Conflict Resolution in Updates through XML views

André Prisco Vargas[1], Vanessa P. Braganholo[2], and Carlos A. Heuser[1]

[1] Instituto de Informática - Federal University of Rio Grande do Sul - Brazil
[2] COPPE - Federal University of Rio de Janeiro - Brazil
[apvargas,heuser]@inf.ufrgs.br, vanessa@cos.ufrj.br

Abstract. 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.

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 [14, 9] before being sent over the Web. The proposes in [14, 9], 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 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 [2], we have proposed PATAXÓ, an approach to update relational databases through XML views. In this approach, XML views are constructed using UXQuery [3], an extension of XQuery, and updates are issued through a very simple update language. The scenario we address in this paper is different in the following senses: (i) In PATAXÓ [2], 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. Thus, we need to know

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 [6, 4, 17, 7] may be used in this case, but need to be adapted for the special features of the updatable XML views produced by PATAXÓ; (ii) In PATAXÓ [2], we rely on the transaction manager of the underlying DBMS. As most DBMS apply the ACID transaction model, this means that we simple lock the database tuples involved in a view until all the updates have been translated to the database. In B2B environments, this is impractical because the transactions may take a long time to complete [5]. 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 conflicts may occur.

In this paper, we propose an approach to solve the open problems listed above. We use PATAXÓ [2] 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 [2], we present a set of rules the view definition query must obey in order for the resulting XML view to be updatable. Basically, this means 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. 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.

Application Scenario Consider companies B and S, introduced above. Company S has a relational DB that stores orders, products and customers. The DB schema is shown in Figure 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 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(b). These updates may have been made directly over the database, or through some other XML view. The main point is that the update over *LineOrder* affects 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 "BLUEPEN" had its price doubled. After 5 hours, it decides to make the changes shown in Figure 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)). Notice

(a)

```
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
```

(b)

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

UPDATE LineOrder
SET price = 0.10
WHERE prodId = "BLUEPEN" AND
numOrder IN (SELECT numOrder
   FROM Order WHERE status="open");
```

Fig. 1. (a) Sample database of company S (b) Updates made over the database

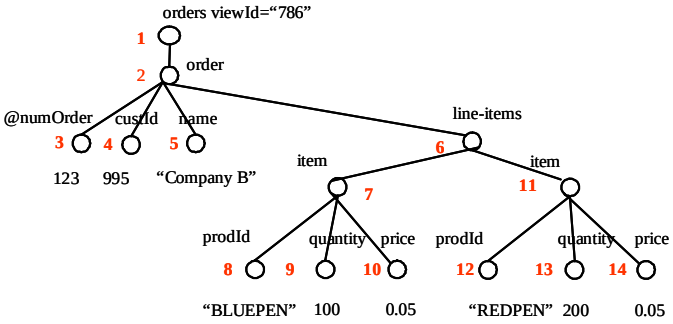

Fig. 2. Original XML view

there that, in order to add a new product in its order, B has to query S for a catalog of products. We assume this has already been done.

When S receives the updated view, it will have to: (i) Detect what were the changes made by B in the XML view; (ii) Detect that the updates shown in Figure 1(b) affect the view returned by B, and detect exactly what are the conflicts; (iii) Decide how to solve the conflicts, and update the database using PATAXÓ.

Contributions and Organization of the Text The main contributions of this paper are: (i) A delta detection technique tailored to the PATAXÓ XML views; (ii) 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; (iii) A conflict resolution technique, based on the structure of the XML view; (iv) 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 related work. Section 3 presents an overview of the PATAXÓ approach. Section 4.1 presents our technique to detect deltas in XML views, and Section 4.2 presents a solution to the problems caused by conflicts. Finally, we conclude in Section 5.

2 Related Work

Extended Transactions As we mentioned in the introduction, in this paper we do not rely only on the ACID transaction model implemented by most of the DBMS. Instead,

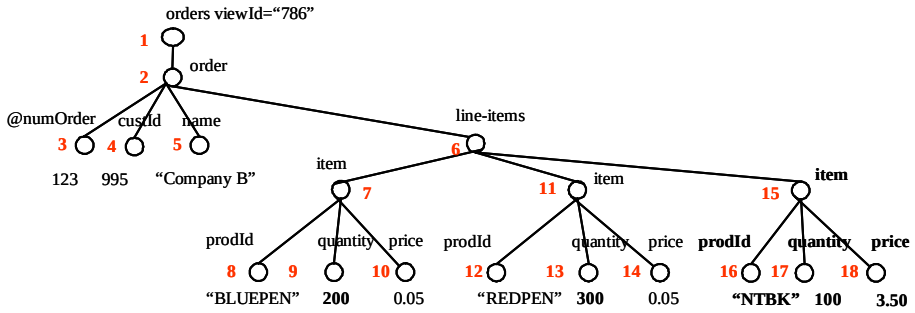

Fig. 3. XML view updated by company B and returned to company S

we propose a mechanism to detect changes that were done offline. This mechanism is responsible for detecting the cases where the offline changes done through the view may conflict to changes done directly through the database. [5] discusses the effects of transaction on objects and the interactions between transactions on several extended transaction models. In our paper, we use the terminology presented in [5], but do not use any of the extended transaction models proposed there. We discuss the reason for that below.

In our paper, although we externally detect conflicts between update operations, we still depend on the ACID transaction model, since the updates are actually executed by the underlying DBMS (we do not want to change the DBMS in anyway). Because of this, even if we have several views being updated at the same time, it is enough to detect conflicts between the database and the view that has just been returned to the system. To exemplify, assume we have a database D and a set of views $V_1, ..., V_n$ specified over D using the exact same view definition query (i.e., the views are identical). Assume also that all of these views are being updated offline. When the updated views are returned to PATAXÓ, we have an order in which they will be analyzed. Assume this order is $V_1, ..., V_n$ (the order in which the views were returned to the system). We then compare V_1 with the current state of D to detect conflicts, and translate non conflicting updates to D. Then, we proceed with the next view in the queue. Notice that the updates done through V_1 are already reflected in D, so we do not need to compare V_1 with V_2 to detect conflicts. Thus, we have isolated transactions.

On a similar line of thought, [1] criticizes the ANSI SQL-92 *Isolation Levels*, and discusses the problems that may occur when several transactions interact over the same data. Since we are not proposing a new transaction manager in our approach, we claim we do not need to worry about such things in our approach.

Harmony Work related to our approach is the Harmony Project [13], in which the authors propose the use of *lenses* to synchronize data in different formats. In [10], the authors propose to use the semantic foundation of *lenses* to support updates through views. Their formal framework treats the database as a concrete format, and views over the database as an abstract format. Then, lenses are used to map concrete data to abstract views (*get component*), and the inverse mapping (the one required to update the database - *putback component*) derives automatically from the get component. After

defined, two abstract views v_1 and v_2 can be synchronized. Comparing to our scenario, we may assume v_1 is the original view and v_2 is the updated view. The concrete format of v_1 is the database D, while the concrete format of v_2 is v_2 itself (in this case we use the identity lens to perform the mapping from v_2 to v_2). When the database is updated, v_1 reflects the changes. The problem here is that when we synchronize v_1 and v_2, we may erroneously reinsert old things in the database. As an example, suppose we have tuples t_1 and t_2 in D. Suppose also that t_1 and t_2 are both in v_1. View v_2, at the beginning, is equal to v_1, so it also has t_1 and t_2. Suppose that, while v_2 is being updated, D is updated to delete t_2. Thus, v_1 will reflect this change, and now it has only t_1. Meanwhile, v_2 is updated to insert t_3, and so it now has t_1, t_2 and t_3. When v_1 is synchronized with v_2, the system finds out that t_2 and t_3 needs to be inserted into v_1 (and consequently into D). It is thus erroneously reinserting t_2. Our approach, in this case, would insert only t_3.

Consistency control of disconnected replicas A problem closely related to our work is the problem of consistency control of disconnected database replicas [11, 15, 12]. To solve such problem, Phatak and Badrinath [12] propose a reconciliation phase that synchronizes operations. [11] 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 [11] in the XML context.

3 The PATAXÓ approach

As mentioned before, PATAXÓ [2] 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 [8]. Basically, a view query definition expressed in UXQuery [3] is internally mapped to a query tree [2]. 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 2 is shown if Figure 4. The interested reader can refer to [2] 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 4, node *order* is a *starred-node* (*-node) [3]. 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

[3] Notice that, despite the fact that the condition *numOrder=123* restricts this view to a single order, node *order* is defined as a starred node. This is because the formal definition of query trees requires that nodes with *source annotations* be starred [2]. In [2], this decision was made to simplify the mapping to relational views – this way, the algorithm does not need to check the *where* annotations to find out whether a given node will have single or multiple instances. More details about the formal definition of query trees can be found in [2]. Notice further that this view would not be *updatable* if it had multiple orders, since the *name* of the customer could be redundant. To solve this problem, orders would have to be nested within customer.

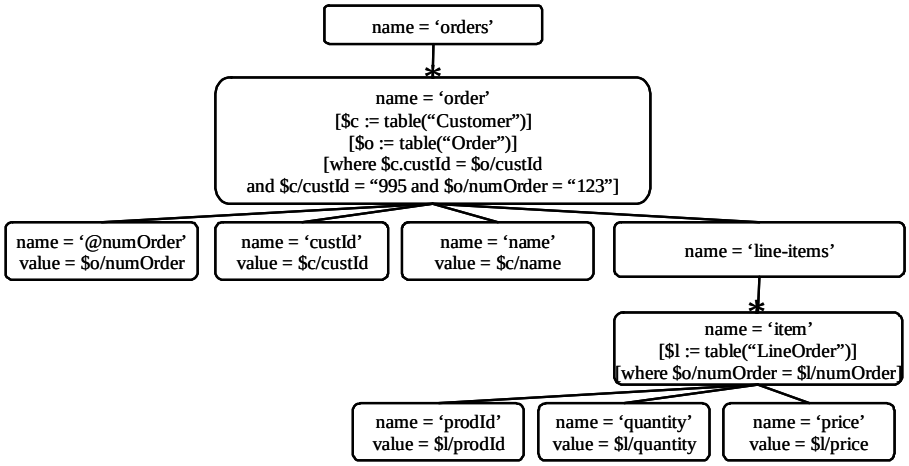

Fig. 4. Query tree that generated the XML view of Figure 2

of this collection a *starred subtree*. The element itself (the root of the subtree), is called *starred element*. In the example of Figure 2 (which is generated from the query tree of Figure 4), nodes 2, 7 and 11 are *starred elements* (since they are produced by starred nodes of the corresponding query tree).

Updates in PATAXÓ As mentioned before, PATAXÓ uses a very simple update language. Basically, it is expressed by a triple $\langle t, \Delta, ref \rangle$, where t is the type of the 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 2 is available in [16]. The remaining restrictions regarding updates are as follows: (i) 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 [2]. 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 3 the inserted subtree `item` (node 15) satisfies this condition. (ii) 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 node `name` (which is not a starred element, and so contradicts rule (ii) 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 `numOrder` equals "123" (see the restrictions on node *order* of Figure 4).

Notice that we do not support "?" cardinality in our model. This is because we map updates over the view to updates of the same type in the relational database (insertions map to insertions, deletions map to deletions, and so on). Supporting optional elements would make us to break this rule. Inserting an optional leaf element would map to modifying a database tuple. In the same way, deleting an optional element would map to modifying the corresponding tuple to NULL. We discuss this in more details in [3].

4 Supporting Disconnected Transactions

In this section, we describe our approach and illustrate it using the order example of Section 1. Our architecture [16] has three main modules: 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 2; the value that will be assigned to attribute `viewId` is controlled by a sequential counter in the Transaction Manager); (ii) adds this same attribute, with the same value, to the root of the view definition query; (iii) adds an 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 user to the system, the Transaction Manager checks its `viewId` (it is a requirement of our approach that the `viewID` is not modified during the transaction) 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 user 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 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 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 are

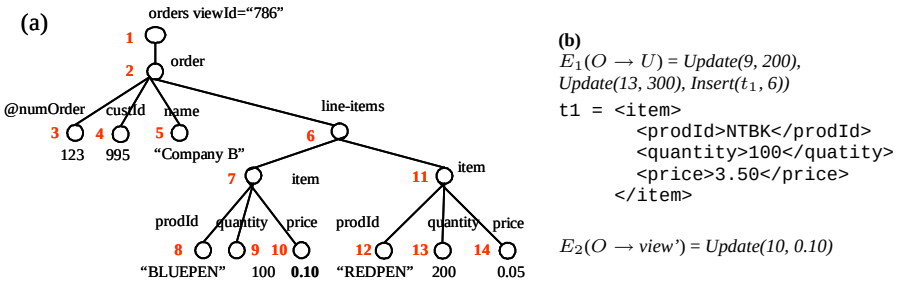

Fig. 5. (a) View' (b) Edit scripts for our example

in the original view. As an example, view' is shown in Figure 5(a). Notice that it reflects the base updates shown in Figure 1(b).

These 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 (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 (Section 4.2), and then notify the user of the result of the updates (Section 4.3).

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 [17], this delta can be expressed by $E(D_1 \rightarrow D_2)$.

We adopt X-Diff [17] 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 [4] 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 [6] and XMLTreeDiff [7] consider ordered models, which does not match the unordered model of our source data (relations).

X-DIFF. According to [17], 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 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)* [17]. 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 5(b) 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*.

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 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). 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 DB primary keys as filters in the path expression. Notice that keys must be kept in the view for it to be updatable [2]. Specifically, for an 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*. 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 $\langle t, \Delta, ref \rangle$.

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

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, we show the translation of an operation of E_1 (Figure 5(b)):

– *Update(9, 200)* $\equiv$ <*modify*, {200}, *orders/order[@numOrder="123" and custId= "995"]/line-item/item[prodId="BLUEPEN"]/quantity*>

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.

4.2 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, if a user increases the quantity of an item in its order, she may not want to proceed with this increase when she knows that the price of the item has increased.

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 (Section 4.1) and sends them to PATAXÓ so it can map them to the underlying relational database.

However, most of the times the views (O and view') will not be equal, which implies in conflicts. A conflict is any update operation that has been issued in the database during the transaction lifetime, and that affects the updates made by the user through the view. We will provide more details on this later on.

In our approach, there are three *operational modes* to deal with conflicts: *restrictive, relaxed* and *super-relaxed* modes. The first one, the *restrictive mode*, no updates are translated when there are differences between the views original and view'. This is a very restrictive approach, where all modifications made over the view are treated as a single atomic transaction.

The second, *relaxed mode*, is a bit less restrictive. 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 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.

Conflict Detection Rules for Relaxed Mode We now present rules for the resolution of conflicts in modifications for the relaxed mode. We leave insertions and deletions for future work.

RULE 1 *(Leaf node within the same starred-element) 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 3 from 100 to 200. This operation can not proceed because it conflicts with the update of node 10 (price of blue pens) in view'.

RULE 2 *(Dependant starred-subtrees) 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.*

This rule captures the dependency between starred subtrees. In the XML view of Figure 3, it is easy to see that each *item* subtree is semantically connected to its parent *order* tree. Thus, rule 2 defines that modifications done in the database that affect the *order* subtree conflicts with modifications to the *item* subtree done through the view.

Notice that in all the above rules, we need to know the correspondence of nodes in views U and *view'*. For example, we need to know that node 12 in the updated view (Figure 3) correspond to node 12 in *view'* (Figure 5(a)). This can be easily done by using a variation of our *merge* algorithm presented later on.

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. In [16], we present the algorithm. The checking is very simple, and once we detect a conflict due to one rule, we do not need to check the other one.

Conflict Detection Rules for Super-Relaxed Mode Finally, the third, less restrictive operational mode is the *super-relaxed* mode. In this mode, we consider a conflict happens only when the update occurs over the same leaf node, or the tuple key has been changed in the database. Formally, we have:

RULE 3 *(Same leaf node) Let* l *be a leaf node in a given XML view* v*. If* l *is modified in the updated view to a value* v_1*, and* l *is modified in view' to a value* v_2*,* $v_1 \neq v_2$*, then the update on node* l *is rejected.*

RULE 4 *(Key node) Let l and k be two leaf nodes in a given XML view v. Let k represent the primary key of the tuple from which l was extracted in the database. If l is modified in the updated view, and k is modified in view', then the update on node l is rejected.*

We consider this a conflict because we use the key value to translate the update. If the key has changed, we can not reach the tuple to update it anymore.

4.3 Notifying the User

In all operational 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. To do so, the system generates a *merge* of the updated data and the current database state. The algorithm starts with the original XML view. Consider $E1$ = $E(O \rightarrow U)$ and $E2$ = $E(O \rightarrow$ *view'*).

1. Take each delete operation u=Delete(x) in $E2$ 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 (we omit it in the remaining definitions). This new element is connected to the parent of x.
2. Take each insert operation u=Insert(T_x, y) in $E2$, insert T_x under y and add a new parent *DB-INSERT* to T_x. Connect the new element as a child of y.
3. Take each modify operation u=Update(x, new-value) in $E2$, add a *DB-MODIFY* element with value *new-value*. Connect it 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 to describe if the update operation was accepted or aborted. Since we are currently detecting conflicts only between modify operations, we are assuming the remaining ones are always accepted.

1. Take each delete operation u=Delete(x) in $E1$, add a new parent *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 $E1$, insert T_x under y and add a new parent *CLIENT-INSERT STATUS="ACCEPT"* to T_x. Connect the new created element as a child of y.
3. Take each modify operation u=Update(x, new-value) in $E1$, add a new element *CLIENT-MODIFY* with value *new-value*. Connect the *CLIENT-MODIFY* element as a child of x. If u is marked in $E1$, then add a *STATUS* attribute to the *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 6. 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.

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

```
<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="ACCEPT">
            <item>
              <prodId>NTBK</prodId>
              <quantity>100</quantity>
              <price>3.50</price>
            </item>
         </pataxo:CLIENT-INSERT>
      </line-items>
   </order>
</orders>
```

Fig. 6. Result of the *merge* algorithm

(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.

5 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 [2] 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 [17]. 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 [2], 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

1. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, San Jose, California, May 1995.
2. V. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, Toronto, Canada, Sept. 2004.
3. V. Braganholo, S. B. Davidson, and C. A. Heuser. PATAXÓ: A framework to allow updates through XML views. *ACM Transactions on Database Systems, TODS*, to appear, 2006.
4. S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, pages 26–37, Tucson, Arizona, May 1997.
5. P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems, TODS*, 19(3):450–491, 1994.
6. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE*, pages 41–52, San Jose, California, Feb. 2002.
7. F. Curbera and D. Epstein. Fast difference and update of XML documents. In *XTech*, San Jose, California, Mar. 1999.
8. U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 8(2):381–416, Sept. 1982.
9. M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in XML. *ACM TODS*, 27(4):438–493, Dec. 2002.
10. J. Foster, M. Greenwald, J. Moore, B. Pierce and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Symposium on Principles of Programming Languages (POPL)*, pages 233–246, Long Beach, CA, USA, 2005. ACM Press.
11. L. Klieb. Distributed disconnected databases. In *Symposium on Applied Computing (SAC)*, pages 322–326, New York, NY, USA, 1996. ACM Press.
12. S. H. Phatak and B. R. Badrinath. Conflict resolution and reconciliation in disconnected databases. In *DEXA*, 1999.
13. B. Pierce, A. Schmitt and M. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, USA, 2003. Superceded by MS-CIS-05-02.
14. J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, Rome, Sept. 2001.
15. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.
16. A. Vargas, V. Braganholo, and C. Heuser. Conflict resolution and difference detection in updates through XML views. Technical Report RP-352, UFRGS, Brazil, Dec. 2005. Available at `www.cos.ufrj.br/~vanessa`.
17. Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for XML documents. In *ICDE*, pages 519–530, India, March 2003.