# Atomicity for XML Databases*

Debmalya Biswas[1], Ashwin Jiwane[2], and Blaise Genest[3]

[1] SAP Research, Vincenz-Priessnitz-Strasse 1, Karlsruhe, Germany
debmalya.biswas@sap.com
[2] Department of Computer Science and Engineering, Indian Institute of Technology, Mumbai, India
ashwinjiwane@cse.iitb.ac.in
[3] IRISA/CNRS, Campus Universitaire de Beaulieu, Rennes, France
genest@crans.org

**Abstract.** With more and more data stored into XML databases, there is a need to provide the same level of failure resilience and robustness that users have come to expect from relational database systems. In this work, we discuss strategies to provide the transactional aspect of atomicity to XML databases. The main contribution of this paper is to propose a novel approach for performing updates-in-place on XML databases, with the undo statements stored in the same high level language as the update statements. Finally, we give experimental results to study the performance/storage trade-off of the updates-in-place strategy (based on our undo proposal) against the deferred updates strategy to providing atomicity.

## 1 Introduction

With more and more data stored into XML databases, there is a need to provide the same level of failure resilience and robustness that users have come to expect from relational database systems. A key ingredient to providing such guarantees is the notion of transactions. Transactions have been around for the last 30 years leading to their stable and efficient implementations in most current commercial relational database systems. Unfortunately, (to the best of our knowledge) there still does not exist a transactional implementation for XML databases. Here, we are talking about native XML databases [1], and not relational databases extended to store XML data [2] that in turn rely on the transactional features of the underlying relational database. The need to be able to update XML however has been widely recognized and addressed by both industry and researchers with the recent publication of the W3C standard XQuery Update Facility 1.0 [3]. Researchers have also started exploring efficient locking protocols for XML [4,5]. In this work, we discuss strategies to provide a transactional implementation for XML databases based on [1,3], with specific focus on the atomicity aspect.

* This work was partially done while the first and second authors were at IRISA/INRIA, France. This work is supported by the CREATE ACTIVEDOC, DOCFLOW and SecureSCM projects.

Transactions [6] provide an abstraction for a group of operations having the following properties: A (Atomicity), C (Consistency), I (Isolation), D (Durability). Atomicity refers to the property that either all the operations in a transaction are executed, or none. There are usually two strategies for performing updates in an atomic fashion:

– Deferred Updates (UD): In this strategy, each transaction $T$ has a corresponding private workspace $W_T$. For each update operation of $T$, a copy of the data item is created in $W_T$, and the update applied on the local copy. Upon commit of $T$, the current values of the data items in $W_T$ need to be reflected atomically to the actual database. As obvious, abortion is very simple and can be achieved by simply purging $W_T$.
– Updates-in-Place (UIP): Here, transactional updates are applied as and when they occur. However, the disadvantage of UIP is the added complexity in the abortion process. To provide atomicity in such a scenario, the "before images" of all updated data items is usually maintained in a log, which can then be used to undo the updates if required.

In this work, we study both approaches for providing atomicity to XML databases. The main contribution of this paper is to propose a novel approach for providing UIP on XML databases. Rather than maintaining the "before images" *data*, our proposal stores undo *operations* written in the high level query/update language. Basically, for each update, it generates the undo statements dynamically (at run-time) that can be used to undo the effects of that update. The transformation rules to dynamically generate the undo operations are presented in Section 3.

The performance trade-off between UD and UIP depends upon the additional overhead of creating and writing copies in UD, and depends upon the time taken to create the undo operations and perform them in the event of a failure in UIP. The amount of storage required to store the undo data is also an area of concern for current database implementations. Long running transactions performing large number of updates often have to be aborted due to insufficient disk space (e.g., the "ORA: Snapshot Too Old" error). Thus, the undo storage requirements of both strategies are important. In our case, the UD strategy in particular leads to storage/performance trade-offs. Recall that for each update, the UD strategy first creates copies of the data items (if they do not already exist in its private workspace), and then performs the updates on the copies. The additional create step per update can clearly be avoided if we create a copy of the whole XML document for the first update on a document, then any subsequent updates on nodes of that document would not need the additional create step. Obviously, this is not efficient from a storage perspective as it would lead to storage redundancy unless all the nodes of a document are updated (as part of one transaction).

We have implemented both the UD and UIP strategies for XML databases, and provide experimental results with respect to the performance/storage trade-off between the two strategies for the issues identified above in Section 4.

Note that this approach of using undo statements to preserve atomicity is in line with using compensation [9] to semantically undo the effects of a transaction. Here, it helps to recall that compensation is not equivalent to the traditional database "undo", rather it is another forward operation that moves the system to an acceptable state on failure. While compensation mechanisms have been accepted and are available in high level languages (e.g., compensation handlers in BPEL [8]), they have not been studied for database updates (at least, not explicitly). Implicitly, when we say that a bank withdrawal can be compensated by a deposit operation, it corresponds to the effects of an update SQL being compensated by another update SQL on the accounts table.

## 2 XML Update Syntax

In this section, we give a brief introduction to the XQuery Update Facility (XUpdate) [3] for performing XML updates. XUpdate adds five new kinds of expressions, called insert, delete, replace, rename, and transform expressions, to the basic XQuery model [7]. For simplicity, we only focus on the insert, delete and replace expressions, also referred to as the update expressions in general. The main differences between XUpdate and SQL, apart from the hierarchical nature of XML, arises from the significance of a node's location in the XML document (XML documents are basically ordered trees.). For example, let us consider the syntax of the XUpdate insert expression:

*InsertExpr ::= "insert" ("node" | "nodes") SourceExpr TargetChoice TargetExpr*
*TargetChoice ::= (("as" ("first" | "last"))? "into") | "after" | "before"*

An insert expression is an updating expression that inserts copies of zero or more source nodes (given by *SourceExpr*) into a designated position with respect to a target node (given by *TargetExpr*). The relative insert position is given by *TargetChoice* having the following semantics: If *before* (or *after*) is specified, then the inserted nodes become the preceding (or following) siblings of the target node. If *as first into* (or *as last into*) is specified, then the inserted nodes become the first (or last) children of the target node. For the detailed specifications of *InsertExpr*, *DeleteExpr* and *ReplaceExpr*, the interested reader is referred to [3].

The evaluation of an update expression results in a pending update list of update primitives, which represents node state changes that have not yet been applied. For example, if *as first into* is specified in the given insert expression, then the pending update list would consist of update primitives of the form:

*insertIntoAsFirst($target, $clist)*

The effects of the above primitive can be interpreted as: Insert *$clist* nodes as the leftmost children of the *$target* node. Note that the node operands of an update primitive are represented by their node identifiers.

The pending update lists generated by evaluation of different update expressions can be merged using the primitive *mergeUpdates*. Update primitives are held on pending update lists until they are made effective by an *applyUpdates* primitive.

We consider a pending update list consisting of a sequence of update (insert/delete/replace) primitives as a transactional unit, with the *applyUpdates* primitive acting as the corresponding Commit operation. The objective then is to ensure that the update primitives in a pending list execute as an atomic unit, i.e. either all the update primitives are applied or none. In the next section, we show how to generate the corresponding undo primitive that can be used to undo the effects of an update primitive in the event of a failure.

## 3 XML Undo Primitives

The underlying intuition of undo primitives is that the effects of an insert (delete) primitive can be canceled by the subsequent execution of a delete (insert) primitive. For example, for the update primitive:

*insertBefore (t,* $\{n_1, n_2, n_3\}$*)*

its undo primitive(s) would be:

*delete(*$n_1$*)*
*delete(*$n_2$*)*
*delete(*$n_3$*)*

where $t$, $n_1$, $n_2$, $n_3$ refer to node identifiers. Note that an update primitive may lead to more than one undo primitive, and vice versa. Basically, for a given pending update list, as its update primitives are processed one by one, we assume that the same processor can use our mechanism to simultaneously generate (and store) the respective undo primitives.

We give the undo generation rules of the different update primitives in the sequel.

### 3.1 Insert

We start with the insert primitive.

*insertBefore ($target as node(), $content as node()+)*

Let $\$content = \{n_1, \cdots, n_m\}$. Then, its undo primitives are:

*delete(*$n_m$*)*, $\cdots$, *delete(*$n_1$*)*

The undo primitives for the other insert primitives: *insertAfter*, *insertInto*, *insertIntoAsFirst* and *insertIntoAsLast*, can be generated analogously.

### 3.2 Delete

The undo primitive generation of the delete primitive is slightly more complicated. This is because of the lack of position specifier in the delete primitive:

*delete($target as node())*

That is, we do not know the position of the node to be deleted in the XML document, and consequently do not know where to re-insert it (if required) to undo the effects of the delete. If we assume some correlation between the node identifier and its position in the document, then we can infer the position of the node from its identifier. However, [3] does not impose any such restrictions on the node identifier generation scheme, and neither do we assume the same here. In the absence of such correlation, we first need to get the position details of the node to be deleted before generating the corresponding undo primitive. There are of course several ways of determining the relative position of the $target node in the document, and the idea is to minimize the number of function calls needed. An approach to generate the undo primitive based on the relative position of a sibling node would be as follows:

```
$sibling = getNextSibling($target)
insertBefore ($sibling, $target)
```

The variant of the above approach based on the preceding sibling would be as follows:

```
$sibling = getPreceedingSibling($target)
insertAfter ($sibling, $target)
```

While the above approaches require only a single additional function call, they would not work if the $target node does not have any siblings (is the only child). Then, we need to get the position of $target relative to its parent node.

```
$parent = getParent($target)
insertInto ($parent, $target)
```

Note that in the worst case (no siblings), the above approach leads to three function calls, i.e. *getNextSibling()*, followed by *getPreceedingSibling()*, and finally *getParent()*. An alternate approach that also needs two additional function calls would be as follows:

```
$parent = getParent($target)
$listChildren = getChildren($parent)
Let $i$ be the position of $target in $listChildren, and $n_i$ refer to the node identifier at position $i$ in $listChildren.
if ($i = 0$)
    then insertInto ($parent, $target)
    else insertAfter ($n_{i-1}$, $target)
```

Clearly, which approach is better depends on the underlying XML data characteristics, and would be difficult to predict in advance.

We can offset the time taken by the additional functional calls by doing some pre-processing. Note that on *applyUpdates()*, the update primitives in a pending update list are processed in the following order [3]: insert, followed by replace, finally followed by delete primitives. As the delete primitives are processed at the end, the positions of the delete primitives can be pre-determined in parallel while processing of the preceding insert and replace primitives.

### 3.3 Replace

The replace primitive is given below:

*replaceNode ($target as node(), $replacement as node()*)*

The semantics of the replace primitive is to replace the *$target* node by the *$replacement* node(s). A replace primitive can be implemented as a combination of insert and delete primitives as follows:

*replaceNode ($target, $replacement)*
$\Leftrightarrow$
*insertAfter ($target, $replacement), delete ($target)*

The undo primitives for replace can then be constructed as follows: Let $\$replacement = \{n_1, \cdots, n_m\}$.

*insertAfter* ($n_m$, *$target)*
*delete*($n_m$), $\cdots$, *delete*($n_1$)

The first *insertAfter* undo primitive relies on the semantics that insertion of a list of nodes preserves their original source order, i.e. $n_2$ is the right sibling of $n_1$, $\cdots$, $n_m$ is the rightmost sibling node to be inserted. Note that here we do not need the additional function calls to generate the undo primitive of delete, as is needed for standalone delete primitives (Section 3.2).

## 4 Experimental Analysis: Performance and Storage

The experiments were performed on the Active XML (AXML) [10] infrastructure. AXML provides a P2P based distributed XML repository. The XML repositories in AXML are implemented based on the *Exists* XML database. *Exists* currently does not provide any transactional features, at least not at the user level. We have extended *Exists* to implement the transactional aspects discussed in this paper.

As mentioned earlier, the main factors affecting the performance trade-off between UD and UIP (implemented based on our proposal) can be summarized as follows: the additional overhead of creating and writing copies in UD versus the additional time needed to perform the undo operations in the event of a failure. Fig. 1 gives some comparison results.

Clearly, if there are no failures, UIP is the fastest (no undoing is required, while the time taken to create the undo operations is negligible). In the event

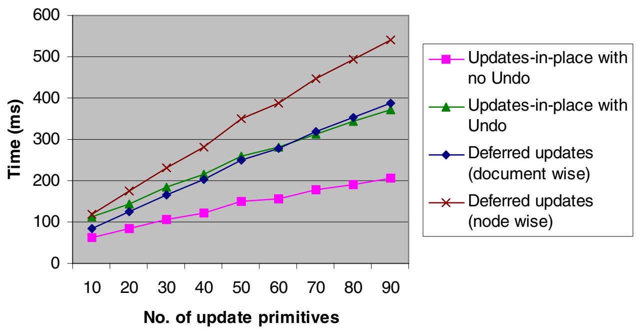
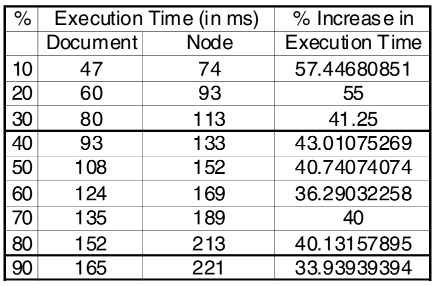


**Fig. 1.** Execution Time vs. Number of update primitives per transaction



| % | Execution Time (in ms) | | % Increase in |
|---|---|---|---|
| | Document | Node | Execution Time |
| 10 | 47 | 74 | 57.44680851 |
| 20 | 60 | 93 | 55 |
| 30 | 80 | 113 | 41.25 |
| 40 | 93 | 133 | 43.01075269 |
| 50 | 108 | 152 | 40.74074074 |
| 60 | 124 | 169 | 36.29032258 |
| 70 | 135 | 189 | 40 |
| 80 | 152 | 213 | 40.13157895 |
| 90 | 165 | 221 | 33.93939394 |

**Fig. 2.** Performance/storage trade-off

of failures, undos are required. When the number of operations is small, (document wise) UD fares slightly better. However, as the number of operations in a transaction increases, this advantage seems to fade away. That is, in mean value considering a reasonable number of failures, we expect UIP based on our proposal to perform better than UD, and in particular for large transactions.

Fig. 2 shows the storage/performance trade-off between creating copies of the whole document against that of the specific affected nodes at a time in UD. If 10% of the nodes of a document are updated, then creating and updating copies node wise takes 57% more time than doing it document wise. On the other hand, the storage space is ten times less than that required by the document wise strategy. On the other hand, if 90% of the the document nodes are updated, then the nodewise strategy only needs 33% more time than the document wise strategy, while saving only 10% of storage space. While it is expected for the overall execution time of the node wise strategy to increase as the percentage of document nodes updated increases, the interesting result is that the percentage difference in execution time actually decreases (from 57% to 34%) as the percentage node updation increases (from 10% to 90%). We infer that this decrease is because as the percentage of nodes updated in a document increases, the probability of the same nodes being updated more than once also increases (for whom, the overhead copy creation time is saved). Compared to UIP based on our proposal, the storage space needed for UIP is very close to that needed for node-wise UD strategy. That is, it seems that UIP is better both in terms of storage space and execution time. The only drawback is obviously its more complex implementation.

## 5 Conclusions

Transactional implementation for XML databases are still in their infancy, with the recent release of the W3C specification to perform XML updates. With updates, comes the natural requirement to be failure resilient, and hence transactions. In this work, we considered the two main approaches for performing updates in an atomic fashion: UD and UIP. We proposed a novel approach for providing UIP on XML data, where the undo data required to perform rollback in the event of a failure/abort is stored in the same high level language as the update statements. This allows for a lightweight implementation (without going into the disk internal details) while providing comparable performance, as verified by experimental results.

## References

1. Open Source Native XML Database, `http://exist.sourceforge.net/`
2. XML Database Benchmark: Transaction Processing over XML (TPoX), `http://tpox.sourceforge.net/`
3. XQuery Update Facility 1.0 Specification, `http://www.w3.org/TR/xquery-update-10/`
4. Dekeyser, S., Hidders, J., Paredaens, J.: A Transactional Model for XML Databases. World Wide Web 7(1), 29–57 (2002)
5. Haustein, M.P., Härder, T.: A Transactional Model for XML Databases. J. Data Knowledge Engineering 61(3), 500–523 (2007)
6. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control. Morgan Kaufmann Publishers, San Francisco (2001)
7. XQuery 1.0: An XML Query Language Specification, `http://www.w3.org/TR/xquery/`
8. Business Process Execution Language for Web Services Specification, `http://www.ibm.com/developerworks/library/specification/ws-bpel/`
9. Biswas, D.: Compensation in the World of Web Services Composition. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 69–80. Springer, Heidelberg (2005)
10. Active XML, `http://activexml.net`