannot express. Second, we examine the power of expression of query trees, and compare it with existing proposals in literature. We have also analyzed the "practicality" of XML views constructed by query trees by collecting examples of real XML views extracted from relational databases and evaluating whether or not query trees can capture them. For these real XML views, query trees were sufficiently expressive. Details can be found in [7].

5.1 Limitations of Query Trees

Although query trees are quite expressive, there are some restrictions.

Values must come from the relational database. We do not allow constants to be introduced as values in leaves, nor do we allow functions to calculate new values from values in the database. Allowing constant values in leaves is potentially useful (for example, to add a version number to the view), but they are not interesting from the perspective of updates to the relational database nor can they themselves be updated since they are not part of the database schema. Calculating a value from a set of values (e.g. taking the average of a relational column) creates a one to many mapping which cannot be updated; research on relational views also disallows this case. However, calculating a new value from a single value in the database (e.g. translating length in centimeters to length in inches) could be allowed as long the reverse function was also specified.

Queries are trees rather than graphs. This restriction disallows recursive queries, which are also disallowed in SilkRoute [15]. For example, suppose the relational database contained a relation Patriarchs(PName, CName) with instance {(John, Marc), (John, Chris), (Justin, John)}. An XML view of this that one might wish to construct would be:

```
<Patriarch>
 <Name>Justin</Name>
 <Children>
    <Name>John</Name>
    <Children> <Name>Marc</Name>
               <Name>Chris</Name>
    </Children>
 </Children>
</Patriarch>
```

Since recursive queries cannot be mapped to select-project-join queries, our technique would have to be extended significantly to reason about them.

On the other hand, query trees are flexible enough to represent heterogeneous structures (e.g. the view in Figure 5).

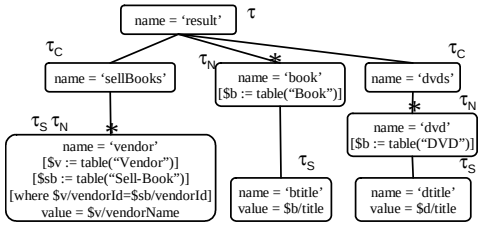

Figure 8: Example of query tree

It can also represent query trees with a repeating leaf node, as shown in Figure 8 (note that *vendor* is labeled with τ_N and τ_S). The XML view resulting from this query tree is as follows:

```
<result>
  <sellBooks>
    <vendor>Amazon</vendor>
    <vendor>Barnes and Nobel</vendor>
  </sellBooks>
  <book><btitle>Unix Network Programming</btitle></book>
  <book><btitle>Computer Networks</btitle></book>
  ...
  <dvds>
    <dvd><dtitle>Friends</dtitle></dvd>
    ...
  </dvds>
</result>
```

It turns out that XML views with heterogeneous content and repeating leaves arise frequently in practice, but that recursive views are not common. We therefore believe that the above restrictions do not limit the usefulness of our approach.

5.2 Power of Expression

We now compare the expressive power of query trees with SilkRoute's *view forests* [15], XPERANTO [22], and DB2 DAD files [9].

XPERANTO [22] can express all queries in XQuery. View forests [15] are capable of expressing any query in the XQueryCore that does not refer to element order, use recursive functions or use is/is not operators. Query trees present the same limitations as [15], and are also not capable of expressing *if/then/else* expressions; sequences of expressions (since we require that the result of the query always be an XML document); function applications; and arithmetic and set operations. Input functions are also a limitation of query trees; in contrast to SilkRoute, variables cannot be bound to the results of expressions.

DB2 XML Extender provides mappings from relations to XML through DAD files. Mappings can be done in two ways: using a single SQL statement (by using the `SQL_stmt` element in the DAD file), or using the RBD_node mapping. The SQL_stmt method allows only a single SQL statement, so XML views with heterogeneous structures (like the one in Figure 5) can not be constructed. The RBD_node method allows heterogeneous structures, since instead of specifying a single SQL statement for the XML extraction, the user specifies, for each XML element or attribute in the XML view, the table and attribute name from which the data must be retrieved. It is also possible to specify conditions for each XML node in the DAD file (join conditions and selection conditions). DB2 DAD files with RDB_node method are equivalent to query trees in expressive power, since all the data come directly from the relational database and functions cannot be applied over the retrieved data. This is meaningful, since DB2 DAD files represent features that are useful in practice, and because this subset can easily be mapped to relational views.

6 Related Work

There are several proposals for exporting and querying XML views of relational databases [8, 15, 22, 23]. For updates, [28] presents a round trip case study, where XML documents are stored in relational databases, reconstructed and then updated. In this case, it is always possible to translate the updates back to the underlying relational database. Our approach differs since we address updates of *legacy* databases through XML views.

Commercial relational databases offer support for extracting XML data from relations as well as restricted types of updates. In SQL Server [11], an XML view generated by an annotated XML Schema can be modified using *updategrams*. To update, the user provides a before and after image of the XML view [12]. The system computes the difference between the images and generates SQL update statements. The views supported by this approach are very restricted: joins are through keys and foreign keys, and nesting is controlled to avoid redundancy. This corresponds to our well-nested query trees, which are therefore provably updatable with respect to all insertions, deletions and modifications. Oracle [14] offers the specification of an annotated XML Schema, but the only possible update is to insert an XML document that agrees with the schema. IBM DB2 XML Extender [9] requires that updates be issued directly in the relational tables.

Native XML databases also support updates [26, 16, 24]. The goal of all these systems differs from ours since they do not update through views.

7 Conclusions

In this paper, we present a technique for updating relational databases through XML views. The views are constructed using query trees, which allow nesting as well as heterogeneous sets of tuples, and can be used to capture mixed content, grouping, as well as repeating text elements and text elements with attributes.

The main contributions of this paper are the mapping of the XML view to a set of underlying relational views, and the mapping of 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 views [13, 17, 2, 20] can be leveraged. As an example, we show how to use the approach of [13] 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 rep-

resent 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 [6]. Similarly, our update language represents an intermediate form that could be mapped into from a number of high-level XML update languages (using a static evaluation of which updates are to be performed). 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.

In future work, we plan to study the updatability of XML views using other proposals of updates through relational views in the literature. We also plan to extend the language to include other features such as aggregates, and to extend the model to include order.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4), Dec. 1981.

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

[4] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *ICDE*, San Jose, California, Feb. 2002.

[5] V. Braganholo, S. Davidson, and C. Heuser. On the updatability of XML views over relational databases. In *Proceedings of WEBDB 2003*, San Diego, CA, June 2003.

[6] V. Braganholo, S. Davidson, and C. Heuser. UXQuery: building updatable XML views over relational databases. In *Brazilian Symposium on Databases*, pages 26–40, Manaus, AM, Brazil, 2003.

[7] V. Braganholo, S. Davidson, and C. Heuser. Propagating XML View Updates to a Relational Database. Technical Report TR-341, UFRGS, Porto Alegre, RS, Brazil, Feb. 2004.

[8] S. Chaudhuri, R. Kaushik, and J. Naughton. On relational support for XML publishing: Beyond sorting and tagging. In *Proceedings of SIGMOD 2003*, San Diego, CA, June 2003.

[9] J. Cheng and J. Xu. XML and DB2. In *Proceedings of ICDE'00*, San Diego, CA, 2000.

[10] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recomendation, Nov. 1999.

[11] A. Conrad. A Survey of Microsoft SQL Server 2000 XML Features. MSDN Library. *http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp*. Jul 2001.

[12] A. Conrad. Interactive microsoft SQL Server & XML online tutorial. *http://www.topxml.com/tutorials/main.asp?id=sqlxml*.

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

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

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

[16] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[17] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of SIGMOD*, pages 154–163, Portland, Oregon, Mar. 1985. ACM.

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

[19] A. Laux and L. Martin. XUpdate WD, Sept. 2000. Working Draft. *http://www.xmldb.org/xupdate/xupdate-wd.html*.

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

[21] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in RIGEL. In *SIGMOD*, pages 71–81, Boston, Massachusetts, 1979.

[22] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of VLDB 2001*, Roma, Italy, Sept. 2001.

[23] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *The VLDB Journal*, pages 65–76, 2000.

[24] Software AG. Tamino XML Server, 2002. *http://www.softwareag.com/tamino/details.htm*.

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

[26] The Apache Software Foundation. Apache Xindice. *http://xml.apache.org/xindice*, 2002.

[27] L. Tucherman, A. L. Furtado, and M. A. Casanova. A pragmatic approach to structured database design. In *VLDB*, pages 219–231, Florence, Italy, Oct. 1983.

[28] L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *Proc. of XML Database Symposium*, Berlin, Germany, Sept. 2003.