Updating relations through XML views

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

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15064 – 90501-970 Porto Alegre, RS

[2]Department of Computing and Sciences – University of Pennsylvania
19104-6389 Philadelphia, PA

vanessa@cos.ufrj.br, susan@cis.upenn.edu, heuser@inf.ufrgs.br

Abstract. *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 present a solution to the XML update problem which uses query trees as the view definition language, and maps XML views to a set of corresponding relational views. In this way, we transform the problem of updating relational databases through XML views into the classical problem of updates through relational views.*

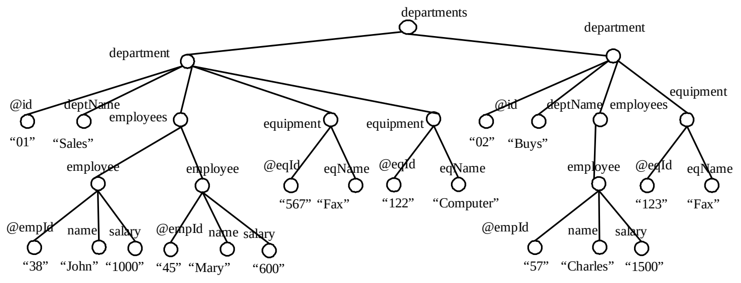
1. Introduction

XML is frequently used as an interface to relational databases. In this scenario, XML documents (views) are exported from relational databases and published, exchanged, or used as the internal representation in user applications. This fact has stimulated research in exporting and querying relational data as XML views [Fernández et al. 2002, Shanmugasundaram et al. 2000, Shanmugasundaram et al. 2001]. 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 the relational database?

The approach we take for solving this problem is to take advantage of the well-studied problem of updates through relational views. We present a solution to the XML view update problem based on a mapping from XML views to relational views. Specifically, we (i) map an XML view to a set of relational views; (ii) map updates over the XML view to updates over the corresponding relational views; and (iii) use existing work on updates through relational views [Dayal and Bernstein 1982] to map the updates to the underlying relational database. The starting point of these mappings is a formalism that we call *query trees*. 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, *query trees* represent a common form of XML views that allows nesting, composed attributes, heterogeneous sets and repeated elements. Query trees are expressive enough to capture the XML views that we have encountered in practice, yet are simple to understand and manipulate. Its expressive power is equivalent to that of IBM DB2 DAD files [Cheng and Xu 2000].

An example of an XML view is shown in Figure 1. It shows *departments*, their *employees* and *equipment*. In this example, and in every example of this paper, we use the

*Vanessa is currently working as a researcher at COPPE/UFRJ.



```
<!ELEMENT departments (department*)>
<!ELEMENT department
    (deptName, employees, equipment*)>
<!ATTLIST department id CDATA #REQUIRED>
<!ELEMENT deptName (#PCDATA)>
<!ELEMENT employees (employee*)>
<!ELEMENT employee (name,salary)>
<!ATTLIST employee empId CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT equipment (eqName)>
<!ATTLIST equipment eqId CDATA #REQUIRED>
<!ELEMENT eqName (#PCDATA)>
```

Figure 1. XML view containing departments, employees and equipment

```
Department(DeptId, DeptName)
    primary key DeptId
Employee(EmpId, EmpName, Salary, DeptId)
    primary key(EmpId), foreign key(DeptId) references Department
Equipment(EqId, EqDescription, DeptId)
    primary key(EqId), foreign key(DeptId) references Department
```

Figure 2. Sample Database

database schema shown in Figure 2. It composed of three tables: *Department*, *Employee* and *Equipment*. Employee carries a foreign key that represents the department in which he/she works. Similarly, equipment carries a foreign key, which denotes the department it belongs to.

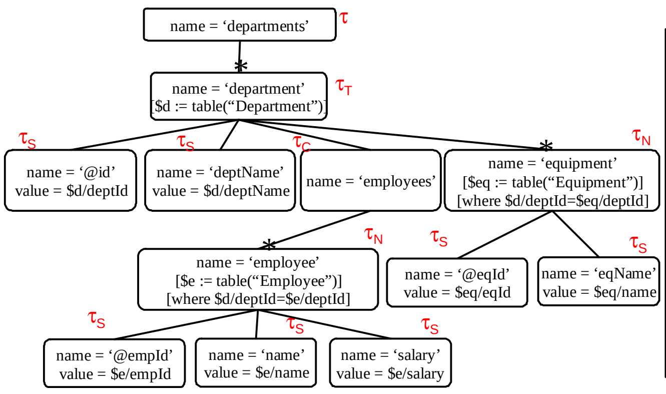
The main contributions of this work are: (i) a view definition formalism (*query trees*); (ii) the mapping of an XML view to a set of corresponding relational views, and the mapping of updates over the XML view to updates over the corresponding relational views. We thus transform the open problem of updating relational databases through XML views into the well-studied problem of updating relations through relational views.

The outline of this paper is as follows. Section 2 presents related work. Query trees are presented in Section 3. Section 4 presents a simple language for updating XML views. The mapping of XML views to a set of underlying relational views is given in Section 5, along with an algorithm for mapping insertions, modifications and deletions on XML views to updates on the underlying relational views. We conclude in Section 6.

2. Related Work

Several papers explore the subject of building and querying XML views over relational databases [Fernández et al. 2002, Shanmugasundaram et al. 2000, Shanmugasundaram et al. 2001]. 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. In our approach the goal is to update the resulting views rather than to query them. Commercial relational databases also offer support for extracting XML views over relational databases. However, they do not present complete solutions for updates.

The closest work on updates through XML views is that of [Wang et al. 2003, Wang and Rundensteiner 2004]. In [Wang et al. 2003], 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



(a)

Ex1: Insertion
ref=/departments/department[@id="01"]/employees
Δ = {`<employee empId="47">`
```
      <name>Joseph</name>
      <salary>9000</salary>
    </employee>}
```

Ex2: Modification
ref=/departments/department[@id="02"]/deptName
Δ = {Purchase}

Ex3: Deletion
ref=/departments/department[@id="01"]/equipment[@eqId="122"]

(b)

Figure 3. (a) Example of query tree (b) Example of update operations

legacy databases. The approach in [Wang and Rundensteiner 2004] presents an extension to the notion of clean source of Dayal and Bernstein [Dayal and Bernstein 1982]. 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 [Braganholo et al. 2003a]. Neither of these papers discuss how updates are translated to the underlying relational database.

3. Query Trees

Query trees can be thought of as an internal representation of the XML view extraction query. We use this abstract representation rather than a user-level view definition language mainly because reasoning about updates and the updatability of an XML view is performed at this level. The characteristics considered in these reasonings are 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.

An example of a query tree can be found in Figure 3, which retrieves departments, employees and equipment. The query tree resembles the structure of the resulting XML view, which is shown in Figure 1. 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.

A query tree is a tree with a set of nodes and a set of edges. 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. All nodes have a name that represents the tag name of the XML element associated with this node in the resulting XML view. Leaf nodes have a value. Leaf nodes which names start with "@" are considered to be XML attributes. Starred nodes (nodes whose incoming edge is starred) may have one or more *source* annotations and zero or more *where* annotations. *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. The value of a leaf node n is of form $\$x/A$, where $\$x$ is bound to table T by a source annotation on n or some ancestor of n and A is an attribute of table T.

Returning to the example of Figure 3, the root *departments* has a set of *department* child nodes (*-edge). The *department* node is annotated with a binding for *$d* (to table Department), and has several children at the end of simple edges (*@id*, *deptName* and *employees*). The value of its *id* attribute is specified by the path *$d/deptId*, and that of *deptName* is specified by *$d/deptName*. The node *employees* has a *-edge children, *employee*. A source annotation on the *employee* node binds *$e* to table Employee, and its where annotation connects tuples in Department to tuples in Employee. Node *equipment* is also a *-node, and it has source and where annotations analogously to Employee.

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: (i) The root has abstract type τ; (ii) Each leaf node has abstract type τ_S; (iii) Each non-leaf node with an incoming simple edge has abstract type τ_C; (iv) Each starred node which is either a leaf node or whose subtree has only simple edges has an abstract type of τ_N; (v) All other starred nodes have abstract type τ_T. 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. An example of this abstract typing is shown in Figure 3.

Before turning to the mapping we claim two facts about query trees that will be used throughout the paper (the proofs are available at [Braganholo et al. 2004]): (i) There is at least one τ_N node in the abstract type of a query tree qt; (ii) 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.

Instantiation of Query Trees. The semantics of a query tree follows the abstract type of its nodes. The algorithm [Braganholo et al. 2004] constructs the XML view resulting from a query tree qt recursively. The basic idea is that the source and where annotations in each starred node n are evaluated in the database instance, 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 evaluation of the query tree in Figure 3 is shown in Figure 1.

View DTD. Query tree views defined over a relational database have a well-defined schema (DTD) that is easily derived from the tree [Braganholo et al. 2004]. As an example, the DTD of the view produced by the query tree of Figure 3 is shown in Figure 1.

4. Update Language

In this section, we present a simple update language for XML views. Updates are specified using path expressions (*ref*) 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.

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 */departments/department[@id="01"]* is */departments/department*.

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. Examples of update operations are presented in Figure 3(b).

Note that not all insertions and deletions make sense since the resulting XML view may not conform to the DTD. For example, the deletion specified by */departments/department/deptName* would not conform to the DTD of Figure 3 since *deptName* is a required subelement. We must also check that Δ's inserted and subtrees deleted are correct. In [Braganholo 2004], we present strict rules for checking for schema conformance.

5. 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 [Dayal and Bernstein 1982, Keller 1985, Bancilhon and Spyratos 1981] 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 to a relational view. As we will show later in this section, there are cases where a single XML view must be mapped to a *set* of relational views.

5.1. 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, and the *split* process deals with query trees that have more than one node of type τ_N. It splits the query tree in several *split trees*, so that each of them has a single node of type τ_N. Then, the *map* process can be applied. In [Braganholo et al. 2004], we formally prove the correctness of the mapping process presented below.

Map. Given a query tree qt with only one τ_N node, the corresponding SQL view statement is generated as follows: (i) 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*; (ii) 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. 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; (iii) 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. 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>
```

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 3 which has two τ_N nodes (*employee* and *equipment*). If we follow the mapping process described above, the tables Employee and Equipment will be joined, resulting in a cartesian product. In this expression, an employee is repeated for each equipment, 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 (*map*).

The splitting process consists of 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.

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.

Using the split trees resulting from this process for the query tree of Figure 3, the corresponding relational views *ViewEmployee* and *ViewEquipment* are:

```
CREATE VIEW VIEWEMPLOYEE AS SELECT d.deptId AS id, d.deptName AS deptName,
     e.empId AS empId, e.name AS name, e.salary AS salary
FROM (Department AS d LEFT JOIN Employee e ON d.deptId=e.deptId);
CREATE VIEW VIEWEQUIPMENT AS SELECT d.deptId AS id, d.deptName AS deptName,
     eq.eqId AS eqId, eq.eqName AS eqName
FROM (Department AS d LEFT JOIN Equipment eq ON d.deptId=eq.deptId)
```

5.2. 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. The formal algorithms for translating insertions, deletions and modifications are presented in [Braganholo 2004], together with the proof of their correction.

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 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 Ex1 in Figure 3, we use the unqualified update path */departments/department/employees* on the query tree of

Figure 3, and find that the type of the update point is $\tau_C(employees)$. Continuing from $\tau_C(employees)$ using the structure of Δ, we discover that the only τ_N node in Δ is its root, which is of type $\tau_N(employee)$. The underlying view affected will therefore be *ViewEmployee*. We then use the update path *ref= /departments/department[@id="01"]/employees* to identify update points in the XML document. In this case, there is only one node. Therefore, a single SQL insert statement against view *ViewEmployee* 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 Ex1, Δ specifies *empId="47"*, *name="Joseph"* and *salary="9000"*. Along the path from the update point to the root in the XML instance of Figure 1, we find *id="01"* and *deptName="Sales"*. Combining this information, we generate the following SQL insert statement:

```
INSERT INTO VIEWEMPLOYEE (id, deptName, empId, name, salary)
VALUES ("01","Sales","47","Joseph", 9000)
```

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 Ex2 in Figure 3. The unqualified *ref* is *departments/department/deptName*. The τ_N nodes in the subtree rooted at *department* (the first τ_T or τ_N ancestor of *deptName*) are $\tau_N(employee)$ and $\tau_N(equipment)$, and we will therefore generate SQL update statements for both *ViewEmployee* and *ViewEquipment*. We then use the qualification *id = "02"* from *ref = departments/department[@id="02"]/deptName* together with the new value in Δ, to yield the following SQL modify statements:

```
UPDATE VIEWEMPLOYEE SET deptName="Purchase" WHERE id="02";
UPDATE VIEWEQUIPMENT SET deptName="Purchase" WHERE id="02"
```

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 Ex3 of Figure 3. The unqualified update path expression is */departments/department/equipment*. The only τ_N node in the subtree indicated by this path in the query tree is τ_N*(equipment)*. This means that the deletion will be performed in *ViewEquipment*. Examining the update path */departments/department[@id="01"]/equipment[@eqId="122"]* yields the label-value pairs *id="01"* and *eqId="122"*. Thus the deletion on the XML view is translated to SQL as: `DELETE FROM VIEWEQUIPMENT WHERE id="01" AND eqId="122"`.

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. Summary and Concluding Remarks

In this work 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 query trees, 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 work is the mapping of XML views 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 relational views [Dayal and Bernstein 1982, Keller 1985, Bancilhon and Spyratos 1981] can be leveraged.

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 [Braganholo et al. 2003b].

Since we use relational views in our solution, we are limited to the cases where it is possible to update those relational views. When it is not possible to map a relational view update to the relational database, the algorithm we have adopted [Dayal and Bernstein 1982] detects it, and passes this information to our system. To diminish the "surprises" the user may have due to this limitation, in [Braganholo 2004] we have made a complete updatability study of XML views. This study presents the characteristics the XML view must have in order for it to be updatable. These characteristics were discovered based on the structure that the underlying relational views would have (based on our mapping rules).

Besides the contributions presented in this paper, in [Braganholo 2004] we also propose: (i) an extension of XQuery to be used as view definition language; (ii) an updatability study of XML views constructed by query trees; (iii) an evaluation of query trees with respect to real-world views; and (iv) a prototype that implements our approach.

This work has resulted in a number of scientific publications. The main ones are [Braganholo et al. 2004, Braganholo et al. 2003b, Braganholo et al. 2003a]. Two related papers were also published in IIWAS 2001 and WTDBD 2002. The SBBD paper [Braganholo et al. 2003b] was nominated to the *best paper award* in 2003. Additionally, four undergraduate projects resulting from this work were supervised by Carlos Heuser and Vanessa Braganholo.

References

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

Braganholo, V. (2004). *From XML to Relational View Updates: applying old solutions to solve a new problem*. Phd thesis, UFRGS, Porto Alegre, RS, Brazil.

Braganholo, V., Davidson, S. B., and Heuser, C. A. (2003a). On the updatability of XML views over relational databases. In *WebDB*, San Diego, CA.

Braganholo, V., Davidson, S. B., and Heuser, C. A. (2003b). UXQuery: building updatable XML views over relational databases. In *SBBD*, pages 26–40, Manaus, AM, Brasil.

Braganholo, V., Davidson, S. B., and Heuser, C. A. (2004). From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, Toronto, Canada.

Cheng, J. and Xu, J. (2000). XML and DB2. In *ICDE*, San Diego, CA.

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

Fernandez, M., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W.-C. (2002). Silkroute: A framework for publishing relational data in XML. *ACM TODS*, 27(4):438–493.

Keller, A. M. (1985). Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, Portland, Oregon.

Shanmugasundaram, J., Kiernan, J., Shekita, E., Fan, C., and Funderburk, J. (2001). Querying XML views of relational data. In *VLDB*, Roma, Italy.

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

Wang, L., Mulchandani, M., and Rundensteiner, E. A. (2003). Updating XQuery views published over relational data: A round-trip case study. In *XML Database Symposium*, Berlin, Germany.

Wang, L. and Rundensteiner, E. A. (2004). On the updatability of XML Views Published over Relational Data. In *International Conference on Conceptual Modeling, ER*, Shanghai, China.