UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANDRÉ P. VARGAS
VANESSA P. BRAGANHOLO
CARLOS A. HEUSER

Conflict Resolution and Difference Detection in Updates through XML views

Research Report
RP-352

Porto Alegre, December 2005

ACKNOWLEDGMENTS

This research is partially supported by CNPq (XML-Broker Research Project, Edital Universal) and FAPERJ.

CONTENTS

LIST OF FIGURES

ABSTRACT

XML is the *de facto* standard for exchanging information over the Web. However, most of the enterprises still store their data in relational databases. In this paper, we focus on B2B scenarios where XML views are extracted from relational databases and sent over the Web to another application that edits them and sends them back after a certain (usually long) period of time. In such transactions, it is unrealistic to lock the base tuples that are in the view to achieve concurrency control. Thus, there are some issues that need to be solved: first, to identify what changes were made in the view, and second, to identify and solve conflicts that may arise due to changes in the database state during the transaction. We address both of these issues in this paper by proposing an approach that uses our XML view update system PATAXÓ.

Keywords: Updates through views, Delta Detection, Conflict Resolution.

1 INTRODUCTION

XML is increasingly being used as an exchange format between business to business (B2B) applications. In this context, a very common scenario is one in which data is stored in relational databases (mainly due to the maturity of the technology) and exported in XML format before being sent over the Web. In literature, there are several work that deal with extracting XML documents (views) over relational databases. Among them we can cite XPeranto (SHANMUGASUNDARAM et al., 2001), SilkRoute (FERNáNDEZ et al., 2002), and others.

These work, however, address only part of the problem, that is, they know how to generate and query XML views over relational databases, but they do not know how to update those views. In B2B environments, enterprises need not only to obtain XML views, but also to update them. An example is a company B (buyer), that buys products from another company – company S (supplier). One could think on B asking S for an *order form*. B would them receive this form (an empty XML view) in a PDA of one of its employees who would fill it in and send it back to S. S would them have to process it and place the new order in its relational database. This scenario is not so complicated, since the initial XML view was empty. There are, however, more complicated cases. Consider the case where B changes its mind and asks S its order back, because it wants to change the quantities of some of the products it had ordered before. In this case, the initial XML view is not empty, and S needs to know what changes B made to it, so it can reflect the changes back to the database.

In previous work (BRAGANHOLO; DAVIDSON; HEUSER, 2004a), we have proposed PATAXÓ, an approach to update relational databases through XML views. In this approach, XML views are constructed using UXQuery (BRAGANHOLO; DAVIDSON; HEUSER, 2003a), an extension of XQuery (BOAG et al., 2005), and updates are issued through a very simple update language. The scenario we address in this paper is different in the following senses:

- In PATAXÓ (BRAGANHOLO; DAVIDSON; HEUSER, 2004a), updates are issued through an update language that allows insertions, deletions and modifications. In this paper, we deal with updates done directly over the XML view, that is, users directly *edit* the XML view.

- By allowing the edition of the XML view, a new problem arises – that of knowing exactly what changes were made to the view. We address this by calculating the *delta* between the original and the updated view. Algorithms in literature (COBENA; ABITEBOUL; MARIAN, 2002; CHAWATHE; GARCIA-MOLINA, 1997; WANG; DEWITT; CAI, 2003; CURBERA; EP-

STEIN, 1999) may be used in this case, but need to be adapted for the special features of the updatable XML views produced by PATAXÓ.

- In PATAXÓ (BRAGANHOLO; DAVIDSON; HEUSER, 2004a), we rely on the transaction manager of the underlying DBMS. As most DBMS apply the ACID transaction model, this means that we simple locked the database tuples involved in a view until all the updates were translated to the database. In B2B environments, this is impractical because the transactions may take a long time to complete (CHRYSANTHIS; RAMAMRITHAM, 1994). Returning to our example, company B could take days to submit the changes to its order back to S. The problem in this case is what to do when the database state changes during the transaction (because of external updates). In such cases, the original XML view may not be valid anymore, and lots of types of conflicts may occur.

In this paper, we propose an approach to solve the open problems listed above. We use PATAXÓ (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) to both generate the XML view and to translate the updates over the XML view back to the underlying relational database. For this to be possible, the update operations that were executed over the XML view need to be detected and specified using the PATAXÓ update language. It is important to notice that not all update operations are valid in this context. For example, PATAXÓ does not allow changing the tags of the XML elements, since this modifies the view schema – this kind of modification can not be mapped back to the underlying relational database.

We assume the XML view is updatable. This means that all updates applied to it can be successfully mapped to the underlying relational database. In (BRAGANHOLO; DAVIDSON; HEUSER, 2004a), we present a set of rules the view definition query must obey in order for the resulting XML view to be updatable. An example of non-updatable view would be a view that repeats the customer name for each item of a given order. This redundancy causes problems in updates, thus the view is not updatable.

1.1 Application Scenarios

To illustrate our proposal, in this section we introduce two applications scenarios.

1.1.1 Forms

A company wants to get information through the Web. This company wants to generate XML forms (using XForms (DUBINKO et al., 2003), for instance) so that its underlying database can be automatically feed by custumers. This data is sent to customers that fill in the form and send the data back to the company (this is done transparently through Web Browsers).

In this context, the application needs to:

- Generate the XML form using PATAXÓ.
- Receive the altered XML document (filled form) and detect what modifications were made to it.
- Update the underlying database through PATAXÓ.

```
Customer (custId, name, address),
   primary key (custId)
Product (prodId, description, curPrice),
   primary key (prodId)
Order (numOrder, date, custId, status),
   primary key (numOrder),
   foreign key (custId) references Customer
LineOrder (numOrder, prodId, quantity, price),
   primary key (numOrder, prodId),
   foreign key (prodId) references Product,
   foreign key (numOrder) references Order
                  (a)
```

```
//increases price of "blue pen"
UPDATE Product
SET curPrice = 0.10
WHERE prodId = "BLUEPEN";

UPDATE LineOrder
SET price = 0.10
WHERE prodId = "BLUEPEN"
AND status="open";


        (b)
```

Figure 1.1: (a) Sample database of company S (b) Updates made over the database

In this scenario, since only insertions are made, there is no conflicts, unless a related information (foreign key) has been deleted from the database by other application.

1.1.2 Mobile applications

This scenario comprises companies B and S, introduced above. Company S has a relational database that stores orders, products and customers. The schema of this database is shown in Figure 1.1(a). Now, let's exemplify the scenario previously described. Company B requests its order to company S so it can alter it. The result of this request is the XML view shown in Figure 1.2 (the numbers near the nodes, shown in red in the Figure, are used so we can refer to a specific node in our examples). While company B is analyzing the view and deciding what changes it will make over it, the relational database of company S is updated as shown in Figure 1.1(b). These updates may have been made directly over the database, or through some other XML view. The main point is that the updates over *LineOrder* affect the XML view that is being analyzed by company B. Specifically, it changes the price of one of the products that B has ordered (blue pen).

Meanwhile, B is still analyzing its order (XML view) and deciding what to change. It does not have any idea that product "blue pen" had its price doubled. After 5 hours, it decides to make the changes shown in Figure 1.3 (the changes are shown in boldface in the figure). The changes are: increase the quantity of blue pens to 200, increase the quantity of red pens to 300, and order a new item (100 notebooks (NTBK)).

When S receives the updated view, it will have to:

- Detect what were the changes made by B in the XML view;
- Detect that the updates shown in Figure 1.1(b) affect the view returned by B, and detect exactly what are the conflicts;
- Decide how to solve the conflicts, and update the database using PATAXÓ.

We will show each of these steps on Section 4. Since this is the most complete scenario, we will use it as the running example of the paper.

1.2 Contributions and Organization of the Text

The main contributions of this paper are:

- A delta detection technique tailored to the PATAXÓ XML views and update language;

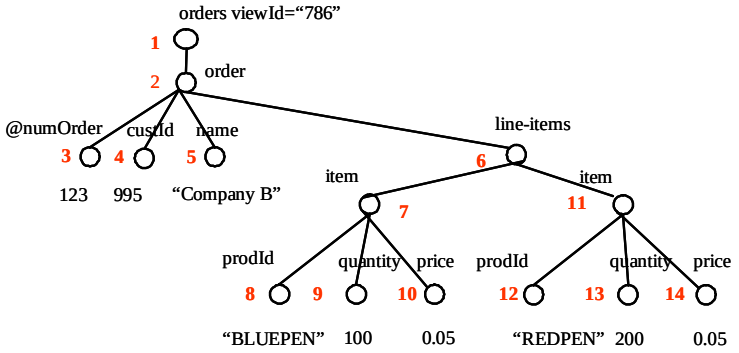

Figure 1.2: Original XML view

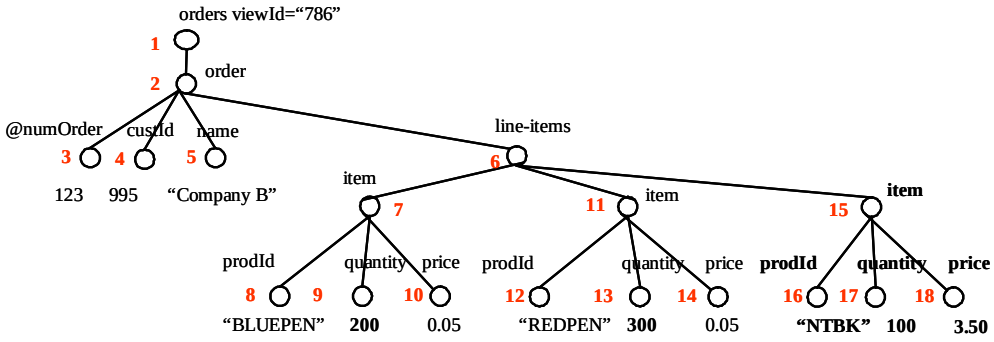

Figure 1.3: XML view updated by company B and returned to company S

- An approach to verify the status of the database during the transaction. This is done by comparing the status of the database in the beginning of the transaction with the status of the database in the time the updated view is returned to the system.
- A conflict resolution technique, based on the structure of the XML view.
- A merge algorithm to XML views that emphasizes the conflicts caused by changes in the database state during the transaction.

The remaining of this paper is organized as follows. Section 2 discusses work related to the problem of detecting changes in XML data instances and on solving conflicts. Section 3 presents an overview of the PATAXÓ approach. Section 4 presents our technique to detect deltas in XML views, and Section 5 presents a solution to the problems caused by conflicts. Finally, we conclude in Section 6.

2 RELATED WORK

There has been several work in literature that detects changes between two XML documents. These changes are specified by a set of update operations (in a *script* format) usually called *delta*. An edit script, thus, is a sequence of basic edit operations that convert one tree into another (CHAWATHE et al., 1996). The first efforts in this area were developed for non-structured data, like textual files or strings. An example of this is CVS (FOGEL; BAR, 2003), a system that is capable of detecting differences between two textual files. The unit of comparison in this case, is a *line* of the file. This is not appropriate for XML files (OLIVEIRA; MURTA; WERNER, 2005), since we may have two XML documents that have the exact same tags, but one of them with textual elements in the same line, like in this example: `...<order><customer>...`, and the other one with textual elements occupying different lines. In this case, CVS would detect differences in the documents, although they are not meaningful.

One of the first proposals to support structured (tree-shaped) data is MH-DIFF (CHAWATHE; GARCIA-MOLINA, 1997). It supports tree manipulation operations like insertion of nodes, deletion of nodes, changes in node labels, moves of subtrees, copies and glues of subtrees. This approach, however, has a restriction in the manipulation of subtrees: it does not support insertion and deletion of subtrees. Insertion and deletion of subtrees are identified node by node, which most of the times does not correspond to the reality.

Algorithms specifically designed to find differences in XML trees are XMLTreeDiff (CURBERA; EPSTEIN, 1999), Xy-Diff (COBENA; ABITEBOUL; MARIAN, 2002) and X-Diff (WANG; DEWITT; CAI, 2003). XMLTreeDiff and XyDiff are both designed for *ordered* XML documents. Thus, they detect changes in the position of a given element among its siblings. X-Diff detects changes in *unordered* XML documents, and thus is very well suited for cases where XML documents represent relational, thus unordered, content.

Techniques for consistency control of disconnected database replicas are presented in (KLIEB, 1996; TERRY et al., 1995; PHATAK; BADRINATH, 1999). To solve such problem, Phatak and Badrinath (PHATAK; BADRINATH, 1999) propose a reconciliation phase that synchronizes operations. (KLIEB, 1996) uses conflict detection and dependencies between operations. However, these approaches do not deal with the XML views problem or require the semantics of the data to be known. In our paper, we use some of the ideas of (KLIEB, 1996) in the XML context.

3 THE PATAXÓ APPROACH

As mentioned before, PATAXÓ (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) is a system that is capable of constructing XML views over relational databases and mapping updates specified over this view back into the underlying relational database. To do so, it uses an existing approach on updates through relational views (DAYAL; BERNSTEIN, 1982). Basically, a view query definition expressed in UXQuery (BRAGANHOLO; DAVIDSON; HEUSER, 2003a) is internally mapped to a query tree (BRAGANHOLO; DAVIDSON; HEUSER, 2004a). Query trees are a formalism that captures the structure and the source of each XML element/attribute of the XML view, together with the restrictions applied to build the view. As an example, the query tree that corresponds to the view query that generates the XML view of Figure 1.2 is shown if Figure 3.1. The interested reader can refer to (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) for a full specification of query trees.

In this paper, it will be important to recognize certain types of nodes in the query tree and in the corresponding view instance. In the query tree of Figure 3.1, node *order* is a *starred-node* (*-node). Each starred node generates a collection of (possibly complex) elements. Each such element carries data from a database tuple, or from a join between two or more tuples (tables Customer and Order, in the example). We call each element of this collection a *starred subtree*. The element itself (the root of the subtree), is called *starred element*. In the example of Figure 1.2 (which is generated from the query tree of Figure 3.1), nodes 2, 7 and 11 are *starred elements* (since they are produced by starred nodes of the corresponding query tree).

To map updates over the XML view to the underlying relational database, PATAXÓ maps a query tree to a *set* of corresponding relational views. In the same way, updates over the XML view are mapped to updates over the corresponding relational views. It then uses the work of Dayal and Bernstein (DAYAL; BERNSTEIN, 1982) to map the updates to the relational database (see (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) for further details on these mappings). In our example, the query tree of Figure 3.1 is mapped into a single relational view specified below:

```
CREATE VIEW VIEWORDER AS
SELECT o.numOrder AS numOrder, c.custId AS custId, c.name AS name,
   l.prodId AS prodId, l.quantity AS quantity, l.price AS price
FROM ((Order AS o INNER JOIN Customer ON c.custId = o.custId) LEFT JOIN
      LineOrder AS l ON o.numOrder = l.numOrder)
WHERE o.numOrder = "123" AND c.custId = "995"
```

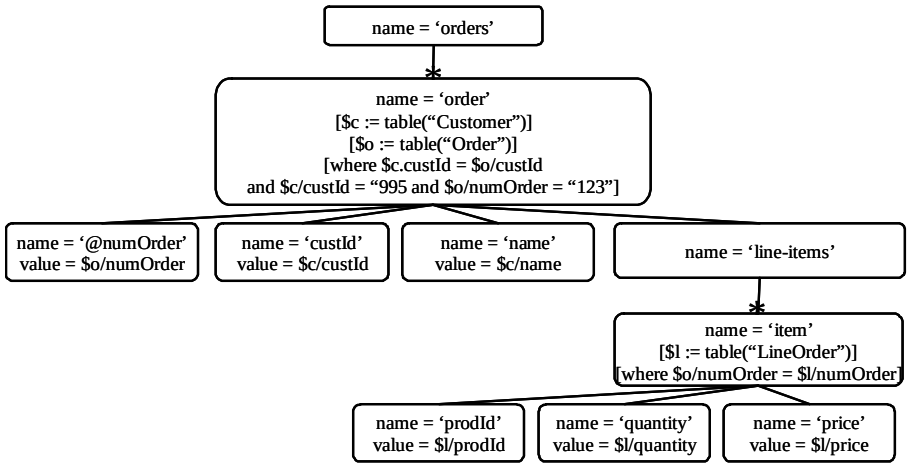

Figure 3.1: Query tree that generated the XML view of Figure 1.2

```
<!ELEMENT orders (order)*>
<!ATTLIST orders viewId CDATA #FIXED "786">
<!ELEMENT order (custId, name, line-items)>
<!ATTLIST order numOrder CDATA #REQUIRED>
<!ELEMENT line-items (item*)>
<!ELEMENT item (prodId, quantity, price)>
```

Figure 3.2: DTD of the XML views on Figures 1.2 and 1.3

To be able to map the view updates to the database, PATAXÓ requires that the database is in BCNF, and that primary keys are preserved in the view, joins are made by key-foreign keys, and nesting is done from the owner relation to the owned relation. These are all properties of updatable views. See (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) for further details.

3.1 Updates in PATAXÓ

In this section, we overview the update language used by PATAXÓ. This will be important in the context of this paper, since the deltas detected by our proposal will be expressed in this update language.

As mentioned before, PATAXÓ uses a very simple update language. Basically, it is expressed by a triple $<t, \Delta, ref>$, where t is the type of the update operation (*insert*, *delete* or *update*), Δ is the subtree to be inserted or an atomic value to be modified, and *ref* is a path expression that points to the update point in the XML view. The update point *ref* is expressed by a simple XPath expression that only contains child access (/) and conjunctive filters.

Not all update specifications are valid, since they need to be mapped back to the underlying relational database. Mainly, the updates applied to the view need to follow the view DTD. PATAXÓ generates the view together with its DTD, and both the view and the DTD are sent to the client application. The DTD of the XML view of Figure 1.2 (and consequently of view on Figure 1.3) is shown on Figure 3.2. It was altered by our system to include the `viewId` attribute – a transaction identifier (not generated by PATAXÓ) which cannot be modified during the entire

transaction. The `#PCDATA` declarations for the leaf nodes are omitted due to space restrictions. The remaining restrictions regarding updates are as follows:

1. Subtrees inserted must represent a (possibly complex/nested) database tuple. This restriction corresponds to adding only subtrees rooted at starred nodes in the query trees of (BRAGANHOLO; DAVIDSON; HEUSER, 2004a). Such elements correspond to elements with cardinality "*" in the DTD. Thus, in this paper, it is enough to know that only subtrees rooted at elements with cardinality "*" in the DTD can be inserted. In Figure 1.3 the inserted subtree `item` (node 15) satisfies this condition.

2. The above restriction is the same for deletions. Subtrees deleted must be rooted at a starred node in the query tree. This means that in the example view, we could delete `order` and `item` subtrees.

All of these restrictions can be verified by checking the updated XML view against the DTD of the original view. As an example, it would not be possible to delete `name` (which is not a starred element, and so contradicts rule 2 above), since this is a required element in the DTD. Also, it is necessary to check that updates, insertions and deletions satisfy the view definition query. As an example, it would not be possible to insert another `order` element in the view, since the view definition requires that this view has only an order with `numOder` equals "123"(see the restrictions on node *order* of Figure 3.1).

We believe that an example of an update operation in the PATAXÓ language and the corresponding update over the relational views will help the reader to understand several of the decisions we have made in our approach (shown in the next section). Thus, to illustrate, suppose an update over the XML view of Figure 1.2, where we want to change the quantity of blue pens to 200. This would be specified in PATAXÓ as: <*modify*, {200}, *orders/order/item[@numOrder="123"and custId="995"]/item[prodId="BLUEPEN"]/quantity*>. PATAXÓ would then translate this to an update over VIEWORDER as follows:

```
UPDATE VIEWORDER
SET quantity = 200
WHERE numOrder="123" AND custId="995" AND prodId="BLUEPEN"
```

4 SUPPORTING DISCONNECTED TRANSACTIONS

In this section, we describe our approach and illustrate it using the order example of Section 1.1. Figure 4.1 shows the architecture of our proposal. The main modules of the system are: the *Transaction Manager*, *Diff Finder* and *Update Manager*.

The *Transaction Manager* is responsible for controlling the currently opened transactions of the system. It receives a view definition query, passes it to PATAXÓ, receives PATAXÓ's answer (the resulting XML view and its DTD), and before sending it to the client, it: (i) adds an `viewId` to the root of the XML view (this attribute is set to 786 in the example view of Figure 1.2)[1]; (ii) adds this same attribute, with the same value, to the root of the view definition query; (iii) adds and attribute declaration in the view DTD for the `viewId`; (iv) stores the XML view, the view definition query and the view DTD in a *Temporary Storage* facility, since they will have to be used later when the updated view is returned to the system.

When the updated view is returned by the client to the system, the Transaction Manager checks its `viewId`[2] and uses it to find the view DTD and the definition query, which are stored in the Temporary Storage facility. Then it uses the DTD to validate the updated view. If the view is not valid, then the transaction is aborted and the client application is notified. In the case it is valid, then the Transaction Manager sends the view definition query to PATAXÓ, and receives a new XML view reflecting the database state at this moment as a response (we will call it *view'*). This new XML view will be used to check the database state. If it is exactly the same as the original XML view (which is also stored in the temporary storage facility), then the updates made to the database during this transaction do not affect the XML view. In this case all view updates made in the updated XML view may be translated back to the database. Notice that, at this stage, we have three copies of the XML view in the system:

- The *original* XML view (O): the view that was sent to the client at the first place. In our example, the original view is shown in Figure 1.2.
- The *updated* XML view (U): the view that was updated by the client and returned to the system. The updated XML view is shown in Figure 1.3.
- The *view'*: a new XML view which is the result of running the view definition query again, right after the updated view arrives in the system. *View'* is used to capture possible conflicts caused by external updates in the base tuples that

[1]The value that will be assigned to attribute `viewId` is controlled by a sequential counter in the Transaction Manager.

[2]It is a requirement of our approach that the `viewID` is not modified during the transaction.

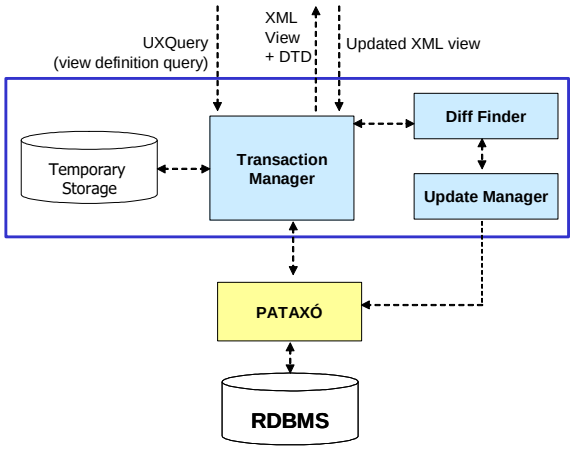

Figure 4.1: Architecture of the proposed solution

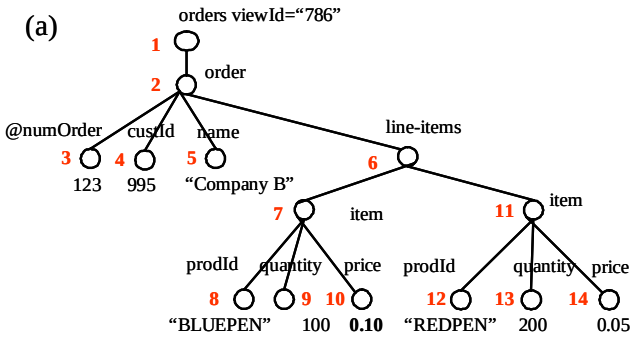

Figure 4.2: View'

are in the original view. As an example, view' is shown in Figure 4.2. Notice that it reflects the base updates shown in Figure 1.1(b).

These three views are sent to the *Diff Finder* module. In this module, two comparisons take place. First, the *original* view is compared to the *updated* view, to find what were the updates made over the view. Next, the *original* view is compared to *view'* to find out if the database state has changed during the transaction. Details on these comparisons are given in Section 4.1.

The deltas found by *Diff Finder* are sent to the *Update Manager*, which analyzes them and detects conflicts. In case there are no conflicts, the Update Manager transforms the updates into updates using the PATAXÓ update language and sends them to PATAXÓ. PATAXÓ then translates them to the relational database. If there are conflicts we try to solve them, as explained in Section 5.

4.1 Detecting deltas in XML views

As mentioned before, the *Diff Finder* is responsible for detecting the changes made in the XML view, and also in the database (through the comparison of *view'* with the original view). To accomplish this, it makes use of an existing diff algorithm that finds *deltas* between two XML views. A *delta* is a set of operations that denotes the difference between two data structures D_1 and D_2 in a way that if we apply *delta* to D_1, we obtain D_2. Using the notation of (WANG; DEWITT; CAI, 2003), this delta can be expressed by $E(D_1 \rightarrow D_2)$.

We adopt X-Diff (WANG; DEWITT; CAI, 2003) as our diff algorithm, mainly because it is capable of detecting the operations supported by PATAXÓ (insertion of subtrees, deletion of subtrees and modification of text values), and considers an unordered model. MH-DIFF (CHAWATHE; GARCIA-MOLINA, 1997) does not support insertion and deletion of subtrees, (it supports only insertion and deletion of single nodes), thus it is not appropriate in our context. Xy-Diff (COBENA; ABITEBOUL; MARIAN, 2002) and XMLTreeDiff (CURBERA; EPSTEIN, 1999) consider ordered models, and so they are not appropriate for our scenario. Further, (WANG; DEWITT; CAI, 2003) shows that in many cases, Xy-Diff mismatches the subtrees, thus generating incorrect results.

Before going further, we present some of the characteristics of X-Diff that will be important in our context.

4.1.1 X-DIFF

According to (WANG; DEWITT; CAI, 2003), the operations detected by X-Diff are as follows:

Insertion of leaf node The operation *Insert(x(name, value), y)* inserts a leaf node x with name *name* and value *value*. Node x is inserted as a child of node y.

Deletion of leaf node Operation *Delete(x)* deletes a leaf node x.

Modification of leaf value A modification of a leaf value is expressed as *Update(x, new-value)*, and it changes the value of node x to *new-value*.

Insertion of subtree Operation *Insert(T_x, y)* inserts a subtree T_x (rooted at node x) as a child of node y.

Deletion of subtree The operation *Delete(T_x)* deletes a subtree T_x (rooted at node x). When there is no doubts about which is x, this operation can be expressed as *Delete(x)*.

An important characteristic of X-Diff is that it uses parent-child relationships to calculate the minimum-cost matching between two trees T_1 and T_2. This parent-child relationship is captured by the use of a node *signature* and also by a hash function. The hash function applied to node y considers its entire subtree. Thus, two equal subtrees in T_1 and T_2 have the same hash value. The node signature of a node x is expressed by *Name(x_1)/.../Name(x_n)/Name(x)/Type(x)*, where (x_1/.../x_n/x) is the path from the root to x, and *Type(x)* is the type of node x. In case x is not an atomic element, its signature does not include *Type(x)* (WANG; DEWITT; CAI, 2003). Matches are made in a way that only nodes with the same signature are matched. Also, nodes with the same hash value are identical subtrees, and thus they are matched by X-Diff.

To exemplify, Figure 4.3 shows the edit script generated by X-Diff for the original (O) and updated (U) views. This Figure also shows the edit script for the original (O) view and view', which is also calculated by *Diff Finder* using X-Diff.

4.1.2 Update Manager

The Update Manager takes the edit script generated by X-Diff and produces a set of update operations in the PATAXÓ update language. Here, there are some issues

$E_1(O \rightarrow U)$ = *Update(9, 200), Update(13, 300), Insert(t_1, 6))*

```
t1 = <item>
       <prodId>NTBK</prodId>
       <quantity>100</quatity>
       <price>3.50</price>
     </item>
```

$E_2(O \rightarrow$ *view'*$)$ = *Update(10, 0.10)*

Figure 4.3: Edit scripts for our example

that need to be taken care of. The main one regards the *update path expressions* (they are referred to as *ref* in the update specification shown in Section 3.1). In PATAXÓ, update operations need to specify an update path, and those are not provided by the edit script generated by X-Diff.

To generate the update path *ref*, we use the base table primary keys as filters in the path expression. Notice that keys must be kept in the view in order for the view to be updated (BRAGANHOLO; DAVIDSON; HEUSER, 2003b, 2004a). Specifically, for an X-Diff operation on node x, we take the path p from x to the view root, and find all the keys that are descendants of nodes in p.

In our example, the keys are *custId, numOrder* and *prodId*[3]. The rules for translating an X-Diff operation into a PATAXÓ operation are as follows. The function *generateRef* uses the primary keys to construct filters, as mentioned above. The general form of a PATAXÓ update operation is $<t, \Delta,$ *ref*$>$.

- *Insert(x(name, value), y)* is mapped to $<$*insert*, x, *generateRef(y)*$>$.
- *Delete(x)* is mapped to $<$*delete*, {}, *generateRef(x)*$>$.
- *Update(x, new-value)* is mapped to $<$*modify*, {*new-value*}, *generateRef(x)*$>$.
- *Insert(T_x, y)* is mapped to $<$*insert*, T_x, *generateRef(y)*$>$.
- *Delete(T_x)* is mapped to $<$*delete*, {}, *generateRef(x)*$>$.

Function *generateRef*(x) works as follows. First, it gets the parent x_n of x, then the parent x_{n-1} of x_n, and continues to get their parents until the root is reached. The obtained elements form a path $p = x_1/.../x_{n-1}/x_n/x$. Then, for each node y in p, it searches for leaf children that are primary keys in the relational database. Use this set of nodes to specify a conjunctive filter that uses the node name and its value in the view. As an example, the operations detected by E_1 in Figure 4.3 are translated as follows:

- *Update(10, 200)* $\equiv <$*modify*, {200}, *orders/order[@numOrder= "123"and custId= "995"]/line-item/item[prodId= "BLUEPEN"]/quantity*$>$
- *Update(15, 300)* $\equiv <$*modify*, {300}, *orders/order[@numOrder= "123"and custId= "995"]/line-item/item[prodId= "REDPEN"]/quantity*$>$

[3]Notice that it is not required that the elements in the XML view have the same name as in the database. If they have different names, it is always possible to find their correspondence in the view definition query.

- *Insert(t_1, 6)* $\equiv$ <*insert*, t_1, *orders/order[@numOrder= "123"and custId= "995"] /line-item*>, with t_1 as shown in Figure 4.3.

PATAXÓ uses the values in the filters in the translation of modifications and deletions, and the values of leaf nodes in the path from the update point to the root in the translation of insertions. This approach, however, causes a problem when some of these values were modified by the user in the view. To solve this, we need to establish an order for the updates. This order must make sure that if an update operation u references a node value x that was modified in the view, then the update operation that modifies x must be issued *before* u. Given this scenario, we establish the following order for the updates: (1) Modifications; (2) Insertions; (3) Deletions.

There is no possibility of deleting a subtree that was previously inserted, since this kind of operation would not be generated by X-Diff. When there is more than one update in each category, then the updates that have the shortest update path (*ref*) are issued first. To illustrate, consider a case where the numOrder is changed (u_1), and the quantity of an item is changed by u_2. Since the numOrder is referenced in the filter of the update path of u_2, then u_1 has to be issued first, so that when u_2 is executed, the database already has the correct value of the numOrder. Notice that this example is not very common in practice, since normally primary key values are not changed.

5 GUARANTEEING DATABASE CONSISTENCY

The detection of conflicts is difficult, because a conflict can have different impacts depending on the application. To illustrate, in our example of orders, the removal of a product from the database means that the customer can not order it anymore. As a counter example, suppose an academic environment in which department D requests a list of all students enrolled in a set of disciplines. Suppose department D is correcting the names of some disciplines, but it is not interested in each particular student enrolled in them. The students are only in the view for statistics reasons, like to check the necessity of creating additional classes of a given discipline. This means that if someone deletes a student from the database while department D is still analyzing the view, the conflict generated by this will not be serious. When department D returns the updated view to the system, no action needs to be taken regarding this student that was not supposed to be in the view. Notice that this student should NOT be added in the database again, since it was in the view when department D requested it in the first place.

The issues above are semantic issues. Unfortunately, a generic system does not know about these issues, and so we take the following approach: The *Diff Finder* uses X-Diff to calculate the edit script for the original XML view O and the view that has the current database state (*view'*). If the edit script is empty, the updates over the updated view can be translated to the database with no conflict. In this case, the Update Manager translates the updates to updates in the PATAXÓ update language (as shown in Section 4.1) and sends them to PATAXÓ so it can map them to the underlying relational database.

However, sometimes the views (original and view') will not be equal, which implies in conflicts. We define a conflict as any update operation that has been issued in the database during the transaction lifetime, and that affects the updates made through the view by the user. In Section 5.1, we present rules to detect conflicts caused by modifications. We leave insertions and deletions for future work.

In our approach, there are two *operational modes* to deal with conflicts. The first one is the *restrictive mode*, in which no updates are translated when there are differences between the views original and view'. This means that, if any update that affects the XML view was made in the database during the transaction (original view $\neq$ view'), then all view updates are rejected. This is a very restrictive approach, where all modifications made over the view are treated as a single atomic transaction.

The second, less restrictive mode of operation is called *relaxed mode*. In this mode, updates that do not cause conflicts are translated to the underlying database. The remaining ones are aborted. To keep database consistency, we assume that some operations do not depend on the others, that is, some updates may coexist

with others done externally, without causing inconsistencies in the database. To recognize such cases, we define a set of rules that are based on the view structure only. Notice that we do not know the semantics of the data in the view nor in the database. Thus, sometimes we may detect an operation to cause conflict even tough semantically it does not cause conflicts. This is the price we pay for not requiring the user to inform the semantics of the data in the view.

5.1 Conflict Detection Rules

In this section, we present rules for the resolution of conflicts in modifications. To detect those conflicts, we assume the following:

A) A user may specify an update based on the value of a non-key attribute. For example, she may want to double the quantity of the blue pen item because of its price. If the price changes in the database, the user may not be interested in doubling the quantity of blue pens anymore. In this example, a non-key attribute (*price*) has influenced the decision of modifying another attribute (*quantity*) of the view.

   In such cases, we abort this operation, since properties of a tuple should not be altered concurrently in this context.

B) The structure of the view impacts in our conflict resolution rules. Due to this, in this paper we assume that the views are structured in the "correct way", that is, we assume they are updatable wrt. insertions, deletions and modifications, according to the rules presented in (BRAGANHOLO; DAVIDSON; HEUSER, 2004b).

With this in mind, we have the following conflict possibilities:

Rule 1: Leaf nodes within the same starred-element. In the structure of the XML view, leaf nodes descending of the same starred element belong to the same database tuple. For example, nodes 8, 9, 10 and 11 in Figure 1.2 are all children of the same starred element (node 7). According to assumption A above, they can not be modified in the view if there is any update to the corresponding tuple in the database. To detect this case, we check that the subtree rooted at node 10 in the original view is equal to a subtree in view'. If not, then no updates over this subtree can be mapped back to the database. Formally, we have:

Definition 5.1 *Let $L = \{l_1, ..., l_n\}$ $(n \geq 1)$) be the set of leaf nodes descending from a starred node s in a given XML view v. Additionally, ensure that s is the first starred ancestor of the nodes in L. If any $l_i \in L$ is modified in the updated view, and some l_j is modified in view' ($i = j$ or $i \neq j$), then the updates in nodes of L are rejected.*

An example of such case can be seen in the modification of node 9 (quantity of blue pens) in Figure 1.3 from 100 to 200. This operation can not be mapped to the database because it conflicts with the update of node 10 (price of blue pens) in view'.

Notice that modifications to nodes belonging to different starred elements, do not conflict. An example is the modification of node 15 (quantity of red pen) from

numOrder	custId	name	prodId	quantity	price
123	995	Company B	BLUEPEN	200	0.05
123	995	Company B	REDPEN	200	0.05
...	...	...	...	...	

Figure 5.1: Original view (Figure 1.2), represented in the relational model

200 to 300 (Figure 1.3). This operation can proceed despite the fact that node 11 (price of blue pen in Figure 4.2) has been changed in view'.

Rule 2: Dependant *starred-subtrees.* Starred subtrees may contain other starred subtrees as descendants. This occurs when there are joins between two base tables in the view definition query. As an example, tables Customer, Order and LineOrder are joined in the XML view of Figure 1.2 (see the view definition in Figure 3.1). If we were to represent the original XML view in a relation, the result would be as shown in Figure 5.1. The fields of the Customer and Order tables repeat for each tuple of LineOrder. In the XML view, this does not happen, since the nesting is done in order to eliminate the redundancy (a property of updatable views (BRAGANHOLO; DAVIDSON; HEUSER, 2004a)). However, it is easy to see that each *item* subtree is semantically connected to its parent *order* tree. We thus define that modifications done in the database that affect the *order* subtree conflicts with modifications to the *item* subtree done through the view. Formally, we have:

Definition 5.2 *Let s_1 and s_2 be two starred subtrees in a given XML view v. Let $L_1 = \{l_{1_1}, ..., l_{1_n}\}$ $(n \geq 1)$) be the set of leaf nodes descending from s_1, but not from its starred subtrees, and $L_2 = \{l_{2_1}, ..., l_{2_k}\}$ $(k \geq 1)$) be the set of leaf nodes descending from s_2, but not from its starred subtrees. Further, let s_1 be an ancestor of s_2. If any $l_{2_i} \in L_2$ is modified in the updated view, and some $l_{1_j} \in L_1$ is modified in view', then the updates conflict, and the modification of l_{2_i} is aborted.*

Notice that in all the above rules, we need to know the correspondence of nodes in the updated view and in *view'*. For example, we need to know that node 12 in the updated view (Figure 1.3) correspond to node 12 in *view'* (Figure 4.2). This can be easily done by using a variation of our *merge* algorithm presented in Section 5.2.

To check for conflicts, each modify operation detected in $E_1(O \rightarrow U)$ is checked against each modify operation in $E_2(O \rightarrow$ *view'*$)$ using the rules above. Formally speaking, we have:

Definition 5.3 *Let O, U, and* view' *be the original view, the updated view, and the view reflecting the current database state, respectively. Let $E_1(O \rightarrow U)$ be the list of modify operations detected between O and U, and $E_2(O \rightarrow$ view'$)$ be the list of modify operation detected between O and* view'*. Each operation u in E_1 and E_2 is of the form* Update(x, new_value)*, where x is the modified node. To detect conflicting operations, run algorithm* detectConflicts(O, U, view', E_1, E_2) *(Algorithm 1).*

```
detectConflicts(O, U, view', E1, E2)

if restrictive mode and E2 is not empty then
  mark all operations in E1 as aborted
else
  for each u1 in E1 do
    for each u2 in E2 do
      /* Rule 1: */
      if u1.x and u2.x descend from the same starred element then
        mark u1 as conflicting (it will be aborted)
        /* no need to check remaining rule */
      else
        /* Rule 2: */
        Let s1 be the first starred ancestor of u1.x
        Let s2 be the first starred ancestor of u2.x
        if s2 is an ancestor of s1 then
          mark u1 as conflicting (it will be aborted)
        end if
      end if
    end for
  end for
end if
```

Algorithm 1: Algorithm *detectConflict*

5.2 Notifying the User

In both configuration modes of our system, we need to inform the user of which update operations were actually translated to the base tables, and which were aborted. The user may want to try to issue them once more. To do so, the system generates a *merge* of the updated data and the current database state. The algorithm works as follows. It starts with the original XML view.

1. Take each delete operation u = Delete(x) in $E(O \rightarrow view')$ and mark x in the original XML view. The markup is made by adding a new parent *pataxo:DB-DELETE* to x, where *pataxo* is a namespace prefix. This new element is connected to the parent of x.

2. Take each insert operation u = Insert(T_x, y) in $E(O \rightarrow view')$, insert T_x under y and add a new parent *pataxo:DB-INSERT* to T_x. Connect the new created element as a child of y. A similar operation is made for the insertion of leaf nodes. Take each operation u = Insert(x(name, value), y), insert it in the original view, add a new parent *pataxo:DB-INSERT* to x and connect it as a child of y.

3. Take each modify operation u = Update(x, new-value) in $E(O \rightarrow view')$, add a new element *pataxo:DB-MODIFY* with value *new-value*. Connect the *pataxo:DB-MODIFY* element as a child of x.

After this, it is necessary to apply the update operations that are in the updated view to the original view, and mark them too. In this step, the markup elements receive a STATUS attribute, which says if the update operation was accepted or

```
<orders viewId="786">
   <order numOrder="123">
      <custId>995</custId>
      <name>Company B</name>
      <line-items>
         <item>
            <prodId>BLUEPEN</prodId>
            <quantity>100
                <pataxo:CLIENT-MODIFY STATUS="ABORT">
                200
                </pataxo:CLIENT-MODIFY>
            </quantity>
            <price>0.05<pataxo:DB-MODIFY>0.10</pataxo:DB-MODIFY> </price>
         </item>
         <item>
            <prodId>REDPEN</prodId>
            <quantity>200
                <pataxo:CLIENT-MODIFY STATUS="ACCEPT">
                300
                <pataxo:CLIENT-MODIFY>
            </quatity>
            <price>0.05</price>
         </item>
         <pataxo:CLIENT-INSERT STATUS="COMMIT">
            <item>
              <prodId>NTBK</prodId>
              <quantity>100</quantity>
              <price>3.50</price>
            </item>
         </pataxo:CLIENT-INSERT>
      </line-items>
   </order>
</orders>
```

Figure 5.2: Result of the *merge* algorithm

aborted. Since we are currently detecting conflicts only between modify operations, we are assuming insertions and deletions are always accepted. We are currently working on the algorithms to detect conflicting insertions and deletions.

1. Take each delete operation u = Delete(x) in $E(O \rightarrow U)$, add a new parent *pataxo:CLIENT-DELETE STATUS="ACCEPT"* to x and connect it to the parent of x.

2. Take each insert operation u = Insert(T_x, y) in $E(O \rightarrow U)$, insert T_x under y and add a new parent *pataxo:CLIENT-INSERT STATUS="ACCEPT"* to T_x. Connect the new created element as a child of y. Also, take each operation u = Insert(x(name, value), y), insert it in the original view, add a new parent *pataxo:CLIENT-INSERT STATUS="ACCEPT"* to x and connect it as a child of y.

3. Take each modify operation u = Update(x, new-value) in $E(O \rightarrow U)$, add a new element *pataxo:CLIENT-MODIFY* with value *new-value*. Connect the *pataxo:CLIENT-MODIFY* element as a child of x. If u is marked in E, then add a *STATUS* attribute to the *pataxo:CLIENT-MODIFY* with value *ABORT*. If not, then add the *STATUS* attribute with value *ACCEPT*.

The result of this merge in our example is shown in Figure 5.2. There may be elements with more than one conflict markup. For example, suppose the client had altered the price of blue pens to 0.02 (the issue of whether this is allowed by the application or not, is out of the scope of this paper). In this case, the element price would have two markups.

```
<price>0.05
   <pataxo:DB-MODIFY>0.10</pataxo:DB-MODIFY>
   <pataxo:CLIENT-MODIFY STATUS="ABORT">0.02</pataxo:CLIENT-MODIFY>
</price>
```

One may argue that we could have tried to automatically solve the conflict caused by the increase of price of red pens. We do not do so because this would probably no be desired in all situations.

After the execution of the merge algorithm, the Transaction Manager receives the new *merged* view (notice that the merged view is an XML document not valid according to the view DTD, since new markup elements were added). It re-generates the view (which is now the original view O), and stores it in the temporary storage facility, since now this is the new *original view*. Then, it sends the *merged* view and view O back to the client application. The client may want to analyze the merged view and to resubmit updates through view O. This second "round"will follow the same execution flow as before. The system will proceed as if it was the first time that updated view arrives in the system.

6 DISCUSSION AND FUTURE WORK

We have presented an approach to support disconnected transactions in updates over relational databases through XML views. Our approach uses PATAXÓ (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) to both generate the views and to translate the updates to the underlying relational database. In this paper, we allow views to be edited, and we automatically detect the changes using X-Diff (WANG; DEWITT; CAI, 2003). We present an algorithm to transform the changes detected by X-Diff into the update language accepted by PATAXÓ. Also, we present a technique to detect conflicts that may be caused by updates over the base relations during the transaction execution. Currently, we only detect conflicts for modifications.

One of the benefits of our approach is that it does not require that updates are done online. In our previous approach (BRAGANHOLO; DAVIDSON; HEUSER, 2004a), the client application must be connected to PATAXÓ in order to issue updates. In this paper, however, we support offline update operations that can be done in offline devices, like PDAs.

This scenario is very common in practice, and we believe that industry will greatly benefit from our work. In the future, we plan to evaluate our approach in real enterprises. Also, we are working on rules to detect conflicts on insertions and deletions. We plan to work on algorithms to solve such conflicts.

REFERENCES

BOAG, S.; CHAMBERLIN, D.; FERNANDEZ, M. F.; FLORESCU, D.; ROBIE, J.; SIMéON, J. **XQuery 1.0**: an XML query language. Available at: <`http://www.w3.org/TR/2005/WD-xquery-20050404/`>. W3C Working Draft.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. UXQuery: building updatable XML views over relational databases. In: SIMPóSIO BRASILEIRO DE BANCO DE DADOS, SBBD, 2003, Manaus, AM, Brazil. **Anais...** [S.l.: s.n.], 2003. p.26–40.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. On the updatability of XML views over relational databases. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 2003, San Diego, CA. **Proceedings...** [S.l.: s.n.], 2003.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. From XML View Updates to Relational View Updates: old solutions to a new problem. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2004, Toronto, Canada. **Proceedings...** San Francisco: Morgan Kaufmann, 2004. p.276–287.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. **Propagating XML View Updates to a Relational Database**. Porto Alegre, RS, Brasil: UFRGS, 2004. (TR-341).

CHAWATHE, S. S.; GARCIA-MOLINA, H. Meaningful Change Detection in Structured Data. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1997, Tucson, Arizona. **Anais...** ACM, 1997. p.26–37.

CHAWATHE, S. S.; RAJARAMAN, A.; GARCIA-MOLINA, H.; WIDOM, J. Change detection in Hierarchically Structured Information. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1996, Montreal, Canada. **Anais...** ACM, 1996. p.493–504.

CHRYSANTHIS, P.; RAMAMRITHAM, K. Synthesis of Extended Transaction Models Using ACTA. **ACM Transactions on Database Systems, TODS**, [S.l.], v.19, n.3, p.450–491, 1994.

COBENA, G.; ABITEBOUL, S.; MARIAN, A. Detecting Changes in XML Documents. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2002, San Jose, California. **Anais...** IEEE Computer Society, 2002. p.41–52.

CURBERA, F.; EPSTEIN, D. Fast Difference and Update of XML documents. In: XTECH, 1999, San Jose, California. **Anais...** [S.l.: s.n.], 1999.

DAYAL, U.; BERNSTEIN, P. A. On the correct translation of update operations on relational views. **ACM Transactions on Database Systems, TODS**, [S.l.], v.8, n.2, p.381–416, Sept. 1982.

DUBINKO, M.; KLOTZ, L. L.; MERRICK, R.; RAMAN, T. V. **XForms 1.0**. 2003. Available at: <http://www.w3.org/TR/2003/REC-xforms-20031014/>. W3C Recommendation.

FERNáNDEZ, M.; KADIYSKA, Y.; SUCIU, D.; MORISHIMA, A.; TAN, W.-C. SilkRoute: a framework for publishing relational data in XML. **ACM Transactions on Database Systems, TODS**, [S.l.], v.27, n.4, p.438–493, Dec. 2002.

FOGEL, K.; BAR, M. **Open Source Development with CVS**. 3.ed. [S.l.]: Paraglyph Press, 2003.

KLIEB, L. Distributed disconnected databases. In: SYMPOSIUM ON APPLIED COMPUTING (SAC), 1996, New York, NY, USA. **Anais...** ACM Press, 1996. p.322–326.

OLIVEIRA, H.; MURTA, L.; WERNER, C. Odyssey-VCS: a flexible version control system for UML model elements. In: INTERNATIONAL WORKSHOP ON SOFTWARE CONFIGURATION MANAGEMENT (SCM), 2005, Lisbon, Portugal. **Anais...** ACM, 2005. p.1–16. Held in conjunction with ESEC/FSE.

PHATAK, S. H.; BADRINATH, B. R. Conflict Resolution and Reconciliation in Disconnected Databases. In: INTERNATIONAL WORKSHOP ON DATABASE & EXPERT SYSTEMS APPLICATIONS (DEXA), 1999. **Anais...** [S.l.: s.n.], 1999.

SHANMUGASUNDARAM, J.; KIERNAN, J.; SHEKITA, E.; FAN, C.; FUNDERBURK, J. Querying XML views of relational data. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2001, Roma, Italy. **Proceedings...** [S.l.: s.n.], 2001.

TERRY, D. B.; THEIMER, M. M.; PETERSEN, K.; DEMERS, A.; SPREITZER, M.; HAUSER, C. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In: SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1995. **Anais...** [S.l.: s.n.], 1995. p.172–183.

WANG, Y.; DEWITT, D. J.; CAI, J.-Y. X-Diff: an effective change detection algorithm for XML documents. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2003, Bangalore, India. **Anais...** IEEE Computer Society, 2003. p.519–530.