Change-Centric Management of Versions in an XML Warehouse*

Amélie Marian
Columbia University / N.Y. USA
amelie@cs.columbia.edu

Serge Abiteboul, Grégory Cobéna
Verso INRIA, Rocquencourt France
<firstname>.<lastname>@inria.fr

Laurent Mignet
Verso INRIA - Vertigo CNAM, France
mignet@cnam.fr

Abstract:

We present a change-centric method to manage versions in a Web WareHouse of XML data. The starting points is a sequence of snapshots of XML documents we obtain from the web. By running a *diff* algorithm, we compute the changes between two consecutive versions. We then represent the sequence using a novel representation of changes based on *completed* deltas and persistent identifiers. We present the foundations of the logical representation and some aspects of the physical storage policy.

The work presented here was developed in the context of the Xyleme project of massive XML warehouse for XML data from the Web. It has been implemented and tested. We briefly discuss the implementation.

Keywords: XML, Delta, Version, Change Control, Warehouse.

1 Introduction

Data publication on the web is constantly increasing. Users are often not only interested in the current values of documents and query answers but also in changes. They want to see changes as information that can be used to learn about the evolution of the web. We present a change-centric representation of changes in a Web Warehouse of XML data. By change-centric, we mean that we focus on deltas, i.e., the changes themselves, as opposed to other approaches that might focus on snapshots or object history. We introduce a logical representation of changes based on completed deltas and an efficient storage policy for it. Finally, we briefly discuss some aspects of the implementation.

XML is becoming the new standard for semistructured data exchange over the Internet [22, 1]. This work is part of the Xyleme project [24, 25] that is studying and building a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing massive volume of XML data found on the Web. In the present paper, we consider the issue of version management support for the Xyleme system.

In such a system, the management of versions is essential for a number of reasons ranging from traditional support for temporal queries, to more specific ones such as index maintenance or support for query subscriptions. Motivations are considered in Section 2.

The system acquires XML data from the Web and maintains it up-to-date [16]. Thus, for a particular document, a sequence of snapshots [2] is obtained. Similarly, continuous queries (queries that are evaluated regularly) produce sequences of query answers, i.e., sequences of snapshots of an XML document. The modifications that occurred between time t_{i-1} and time t_i can be computed using a *diff* algorithm. We developed our own *diff* algorithm adapted to XML data and our specific requirements. The algorithm is presented in [9]. The sequence of snapshots and the results of *diff* between consecutive ones form the basis of our management of versions. Note that the system does not have a real time vision of the data. The time we fetch a document may differ from the time of the last update as posted in the header of the document that itself may differ from the actual time of this update. Furthermore, we may typically "miss" some updates. This has to be accepted in the current Web context. In the paper, we ignore this issue; and when we mention the time of a version, we mean the time the system acquired this version.

The main issues that are addressed in the paper are the choices of a logical representation and that of a storage policy for versions adapted to the control of changes. Our logical representation is based on *deltas* in the style of [11]. Its two main components are:

1. Persistent identifiers. In our logical representation, all XML nodes are assigned a persistent identifier, that we call XID for Xyleme ID. A main role of the *diff* algorithm is to assign these identifiers.

Work supported in part by R.N.R.T.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

The use of persistent identifiers is essential to describe changes and also query changes efficiently.

2. Completed deltas. Simple deltas are lossy and cannot be inverted. For instance, updates typically ignore the old value. We introduce *completed deltas* which are deltas containing additional information that are reminiscent of traditional ways of representing logs in database systems. Completed deltas can be inverted and composed.

Our physical storage policy is based on storing the current version of the document, an *XID-map* to handle XIDs and a single XML document containing all *forward completed deltas*. The XID-map is a novel concept that allows to attach persistent XIDs to every node in a storage-efficient manner. A forward completed delta describes the changes between two consecutive versions going from the old one to the new one, and also contains information for the opposite direction. The choice of this storage policy was motivated by an analysis of the Xyleme change-management requirements which we discuss in the paper. In particular, this storage policy presents the advantage that it is possible to install a new version with almost no update to objects already in the store. A disadvantage is that we store redundant information. We present a delta compression technique to periodically recover space. Others works on XML versionning have been proposed by [7, 8].

As previously mentioned, the present work has been realized in the context of the Xyleme project. The versioning system is implemented (like the rest of the system) in C++, under Linux, with Corba for communications. We present some experimentation.

We consider motivations in Section 2. In Section 3, we present the logical representation based on XIDs and (completed) deltas. We discuss our storage policy in Section 4. The last section is a conclusion. Due to space limitations, proofs and details of algorithms are omitted.

2 Motivations

In this section, we discuss motivations for using deltas in the context of Xyleme. Most of these motivations clearly apply to a much larger setting. We also consider specific requirements of the system that guided the choices of the logical and physical representations. Further motivations for these choices are given in the following sections.

Deltas serve many purposes in such an XML warehouse environment:

Versions. We may want to version a particular document, (part of) a Web site, or the results of a continuous query. This is the most standard use of versions, namely recording history.

Querying the past. One might want to ask a query about the past, e.g., ask for the value of some particular element at some previous time. Also, one might want to query changes, e.g., ask for the list of all items recently introduced in a catalog.

Learning about changes. A user may request some XML documents or some XML query result, say V_i at time i. Later, the user may request a new version, say V_{now}. The delta from time i to *now* is a possible description of the changes. It allows to update the old version V_i and also to explain the changes to the user. This is in the spirit, for instance, of the Information and Content Exchange, ICE [23, 12, 14].

Monitoring changes. We implemented a subscription system [17]. We want to be able to detect changes of interest in XML documents, e.g., that a new product has been added to a particular catalogue. To do that, at the time we obtain a new version of some data, we compute the *delta* and verify if some of the changes that have been detected are relevant to some subscriptions. Related works on subscription systems that use filtering tools for information dissemination have been presented, e.g., in [26, 4].

Indexing In Xyleme, we maintain a full-text index over a large volume of XML documents. To support queries using the structure of data, we need to store structural information for every indexed word of the document [3]. We are considering the possibility to use the delta to maintain such indexes.

These are motivations for deltas in the context of Xyleme. Obviously, they apply to a much wider context. There are other possible uses of deltas. To see one, consider resource sharing. Different users may be simultaneously updating (off-line) the same XML document. Deltas turn out to be useful to synchronize the respective versions. They describe the modifications and facilitate the detection of potential conflicts. This is in the style of, e.g., CVS [10].

In our context, the starting point is a sequence of snapshots of an XML document that is obtained from the Web (or computed in the case of continuous queries). Each new version is processed with the previous one with our *diff* algorithm to match nodes in the two versions, i.e., "identify" them. This also allows us to compute the delta between the two versions. For each versioned documents, we store the last version and *the sequence of completed deltas*. For each i, the completed delta $\overline{\Delta}_{i,i+1}$ describes how to go from one version V_i to the next one V_{i+1}, and from V_{i+1} to V_i. The reasons for using completed deltas and not "simple" deltas will be explained at length. In short, this is because the computation of some operations such as the composition of deltas and the inversion of deltas is simply not possible with simple deltas without costly reconstructions of versions.

Thus, at each point in time, we store the last version, and the sequence of completed deltas from the origin. There are many possible alternatives for storing the history of the data. The choice of a logical representation of change and of a physical one

clearly depends on the pattern of use of the system. In our choice, we assumed the general requirements of Xyleme. In Xyleme, it is considered critical to be able to obtain the changes between the data at time t_i and its current value, and query these changes. Other aspects such as rebuilding the document as it was at some time t_i should be supported although they are not considered as critical in terms of performance. The most critical requirement is that we want to be able to install a new version very efficiently. More precisely, we want to be able to install new versions roughly at the speed we can obtain and index data. We will see how this impacts on the choices we made.

3 The Model

We introduce a simplified model for XML documents that is sufficient to discuss changes. We then consider three models of changes. First, we consider edit-scripts that are *sequences* of basic operations. Then we introduce deltas that are *sets* of basic operations and present the advantage of being a more global description of changes. Finally, we propose the notion of completed deltas that overcomes certain shortcomings of deltas. Completed deltas are in some sense connected to logs considered in database systems. We show nice mathematical and practical aspects of completed deltas.

The starting point of our work is a sequence $V_1, ..., V_n$ of snapshots, i.e., the versions of some XML data at time $t_1, \ldots, t_n$. A problem with respect to change is that there is no means to detect that two nodes in consecutive snapshots correspond to the same entity. To represent changes in a natural manner, we need to be able to track XML nodes through time. For this, we use persistent identifiers that we call *Xyleme identifiers, XIDs*. There are many motivations for XIDs. For instance:

- Suppose the price of a product has been modified to a new value v. This change may be easily described by $update(n, v)$ where n is the XID of the text node corresponding to this product price (e.g., $update(8, 150)$).
- Suppose we want to reconstruct the history of a product or its description at a certain date. If we have an identifier for the product, it is easy to obtain such an information using an appropriate indexing mechanism.

Since nodes in documents found on the Web in general do not have identifiers, we have to provide these identifiers. To do that, we designed and implemented a *diff* algorithm that is described in [9]. Thus we assume that the nodes in the various versions come equipped with XIDs. See Figure 1 where only some matchings are shown. XIDs considered here form a *logical* concept that can be used to denote an XML node in a persistent manner. We will discuss particular implementations of XIDs further.

The basis for our representation of changes are trees where all nodes have identifiers. For our presentation, we use a simplified model that is sufficient to describe the main aspects of changes. In the implementation, we of course deal with the complete XML model. Formally, we will assume that XIDs are from the set $\mathcal{N}$ of integers. Values are from a set $\mathcal{Q}$, e.g. the set of strings. The simplified model is as follows:

Definition: An *XML tree* is a pair $\langle t, \nu \rangle$ where (i) t is a finite ordered tree with nodes from $\mathcal{N}$; and (ii) ν, the *value mapping*, assigns a value (possibly null) in $\mathcal{Q}$ to each node in t.

In the complete model, we need to distinguish between text, element and attribute nodes because they behave slightly differently for some of the change operations we study. The value of an element node is its label, whereas for a text node, it is a PCDATA. Note that, for instance, a text node cannot have children and has a fixed label. We will mention some differences further on.

Edit-scripts

One can modify an XML-tree T using the following basic operations:

1. *delete(m)* that deletes the XML tree rooted in node m where m is not the root of T.
2. *insert(n,k,T')* that inserts the XML tree T' as the k-th child of n.
3. *move(n,k,m)* that moves the XML tree rooted in node m to be the k-th child of n.
4. *update(m,v)* that changes the value of a node m to v.

The resulting tree (with identifiers) is defined in the obvious way. There are clearly consistency conditions. E.g., for the insertion, T must have a node n with at least $k-1$ children and T, T' should not have XIDs in common. If an operation ω is *consistent for* a tree T, $\omega(T)$ is defined in the obvious manner. An *edit-script* is a sequence of such operations. The script $\omega_1; ...; \omega_n$ is consistent for a tree T is for each i, $\omega_{i+1}; ...; \omega_n$ is consistent for $\omega_1; ...; \omega_i(T)$ where the result of applying a script to a tree is defined in an obvious way.

Remark 3.1 This model of changes is rather simplistic. Our implementation does consider a larger set of basic operations to handle the general XML model, e.g., attribute operations and label updates. One could also consider more sophisticated update operations, e.g., the means to insert a string in position k of an existing string, or to increment an integer. Although these would be interesting to consider from a practical viewpoint, they would not change the framework in any substantial manner, so they will be ignored here. □

Consider a tree T of root 0, with children 1,2,3 with values "a", "b", "c" and the following edit-scripts:

- update(3,"d");move(0,2,3);delete(1);update(3,"e")

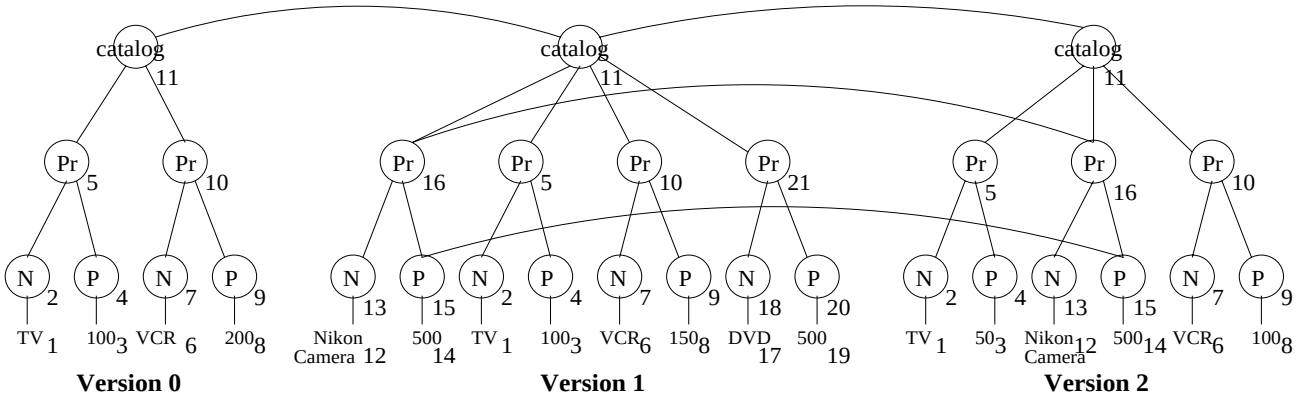


Figure 1: A sequence of snapshots with identified nodes

$\mathbf{\Delta_{1,2}}$ (Forward Delta)	$\mathbf{\Delta_{2,1}}$ (Backward Delta)	$\overline{\mathbf{\Delta}}_{\mathbf{1,2}}$ (Completed Delta)
delete (21)	insert (11, 4, B)	$\overline{delete}$ (11, 4, B)
move (11, 2, 16)	move (11, 1, 16)	$\overline{move}$ (11, 2, 16, 11, 1)
update (3, 50)	update (3, 100)	$\overline{update}$ (3, 50, 100)
update(8, 100)	update (8, 150)	$\overline{update}$ (8, 100, 150)

Table 1: Examples of deltas

- delete(1);move(0,1,3);update(3,"e")

These two scripts have the same effect on T. After applying each of them, node 3 becomes the first child of 0 and its value is "*e*". The second one is more "interesting" in that it provides the final value and position of node 3.

Delta

As we saw, there may be many ways to describe the changes between two consecutive versions using edit scripts even when nodes have identifiers. An alternative is to use *deltas*. A delta consists of a *set* of basic operations : delete (D), update (U), insert (I) and move(M). The focus is on avoiding to specify an order of execution as in an edit-script. For example, *deltas* operations use positions that refer to one of the two document's snapshot (e.g. V_1 for delete, V_2 for insert). Given two snapshots of a document with identified nodes, there is a unique delta describing operations that transform one snapshots into the other. Given two trees T, T', a *delta* Δ from T to T' is a set of operations satisfying the following properties:

deletes for every node n that is in T and not in T' and whose parent is in T', then $D(n)$ is in Δ. Δ contains no other deletes.

inserts for every node n that is in T' and not in T and whose parent is in T, then $I(m, k, T_1)$ is in Δ where m is the parent of n, k the rank (position in the parent) of n, and T_1 is the tree rooted at n with all nodes from T pruned. Δ contains no other inserts.

updates for every node n whose value v in T' is not that of n in T, then $U(n, v)$ is in Δ. Δ contains no other updates.

moves Δ also contains <u>*some*</u> moves $M(n, k, m)$ where m is in T' the k-th child of n and m occurs in T with a different parent or position.

absent nodes for each node n belonging to both T and T', the set of children of n that were neither deleted, nor inserted, nor moved, are the same in T and T' and they are in the same order.

In a delta, if a node is a third argument of a move, it is called a m-node. The first argument of a delete, (resp. update), is called d-node, (resp. u-node). The root of an inserted tree is called an i-node. Note that a node is at most one between d-node, i-node, and m-node.

Operations in a delta Δ represent the set of changes needed to go from some instance T to some T'. Operations in a delta are not ordered in the sense that deltas do not provide any explicit order.

On the other hand, applying the delta on the snapshot of the document requires to execute operations one-by-one, and thus to order them. For example, parents node are inserted before children. For insert operations with the same parent, our solution, simple and efficient, consist in starting with the lower positions. Thus, when the insert operation is executed, the actual insert position is equal to the position of the node in the final snapshot, as described in the operation's parameters. This order is reversed for delete operations.

Theorem 3.2 Let T be a tree and Δ a set of operations, then there exist at most one T' such that Δ is a delta from T to T'.

Given T, Δ, $\Delta(T)$ is the tree (if it exists) such that Δ is a delta from T to this tree; otherwise $\Delta(T)$ is undefined. Note also that it is relatively easy given T and Δ to compute $\Delta(T)$.

Let Δ_1, Δ_2 be two deltas from T to T'. By definition, they have the same inserts, deletes and updates operations. They may differ in the moves only. A Δ from T to T' is said to be *minimum* if there is no strict subset of Δ that is also a delta from T to T'. Clearly, a delta can be "minimized" by removing redundant moves. We next consider an interesting observations on the absent nodes. For each tree T and node n in T, we call $children(T, n)$ the word consisting of the list of XIDs of the children of n. Then we have:

Proposition 3.3 Let Δ be a minimum delta from T to T' and n a node in T. Then the absent children of n, i.e., the children that are neither d-nodes, i-nodes or m-nodes, is a largest common subsequence for $children(T, n)$ and $children(T', n)$.

Thus given T and T', the choice of a Δ from T to T' is specified by the choice of such a largest common subsequence for each node common to T and T'.

Deltas present severe shortcomings from an information viewpoint. A shortcoming for deltas is that it is not possible from a delta to construct an edit-script (without using the original instance). Also, given Δ_1 that applies to some T and Δ_2 that applies to $\Delta_1(T)$, one cannot compute from Δ_1 and Δ_2, a delta that would correspond to their composition without using T. Similarly, deltas cannot be inverted, i.e., given a Δ that transforms some T into T', one cannot compute Δ' that would transform T' into T without using T. These are, we believe, fundamental reasons why database logs often record more information than just deltas.

This leads to introduce *completed deltas* that will be at the core of our representation of sequences of versions of a document.

The Group of Completed Deltas

To be able to compose deltas (without using the instance) and invert them, we introduce "completed" deltas. In completed deltas, we keep, for instance, the deleted tree in case of a deletion. In some sense, a completed delta $\overline{\Delta}$ contains how to transform a tree T into some tree T' and how to go back from T' to T. The operations in completed deltas are as follows:

1. $\overline{delete}(n, k, T_1)$ that deletes the XML tree T_1 whose root is the k-th child of node n.
2. $\overline{update}(n, v, ov)$ where ov is the old value.
3. $\overline{insert}(n, k, T_1)$ that inserts the XML tree T_1 as a k-the child of node n.
4. $\overline{move}(n,k,m,p,q)$ that moves the XML tree rooted in node m q-th child of p to be the k-th child of n.

Definition: A set $\overline{\Delta}$ of these operations is a *completed delta* if there exist T, T' such that $\overline{\Delta}$ is the set of operations that transforms T to T'.

Completed deltas (together with edit-scripts of completed basic operations) form a sound basis for capturing changes in trees.

Completed delta operator Let $\overline{\Delta}$ be a completed delta from T to T'. We can can easily compute a simple *forward* delta by ignoring some information. Let us denote it by Δ_s. By definition, we will let $\overline{\Delta}(T) = \Delta_s(T)$ for each T.

Composition Composition can be defined on completed deltas. One can easily obtain the lists of inserted, deleted, moved and updated nodes. The values of the updates and the parent of inserted/deleted nodes are also easy to maintain. The main difficulty is to update the positions of *insert(delete)* operations in the first(second) delta to link to positions in the last(first) snapshot of the document. For this, we do for each node some book-keeping within the children's position.

Inverse Given a completed delta $\overline{\Delta}$, let us call $\overline{\Delta}^{-1}$ the completed delta obtained by exchanging inserts and deletes, the old/new values for updates and permuting arguments of moves. Observe that for each T:

$$\overline{\Delta}^{-1}(\overline{\Delta}(T)) = T.$$

Identity Finally, let $\Delta_0 = \emptyset$ (the empty set of completed operations). Then for each $\overline{\Delta}$, we have $[\overline{\Delta}; \Delta_0] = [\Delta_0; \overline{\Delta}] = \overline{\Delta}$.

We are now ready to state:

Theorem 3.4 Completed deltas with the composition operation form a group.

Consider a start instance V_0 and a sequence of completed deltas $\overline{\Delta}_{i,i+1}$ such that, for each i, $\overline{\Delta}_{i,i+1}$ is consistent with $V_i = \overline{\Delta}_{1,2}; ...; \overline{\Delta}_{i-1,i}(V_0)$. Then we have:

$$\begin{array}{ll} V_j = \overline{\Delta}_{i,i+1}; ...; \overline{\Delta}_{j-1,j}(V_i) & i < j \\ V_j = \overline{\Delta}^{-1}_{i-1,i}; ...; \overline{\Delta}^{-1}_{j,j+1}(V_i) & i > j \\ V_j = \overline{\Delta}^{-1}_{n-1,n}; ...; \overline{\Delta}^{-1}_{j,j+1}(V_n) & \text{for each } j. \end{array}$$

Thus from V_n and the sequence of completed deltas, we can reconstruct all possible versions of the document. This is in a nutshell the proof of correctness of our storage policy since we are storing V_n and $\overline{\Delta}_{i,i+1}$ for each i.

Remark 3.5 We mentioned that it is not possible to transform edit-scripts into deltas and conversely (without access to the original instance). This can be achieved for completed deltas. Let $\overline{\Delta}$ be a completed delta. To transform it into a script over completed operations, we specify an ordering of the inserts and one of the deletes. It turns out to be useful to "split" a move into an insert and a delete operation. One can

```
<!-- sequence of changes       -->
<completed-delta sequence>
<!--change between 2 versions -->
  <completed-delta t="2001/02/12/12:018">
    <update n="234" ov="Versions"
       v="Change-Based Versions"/>
<!-- inserted tree cannot      -->
<!-- be an attribute-->
    <insert n="5" k="3">
      <T><ack status="no show">
      Work supported by R.N.R.T.</ack></T>
    </insert>
  </completed-delta>
  <completed-delta t="2001/02/13/14:15:18">
    <update n="234"
       ov="Change-Based Versions"
       vn="Change-Centric Versions"/>
    <move n="4" k="4" m="345" p="5" q="3"/>
<!-- attribute update          -->
    <update-att n="445" l="status"
       v="show" ov="no show"/>
  </completed-delta>
</completed-delta sequence>
```

Figure 2: A Sequence of Completed Deltas in XML

thus obtain an equivalent script that uses inserts and deletes only. We can then use rewrite rules based on algebraic properties of updates to rewrite the sequence into an equivalent sequence that will bring together inserts and deletes corresponding to the same move operation. For instance, such a rewrite rule is:

$$I(m,l,T); I(m,k,T') \rightarrow \\ I(m,k-1,T'); I(m,l,T) \text{ if } k > l$$

Finally, an insert and a delete corresponding to a move that are consecutive in the edit script can be replaced by a move operation. □

4 Versioning XML Documents

In this section, we first present our choice of a storage strategy. We describe various alternatives and motivate the strategy we chose. We then discuss the management of XIDs, the identifiers of nodes. Finally, we discuss in more details some implementation aspects of the storage.

4.1 Physical organization

As already mentioned, we store the last version (the current one) in the repository as well as an XML document containing the *sequence of forward completed deltas*, i.e., $(\overline{\Delta}_{i,i+1})$. See Figure 2. We also store the *XID-map* of the current version that provides the means to obtain the XIDs of the current nodes. This suffices to reconstruct the sequence of snapshots. Since we use completed deltas $(\overline{\Delta})$, no information is lost.

We use a native XML repository that stores XML data as tree [13]. We also represent changes as XML trees which facilitates querying them and sending them to clients. Changes described in XML may seem quite verbose storage-wise (see Figure 2). However, a lot of the redundancy is introduced by the tags, such as *insert, delete* etc. The repository we use represents such tags as integers, so they are not repeated in the store. Moreover, the crux of our technique is that to check in a new version mainly consists of adding new data to the store (V_{now} and $\Delta_{now-1,now}$). This is typically faster (in the repository we use as well as in many repositories) than updating data in place. This is an important aspect of the choice of a storage policy. We next compare it to 3 alternative representations. Others physical representations are described in [8, 19].

Storing 1st + last versions + forward deltas. The computation of $\Delta_{i,now}$ is slightly better with this storage than with completed deltas because simple deltas use less space. So, typically, fewer disk pages will have to be loaded to compute $\Delta_{i,now}$. The main issue is that we are storing two versions of each document. Another drawback is that getting a recent version V_i may be very costly since we have to start from V_0.

Storing last version + backwards deltas. The computation of V_i for some recent i is rather efficient. This solution [20] saves some space compared to the previous one since only one complete version is stored. It is also less space consuming that the method we chose since simple deltas are typically smaller than completed ones. The main drawback is for the computation of $\Delta_{i,now}$. Since simple deltas cannot be inverted (see above), the information missing from the delta may have to be found in V_{now}, which requires loading and processing V_{now} to obtain $\Delta_{i,now}$.

Storing a history. We could imagine storing a history in the style of DOEM [6]. This is more in the spirit of typical storage of versioned object databases [5]. In such a representation, an object contains the entire history of an XML node. This is clearly a better approach for temporal queries. However, for each new version, we have to modify in the store all the objects that were modified since the last one. This update in place is typically very costly in terms of processing.

As mentioned before, we decided to store the last version and the sequence of forward completed deltas. We believe it is a good compromise. The most recent version is available. Forward deltas (by pruning of the completed deltas) and backward deltas (by inversion and pruning) are available. We do not have to perform updates to the store, only appends. From a storage viewpoint, it is certainly not the best since completed deltas are more space consuming than, e.g., simple backward deltas. We will see how compression allows us to reduce redundant storage to a reasonable level.

4.2 Management of XIDs

In this section, we consider a critical issue in our method, namely the management of Xyleme IDs, XIDs. XIDs are persistent identifiers given to all nodes of a document. Xyleme also uses node identification

for full-text indexing. However, the requirements for full-text indexing differ from that of versioning. Thus, XIDs are only used for change management and in particular by the versioning module.

An *XID-map* provides a mapping between the nodes of a tree and some integers that identify these nodes. The XID-map also specifies what is the next available integer to avoid reassigning the ID of a node that has been deleted to a new node. An example of XID-map and the tree it applies to are shown in Figure 3. The XID-map is based on the listing of XIDs of the nodes in postorder traversal of the tree. In the listing, we use ranges which often produces important space savings. In the example, the next available XID is 29. The XID-map specifies that we should traverse the tree in postorder and assign integers from *(1-3,7-13,5,14-28)* while doing so, i.e., 1, 2, 3, 7, 8, etc. This particular mapping could have been obtained, for instance, from a tree with XID-map *(1-28)|29* if nodes 4 and 6 were deleted and node 5 was moved. The XID-map assigns a unique (persistent) integer to each node. We argue that it does it in a compact (storage-wise) manner.

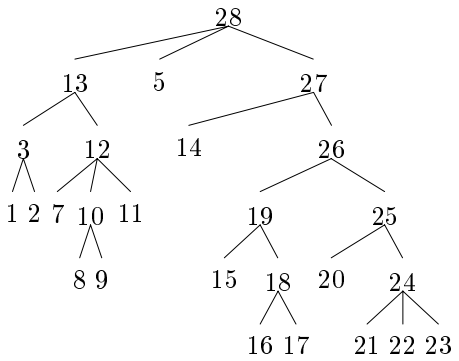


Figure 3: Tree with XID-map *(1-3,7-13,5,14-28)|29*

The method for creating and managing XID-maps is described next:

Initialization. At initialization, the *XID-map* is $1-n \mid n+1$ where n is the number of nodes in the tree. It states that the tree should be visited in postorder assigning integers from 1 to n and that $n+1$ is the next available integer.

Evolution. Note that XIDs are persistent names. In particular, an original node will always keep this initial identification even if moved. For insertions, we assign new integers to the nodes in the inserted subtrees using again a postorder traversal for the inserted subtrees.

The matching with the old version and the assignment of XIDs to the new nodes provide XIDs for all nodes of the new version. The XID-map for this new version is constructed by traversing the resulting tree in postorder and recording the identifiers of nodes that are traversed.

To see an example, consider the XID-map $1-534 \mid 535$. Suppose a subtree is deleted. At this stage, the XIDs in a subtree consist in consecutive integers, say $112-156$. The XID-map is now $1-111, 157-534 \mid 535$. Now, suppose we insert a new subtree of 22 nodes just before node 388 in post-order traversal. The resulting XID-map is now $1-111, 157-387, 535-556, 388-534 \mid 557$.

Note that the XID-map of a tree provides identifiers to every node of the tree. Observe also that the identification is stored separately from the tree data itself. One might consider storing the XIDs inside the XML document, e.g., add one extra attribute per element and store the XID in it. The main drawback of this method is space. This would add one attribute per node in the document and may increase the size of the document in a not negligible way (roughly 20% or more depending of the nature of the storage and the specific document). Besides, it involves changing (internally) the document, which leads to extra work when accessing the document or processing queries.

The XID-map allows to identify the nodes in each particular version in a unique manner. It provides identification for all nodes. It is stored only for the current version and is stored separately from the document. Portions of the XID-map for deleted subtrees are also stored in the delta. When a user requests the current version (and is interested in changes and not only snapshots), the user is sent the current version together with its XID-map. Future changes will always refer to the XIDs based on the assignment specified by the XID-map.

Remark 4.1 Observe that for a document of n nodes, the length of the list in the XID-map may grow in the worst case to $n+1$ integers, more precisely, to order of $n \log n$ bits. However, observe that, in general, the list is much smaller than n integers. Indeed, the size of this list may grow linearly in the number of changes and become as large as the number of nodes in the document. In practice, it does not because some operations may reduce the length of the XID-map and others such as insertions of large subtrees will tend to reduce the ratio between the size of the XID-map and the number of nodes in the document. Thus, one may expect, in general, the XID-map to be much smaller than $n \log n$. □

For XIDs, attributes play a special role. Observe that XML does not allow a node n to have two attributes with the same name, say a. So, "attribute a of node n" is a complete identification. This is why we do not assign XIDs to attributes but only to element and text nodes. Besides, attributes are not ordered in XML, so it is not obvious to extend the notion of XIDs that is essentially based on order to attributes that are by definition unordered in XML.

To conclude this section, we mention techniques for identifying nodes that we are considering but are not using yet.

Using DTDs. Some elements could be handled like attributes. Suppose for instance that the DTD states that each *product* has a single sub-element called *description*. Then we could identity such a *description* node using the XID of the product and its tag.

Semantic IDs. We assign Xyleme IDs to all nodes independently of the document content. In some cases, the data itself may contain meaningful identifiers. Observe that from a user viewpoint, such identifiers carry more meaning that system-generated IDs. In particular, in the XML world, IDs (together with IDREFs) are typically used to denote elements and are therefore primary candidate for serving as semantic IDs. It would be possible to use such data as XIDs. However, note that the persistence of such identifiers is not guaranteed by XML.

Using position. They are many ways to specify compactly positions in a tree. For instance, Xyleme uses for indexing purposes, a *prefix/postfix/level coding* that allows to identify a node with a triplet (i, j, k) where i is its ranking in pre-order traversal of the document, j that in postorder and k its level. The drawback of all these techniques is that such identifications are not persistent. When the structure of the tree changes, so does the identifier of the node. We are currently working on an identification mechanism that would combine persistence as the XID scheme with positional information needed by query processing such as being able to determine that a node is an ancestor of another one.

Selective XIDs. Lastly, one may consider identifying with XIDs only certain kinds of elements, typically those that are likely to change. E.g., in a catalog, new products are going to be added and prices to be changed, but the identification of products and to some extent, their characteristics is less likely to change.

4.3 Implementation aspects

We consider next some implementation aspects.

Each completed delta is stored as an XML document. When the system decides to install a new version of some data, the following steps occur:

1. The new version is obtained from the Web and the previous version is loaded from disk.
2. *diff* is run between the two versions and provides a matching between nodes of the two versions.
3. XIDs are attached to the nodes in the old version using the XID-map of the old version. The nodes in the new version that were matched to existing nodes acquire the corresponding XIDs. Inserted nodes get new XIDs. The XID map of the new version is constructed.
4. The completed delta is computed.
5. The new version and the new XID-map are stored. The completed delta is appended to the delta. The old version is deleted.

A main issue w.r.t. completed deltas is the storage of redundant information. For example, if an element has been detected as updated at versions i and j, the new value of its update in $\overline{\Delta}_{i-1,i}$ and the old value of its update in $\overline{\Delta}_{j-1,j}$ are the same. The repeated values may be large strings. Similarly, a subtree inserted then deleted appears twice in the completed deltas.

Our storage strategy therefore possibly introduces some redundancy. It is possible to reduce that cost by keeping track of the data that is already recorded and not redundantly store it. This makes query processing much harder. One could also use pointers to values already in the store but that would make the installation of a new version quite costly. We decided to use this very redundant storage policy with the possibility to apply periodically compression steps.

Compression During compression, we first read the history to detect all "large" objects that are stored redundantly. For data smaller that the size of the pointers, it is simply more efficient to simply duplicate the data. Note that for this detection, we can rely on the XIDs that provide persistent identification for the objects. Then we use pointers to physical locations in the store. We process the sequence of deltas starting from the most recent one replacing repeated large objects by pointers to more recent locations in the delta. These are rather standard issues that will not be detailed here. With this compression step, one can show that the storage is comparable to that obtained in a more standard versioned database.

To conclude this section, we briefly mention some complementary techniques. In the current implementation, we did implement the first one, aggregation. The last two, intermediate versions and archiving, are not yet supported.

Aggregation Typically, the granularity one would like for a document varies in time. E.g., one might want to have biweekly versions for the last month, weekly for the previous one, monthly for the previous year and yearly before. For that, it may be necessary to aggregate consecutive deltas. This will typically result in some space saving but at the cost of loss of information.

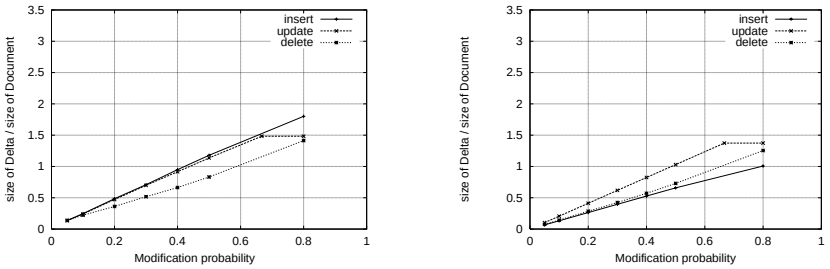
Intermediate Versions It may be useful to store intermediate versions from time to time. Intermediate versions complicates change queries and monitoring but speed up, for instance, the recomputation of old states [21].

Archiving In our approach, archiving is straightforward. It suffices to archive the sequence of deltas before a certain date.

Some measures on the size of deltas are given next.

4.4 On the size of deltas

To evaluate the physical representation, we measured deltas generated by a change simulator. The change simulator produces, given an input XML document, modified versions of the document. The simulator can be controlled by some parameters such as the probability of an insert, delete or update [9].



(a) Modifications on a small document (4K)

(b) Modifications on a large document (331K)

Figure 4: Ratio between the size of the XML delta and the size of the XML Document

Figure 4 gives the ratio between the delta documents and the original documents for documents of 2 different sizes. Each point was computed from 1000 test results. Not surprisingly, the size grows linearly, reasonably in the ratio of changes. For small documents, it reaches faster the size of the document. Clearly, when the document is small, any overhead costs a lot as a fraction of the document size.

Note that when the size of the delta is important compared to that of the document itself, it may seem more appropriate to simply keep the versions. However, the delta is more informative since it also keeps change information that would have to be recomputed if we store simply the versions.

To continue with this analysis, consider Figure 5 that illustrates the relative sizes of different informations for a sequence of 10 changes. Sizes are given as a percentages of the size of storing all the versions. The first graph gives the values for a modification ratio of 15% (5% update, 5% insert, 5% delete), the second one for a ratio of 30% (10%, 10%, 10%). For each document size, we consider (i) the total size of the versions; (ii) the size of the sequence of completed deltas; (iii) the size of the sequence of simple deltas; (iv) the size of the aggregated delta (that ignores the intermediate steps). For the completed delta, this is without the compression phase. These measures give some intuition of the storage overhead that is incurred if we want to support some functionalities, and, in particular, of the cost of keeping the completion.

We measure here the size of deltas as text files. The actual storage in our XML repository [13] would require a little less space than the text size. This factor would not have much impact on our measures. A limitation is that we measure changes made by the simulator. It would be interesting to experiment with real data gathered on the Web. We are currently conducting such experiments. Finally, it would be interesting to see the impact of the structure of the document (e.g., deep vs bushy trees, regular vs. irregular trees) on the size of the delta. We plan further studies in these directions.

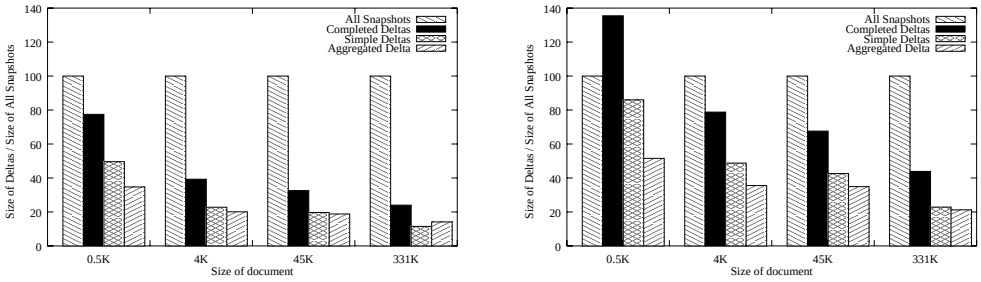
5 Conclusion

All the ideas described here have been implemented and tested. In particular, we implemented:

1. the management of deltas in a native XML repository (Natix [13]). When a new version arrives we compute the changes and modify the history by a simple append.
2. the computation of the composition of completed deltas. This serves many purposes. The main difficulty is the computation of positions for the moves and insertions.
3. the application of a delta to an instance, i.e., the computation of $\Delta(V)$ given V and Δ.
4. the projection of delta *forward* and *backward*. The forward one is used after composition to be able, for instance, to send a simple forward delta $\Delta_{t,now}$ to a user. The backward is used after composition to compute $\Delta_{now,t}$. It allows then to reconstruct old versions.
5. A GUI that can be used to display changes to the user, if requested. The GUI is described in a technical report [18].

The *diff* algorithm, a core part of the system, is described in [9] together with some performance measures. The change simulator that we used is also described there in more details. Many issues need to be further investigated:

1. We plan to pursue the study of the foundations of change composition (e.g., the group of completed deltas) in more depth and in particular the study of the rewrite system of update operations.
2. We believe that there is a lot of room for optimization in the storage of delta, e.g., using the compression technique described here and following other directions such as [15]. Also, a very interesting issue is to develop learning tools that, based on the sequence of versions of a document (or a site) and on the needs of users, adopt the best storing strategy for it.
3. An interesting issue is that of the processing of temporal queries based on the change-centric representation we proposed here.



(a) 15% Modification on the Document

(b) 30% Modification on the Document

Figure 5: Relative sizes of deltas on a sequence of 10 snapshots of a document

4. It would be interesting to develop new strategies for allocating identifiers to nodes in an XML tree. In particular, we are investigating a identification strategy that would provide persistent identifiers and would also give indications on the structure of the data.

Acknowledgments We would like to thank many people who participated in Xyleme meetings where we discussed the ideas that led to the present paper and, in particular, B. Amann, S. Cluet, G. Ferran, G. Jomier, J. Jouglet, C.-C. Kanne, D. Le Niniven, F. Llirbat, T. Milo, G. Moerkotte, B. Nguyen, M. Preda, L.Segoufin and V. Vianu.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, 2000.

[2] M.E. Adiba and B.G Lindsay. Database snapshots. In *VLDB*, 1980.

[3] V. Aguiléra, S. Cluet, P. Veltri, D. Vodislav, and Fanny Wattez. Querying XML Documents in Xyleme. In *Proceedings of the ACM-SIGIR Workshop on XML and Information Retrieval*, 2000.

[4] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, pages 53–64, 2000.

[5] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In *VLDB*, 1990.

[6] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *Theory and Practice of Object Systems*, 5(3):143–162, 1999.

[7] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. A comparative study of version management schemes for XML documents. Technical Report TR-51, TimeCenter, 2000.

[8] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient management of Multiversion Documents by Object Referencing. In *Proceedings of 13th International Conference on Very Large Data Bases*, 2001.

[9] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. Technical report, INRIA - Columbia University, 2001.

[10] Concurrent versions system. http://www.cvshome.org/.

[11] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *SIGMOD*, 1996.

[12] Oasis, ICE resources, http://www.oasis-open.org/cover/ice.html.

[13] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. Technical Report 8/99, University of Mannheim, 1999.

[14] Kinecta, http://www.kinecta.com/products.html.

[15] H. Liefke and D.Suciu. XMill : an Efficient Compressor for XML Data. In *SIGMOD*, 2000.

[16] L. Mignet, Mihai Preda, S. Abiteboul, S. Ailleret, B. Amann, and A. Marian. Acquiring XML pages for a WebHouse. In *proceedings of Base de Données Avancées conference*, 2000.

[17] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring xml data on the web. In *SIGMOD*, 2001.

[18] D. Le Niniven. Rapport de stage DESS : Interface homme-machine pour le suivi des evolutions de repository XML, 2000. Université Paris Sud.

[19] Michael Stonebraker. The design of the postgres storage system. In *Proceedings of 13th International Conference on Very Large Data Bases*, pages 289–300, 1987.

[20] Walter F. Tichy. Rcs- a system for version control. 15(7):637–654, 1985.

[21] Kristian Torp, Leo Mark, and Christian S. Jensen. Efficient differential timeslice computation. *TKDE*, 10(4):599–611, 1998.

[22] W3C. EXtensible Markup Language (xml) 1.0. http://www.w3.org/TR/REC-xml.

[23] N. Webber, C. O'Connell, B. Hunt, R. Levine, L.Popkin, and G. Larose. The Information and Content Exchange (ICE) Protocol. http://www.w3.org/TR/NOTE-ice.

[24] Xyleme Project. `www-rocq.inria.fr/verso`.

[25] Xyleme. `www.xyleme.com`.

[26] T. W. Yan and Garcia-Molina H. The SIFT Information Dissemination System. In *TODS 24(4): 529-565*, 1999.