Change-centric management of versions in an XML Warehouse*

Amélie Marian Serge Abiteboul Laurent Mignet

INRIA-Rocquencourt
78153 Le Chesnay
France
email: firstname.lastname@inria.fr

July 2000

Abstract

We consider the management of changes in a Web Warehouse of XML data. Our approach is change-centric in that it focuses on deltas, i.e., the changes themselves vs. other approaches based on snapshots or integrated representations.

We study a logical representation of changes based on deltas and some particular identifiers, XIDs. We consider implementation issues and analyze the advantages and disadvantages of several storage policies. Based on some requirements (certain functionalities that we want to support efficiently), we motivate the choice of our particular storage policy. It is based on storing the last version, "completed" forward deltas and on an original identification scheme for XML nodes.

We report briefly on the status of an implementation within the Xyleme project.

Keywords: XML, Datawarehouse, Versions, Deltas, Temporal Queries, Change Control.

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 their evolution. They want to see changes as information that can be consulted or queried. Indeed, change detection and notification is becoming an important part of web services, e.g., [23, 15, 28]. This is the problem considered here in the context of a data warehouse for XML data.

XML is becoming the new standard for semistructured data exchange over the Internet [26, 2]. We propose an approach for change management in a web warehouse of XML data. This work is part of the Xyleme project [29] which proposes to study and build a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing all the XML data available on the planet.

Thanks to the properties of this emerging language, Xyleme will be able to deliver very high level services which are difficult or impossible to support with the current web technologies. Notably:

- Version management and temporal queries in DOEM-QL [11] style, allowing to follow the evolution of the XML documents,

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

- Query subscriptions [7], a change monitoring service (such as [10, 1]), to notify end-users about interesting events in the repository,
- A smart and user friendly search engine in an XML-QL [13] or XQL [25] style, able to answer queries relying on the structure of the XML documents but also, to synthesize information coming from distinct documents [5],
- Semantic data integration to provide a single view of heterogeneous, though semantically related data.

XML data is retrieved from the web by Xyleme Acquisition module [22] which uses a web crawler to get new XML documents and to refresh old ones according to a refresh policy.

Because Xyleme is a warehouse of web XML data, it is impossible to have a real time vision of the data. The time of a document version is the time when Xyleme decides to refresh the document. We thereby get snapshots [4] of the documents. Similarly, for continuous queries, i.e., queries that are regularly evaluated, we get snapshots of the answer. We can compute the modifications that occurred between time t_{i-1} and time t_i using a *diff* algorithm. There has been a lot of works on such algorithms on relational, e.g. [20], or tree data, e.g., [27, 9]. Intuitively, they focus on computing a minimum *edit script* between the versions at time t_{i-1} and t_i.

The sequence of snapshots and the diffs between consecutive ones form the basis of the logical representation of temporal data we use. We opted for a change-centric approach as opposed to a data-centric one that is more natural for database versioning. Our logical representation is based on *deltas* in the style of [14]. It is reminiscent of traditional ways of representing logs in database systems. Deltas are often lossy and cannot be inverted. We introduce *completed deltas* which are deltas containing additional information so they can be reversed. In our logical representation, all XML nodes are assigned a persistent identifier, that we call XID for Xyleme ID. The use of persistent identifiers is essential to describe changes and also query changes efficiently.

Main issues that are addressed in the paper are the choice of a representation system and a storage policy. Our choices are based on the following requirements. We want to be able to (in order of importance):

1. obtain the current version, by far the most frequent query even for versioned data.
2. get the modifications between the value of the data at time t_i and its current value. The data can be the version of a document or the answer to a certain query. We are primarily interested in the changes to this data.
3. subscribe to change notifications and query these changes.
4. compute temporal queries.
5. rebuild the document as it was at some time t_i.

Our physical storage policy is based on storing the current version, an XID-map (to handle XIDs) and the "forward unit completed deltas". This choice is motivated by the functionalities we expect from the system. The XID-map is a novel concept that allows to attach persistent XIDs to every node in a storage-efficient manner. When a new version is installed, the new unit delta is appended to the existing delta for that page. This can be done without having to update data in place in the store. The price for this is that we store

redundant information. We show a delta compression technique that can be used periodically to recover space.

As previously mentioned, this work is done in the context of the Xyleme project. The project is still in an early phase and we only start obtaining large quantities of data. The present paper is a preliminary report on the on-going work about change management. The choice of the storage was guided by an analysis of the requirements of this change management (discussed here) and some back-of-the-envelope performance evaluation. We are currently implementing the versioning system and started performing extensive tests to (hopefully) validate our choices.

In Section 2, we present the logical representation based on XIDs and (completed) deltas. In Section 3, we discuss our storage policy. We give a synthesis of the possible uses of our deltas in Section 4. In Section 5 we give a brief description of our implementation and future works.

2 Logical Representation

In this section, we introduce a logical representation for changes in XML data. This representation relies on what we call *deltas* (essentially edit scripts) and *identifiers* for all XML element and text nodes.

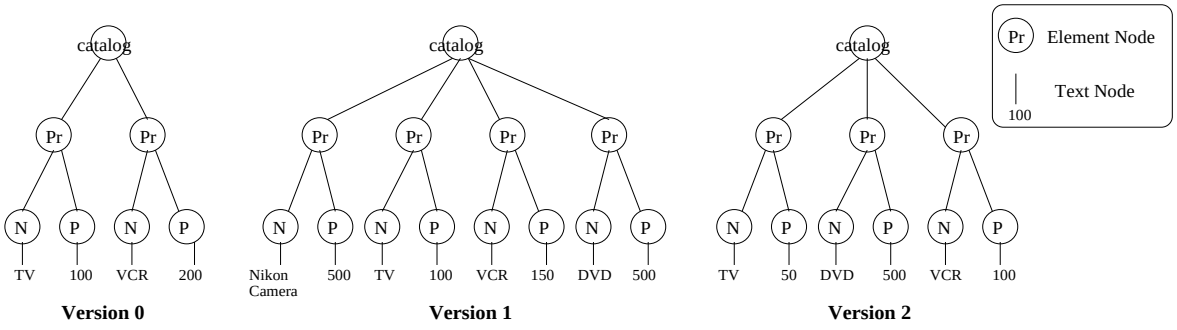


Figure 1: Snapshots of a catalog document

The starting point for our study of changes is a sequence $V_1, ..., V_n$ of snapshots of a single document D or of the answer to a query Q (See Figure 1). Thus, we assume here that we do not have direct access to updates, which is typically a difference with a more standard database context. We detect changes by polling regularly the document from the web or evaluating the continuous query. We use a *diff* algorithm that extracts the changes based on some heuristic, e.g., minimum edit script. For instance, consider the first two snapshots in Figure 1. The result of the *diff* algorithm that we use [8] is shown in Figure 2 where **A** and **B** denote XML trees. The doted lines represent matchings between XML nodes of the two versions discovered by *diff*.

Observe that we obtain versions of D at time $t_1, ..., t_n$. We do not have finer precision on what happened between time t_i and time t_{i+1}.

We next describe the formal model that is based on unit deltas computed from the *diff* algorithm matching and on identifiers. Based on that, we construct a general representation of changes in a natural manner.

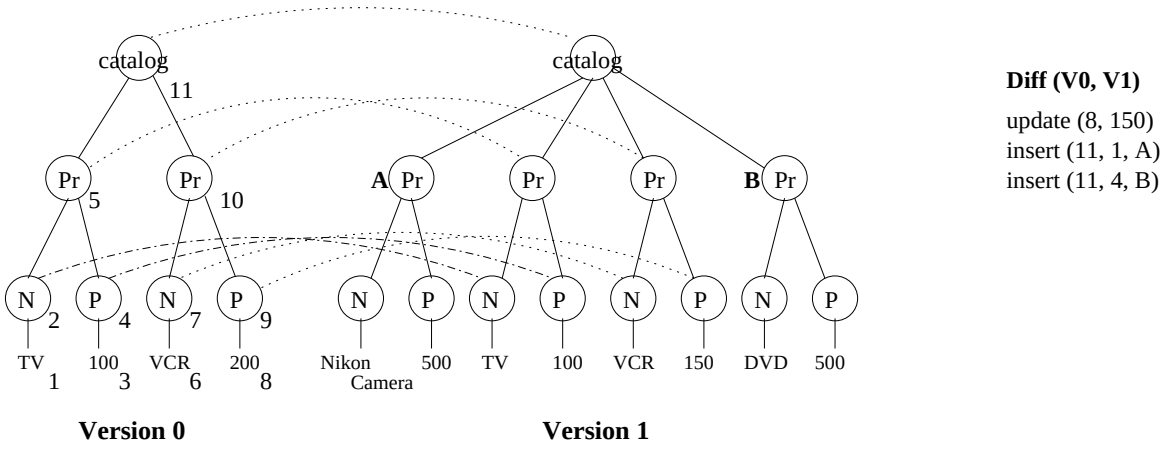


Figure 2: Result of *diff* between Versions 0 and 1 of the catalog

Xyleme IDs and versions We need to be able to track XML nodes through time. For this, we need identifiers that we call *Xyleme identifiers, XIDs*. There are many motivations for XIDs. For instance:

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

Observe that XIDs considered here are a *logical* concept that can be used to denote an XML node in a persistent manner. Any physical means of denoting the particular node can be used. We will discuss particular implementations of XIDs further.

We will assume that XIDs are from a set $\mathcal{N}$. (In our implementation $\mathcal{N}$ will be the set of integers.) Values are from a set $\mathcal{Q}$, e.g. the set of strings. In the present paper, we use without loss of generality the following simplified model for XML.

Definition: An *XML tree* is a triple $\langle t, \lambda, \nu \rangle$ where

1. t is a finite ordered tree with nodes from $\mathcal{N}$;
2. λ, the *name mapping*, assigns a name in $\mathcal{Q}$ to each node in t;
3. ν, the *data value mapping*, assigns a value in $\mathcal{Q}$ to each node in t. Note that ν is a partial mapping.

An example of an XML document and a graphical representation of its corresponding tree are given in Figure 3. For some names, we use in the figure a compact notation, e.g., *Pr* for *Product*.

The tree is ordered in the sense that it includes an ordering of the children of each node. Observe that we do not consider attributes here. We will see further on how attributes are handled. (The system we are implementing deals with the general XML model.)

```
<catalog>
    <Product>
        <Name>TV</Name>
        <Price>50</Price>
    </Product>
    <Product>
       <Name>Nikon Camera</Name>
       <Price>500</Price>
    </Product>
    <Product>
        <Name>VCR</Name>
        <Price>100</Price>
    </Product>
</catalog>
```

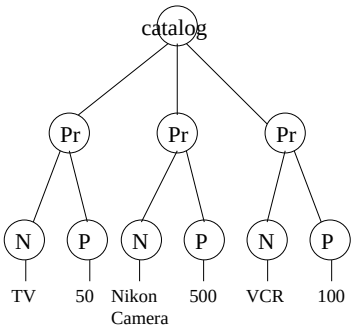


Figure 3: An XML tree representation

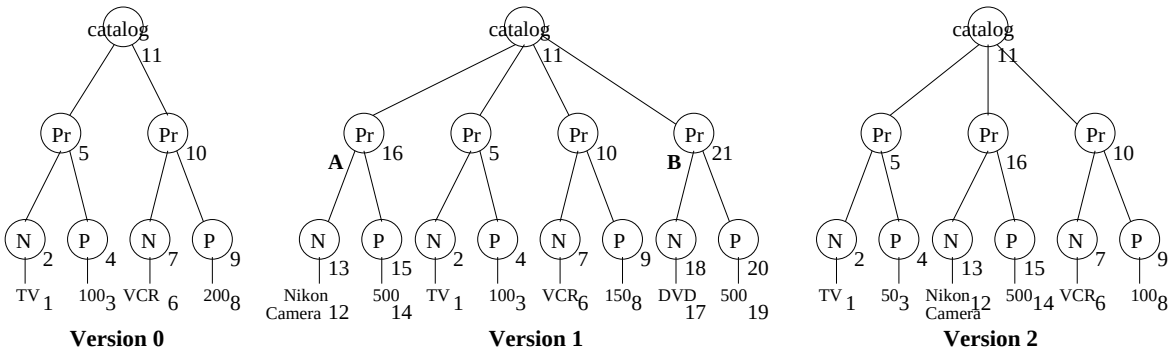
Deltas Let $\vec{V}$ be a sequence of snapshots. The *diff* algorithm provides a matching between nodes in consecutive versions. Consider in Figure 4 the same sequence of snapshots as in Figure 1 with the XIDs and the matching provided by the *diff* algorithm. In particular, *diff*(V_i, V_{i+1}) provides $\Delta_{i,i+1}(\vec{V})$, that we will note $\Delta_{i,i+1}$ when $\vec{V}$ is understood. We also assume that it provides the reverse one-step delta $\Delta_{i+1,i}(\vec{V})$, or $\Delta_{i+1,i}$ when $\vec{V}$ is understood. Both are edit scripts consisting in sets of operations over XML trees. The operations are:

1. *delete(n)* that deletes the XML tree rooted in node n
2. *update(n,v)* that changes the value of the text node n to v
3. *insert(n,k,T)* that inserts the XML tree T as a child of the element node n in position k
4. *move(n,k,m)* that moves the XML tree rooted in node m to be the child of n in position k.
5. *move_update(n,k,m,v)* that is a combination of move and update operations.

In the following, changes will be described by deltas, i.e., *sets* of such operations. Operations in a delta $\Delta_{i,j}$ represent the set of change operations needed to go from a state to another one V_j. Operations in a delta are not ordered in the sense that deltas do not provide any explicit order. On the other hand, there is an implicit order. Suppose that n has 3 children, a, b, c and that a delta contains *insert(n,1,*T_1*), insert(n,2,*T_2*)*. If T_1 is inserted first, the result is $T = (T_1, T_2, a, b, c)$, whereas if T_2 is inserted first, we get $T' = (T_1, a, T_2, b, c)$. To avoid such ambiguity, we decide that objects are inserted in the order of the insertion index, so the index actually provides the index of the object in the new version. Thus, for instance, in the example, the result is T, so T_1 will end up being in position 1 and T_2 in position 2.

Observe that such deltas potentially lead to consistency issues. For instance, we can apply a *delete(n)* operation only if there exists a node n. Also, for instance, we cannot move two nodes as children of the same one in the same position. In the following, we will assume that all the deltas we deal with are consistent. Deltas may be viewed as operators. As such, we assume that the *diff* algorithm produces deltas that are consistent, i.e., for each i,

$$V_i \circ \Delta_{i,i+1} = V_{i+1} \quad V_{i+1} \circ \Delta_{i+1,i} = V_i$$



Diff (V0, V1)

update (8, 150)
insert (11, 1, A)
insert (11,4, B)

Diff (V1,V2)

delete (21)
move (11, 2, 16)
update (3, 50)
update (8, 100)

Figure 4: Snapshots of the catalog document using XIDs

This model of changes is rather simplistic. In our implementation, we also consider 3 operations on attributes: attribute insertion, modification and deletion (but no move). One could consider a yet richer set of operations. For instance, one could consider a name changing operation that would allow to change the name of a node, e.g., from *subsection* to *section*. One could consider also more sophisticate update operations, e.g., the means to insert a string in position k of an existing string, or an increment for integers. Although these would be interesting to introduce from a practical viewpoint, they would not change the framework in any substantial manner, so they will be ignored here.

We assume here that an insertion of a subtree is a single operation. Clearly, from a cost viewpoint (in particular for determining a minimum edit script), such an operation may be quite expensive if the tree is large.

Operations on deltas Let us define $\Delta_{i,i} = \phi$, the empty set of operations. The composition of deltas can be easily defined. In some cases such as updates, it is straightforward. E.g.,

$$\{update(8, 150)\} \circ \{update(8, 100)\} \equiv \{update(8, 100)\}$$

For other modifications, it is more complex because of positions. E.g.,

$$\{insert(24, 1, X)\} \circ \{insert(24, 1, Y)\} \equiv \{insert(24, 2, X), insert(24, 1, Y)\}$$

Based on that composition, we can define:

$$\begin{aligned} \Delta_{i,j} &= \Delta_{i,i+1} \circ \ldots \circ \Delta_{j-1,j} \quad \text{for } i < j \\ \Delta_{i,j} &= \Delta_{i,i-1} \circ \ldots \circ \Delta_{j+1,j} \quad \text{for } i > j \end{aligned}$$

It can be shown that:

$$\begin{aligned} V_j &= V_i \circ \Delta_{i,i+1} \circ \ldots \circ \Delta_{j-1,j} = V_i \circ \Delta_{i,j} \quad \text{for } i < j \\ V_j &= V_i \circ \Delta_{i,i-1} \circ \ldots \circ \Delta_{j+1,j} = V_i \circ \Delta_{i,j} \quad \text{for } i > j \end{aligned}$$

All different operations can be aggregated together. For instance if a node inserted in $\Delta_{i-1,i}$ is updated in $\Delta_{i,i+1}$, the corresponding operations in the aggregated delta $\Delta_{i-1,i+1}$ is an

insertion of the node but with its new value, a node updated in $\Delta_{i-1,i}$ and moved in $\Delta_{i-1,i+1}$ will be moved and updated (using the move_update operation) in $\Delta_{i-1,i+1}$. Aggregating deltas is not trivial, we already said that aggregating can be complex because of positions, but the different combinations of operations must also be reviewed carefully to avoid errors while aggregating.

Observe that such deltas allow to aggregate modifications from several deltas on the same document to get a new delta which has a higher granularity. If $\Delta_{i,j}$ gives the modifications from V_i to V_j, and $\Delta_{j,k}$ the modifications from V_j to V_k, $\Delta_{i,k} = \Delta_{i,j} \circ \Delta_{j,k}$ will give the modifications from V_i to V_k. However, observe that, in general, the aggregation of the unit deltas from time i to time j ($\Delta_{i,j}$) is not equal to the result of the *diff* algorithm over V_i and V_j (*diff*(V_i, V_j)). Observe also that it is only possible to aggregate deltas that do not have time gaps between them, i.e., $\Delta_{i,j} \circ \Delta_{k,l}$ is only defined if $j = k$

A main practical issue is that it is not possible to obtain $\Delta_{j,i}$ knowing $\Delta_{i,j}$, and in particular for $j = i + 1$ since some information is missing that can only be found in V_i. For instance, suppose $\Delta_{i,i+1}$ contains an update operation *update(5,"100")*. The delta ignores the old value of node 5 that is needed in the reverse delta. The loss of information arises for updates (old vs. new value) as well as for insert/delete (inserted/deleted trees). To see why this is important in practice, consider the following four possible representations of a sequence of snapshots:

$$(a)\ V_1, \Delta_{1,2}, \ldots \Delta_{now-1,now}; \quad (b)\ \Delta_{2,1}, \ldots \Delta_{now,now-1}, V_{now};$$
$$(c)\ \Delta_{1,2}, \ldots \Delta_{now-1,now}, V_{now}; \quad (d)\ V_1, \Delta_{2,1}, \ldots \Delta_{now,now-1};$$

Only (a) and (b) are lossless and allow to reconstruct the sequence. This is unfortunate since, in practice, one would appreciate to have available V_{now} and the forward deltas, i.e., representation (c).

This motivates the introduction of completed deltas.

Completed Deltas Completed deltas will be at the core of our representation of versions. In completed deltas, we keep the old and the new values in case of updates. Similarly, we keep the deleted tree in case of a deletion. In some sense, $\overline{\Delta}_{i,j}$ contains both the forward and the backward delta's. This is not a new notion. Logs in database systems also describe changes and often keep the old and new values. Here completed deltas are meant as logical data, that can be, for instance, exported or queried. We will consider issues related to their storage in the next section.

The operations in completed deltas are as follows:

1. $\overline{delete}$*(n,k,T)* that deletes the XML tree T from a child of node n in position k;
2. $\overline{update}$*(n,v,ov)* where *ov* is the old value;
3. $\overline{insert}$*(n,k,T)* that inserts the XML tree T as a child of node n in position k;
4. $\overline{move}$*(n,k,m,p,q)* that moves the XML tree rooted in node m from node p in position q to be the child of n in position k.
5. $\overline{move\_update}$*(n,k,m,p,q,v,ov)* which is a concatenation of a move and an update operation.

$\Delta_{1,2}$	$\Delta_{2,1}$	$\overline{\Delta}_{1,2}$
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 unit deltas

Table 1 gives the forward and backward *simple unit deltas* (Δ) from Version 1 to Version 2 from Figure 1 as well as the corresponding *completed unit delta* ($\overline{\Delta}$).

Completed deltas can be composed according to similar rules as regular deltas. Furthermore, it is also possible to invert completed delta's. So, in particular, we have, for each i, j, k, l:

$$(\overline{\Delta}_{i,j})^{-1} = \overline{\Delta}_{j,i} \qquad ((\overline{\Delta}_{i,j})^{-1})^{-1} = \overline{\Delta}_{i,j}$$
$$\overline{\Delta}_{i,j} \circ \overline{\Delta}_{j,k} = \overline{\Delta}_{i,k} \qquad \overline{\Delta}_{i,j} \circ \overline{\Delta}_{j,i} = \phi$$
$$(\overline{\Delta}_{i,j} \circ \overline{\Delta}_{j,k})^{-1} = \overline{\Delta}_{i,k}^{-1} = \quad \overline{\Delta}_{k,i} = \overline{\Delta}_{k,j} \circ \overline{\Delta}_{j,i}$$

In practice, the following properties are of particular importance: for each i, j,

$$\overline{\Delta}_{i,j} = \overline{\Delta}_{i,i+1} \circ ... \circ \overline{\Delta}_{j-1,j} \quad i < j$$
$$\overline{\Delta}_{i,j} = \overline{\Delta}_{i-1,i}^{-1} \circ ... \circ \overline{\Delta}_{j,j+1}^{-1} \quad i > j$$
$$V_i = V_n \circ \overline{\Delta}_{n,i}$$

Thus from V_n and the list of unit deltas ($\overline{\Delta}_{i,i+1}$) we can reconstruct all the versions and all possible completed deltas.

Remark 2.1 We discussed already two data representations. The first based on a sequence of snapshots and the second based on (completed) deltas. To process temporal queries, it may be useful to have one general representation of the history of the document. One such representation, called DOEM, is proposed in [9, 11] using annotations to describe changes. To illustrate, the history $H_{0,2}$ for the running example is given in Figure 5 using a representation ala DOEM. The main differences with DOEM are that (i) annotations are attached to nodes in our model and (ii) we take the ordering into account. Point (i) comes from the fact that the data we considered are trees, vs. graphs in the DOEM model. □

Remark 2.2 Storing delta is expensive. One can imagine having various level of services for different data managed by Xyleme. The most intense one provides the full functionalities based on completed deltas. Some data may be stored with simple deltas only, to allow users to refresh their data without supporting version management. Finally, for a large quantity of data, no temporal support may be provided. □

3 Using deltas, in brief

In the previous section, we considered the change model. Before getting to its physical representation (in the next section), we briefly consider in this section how deltas may be used.

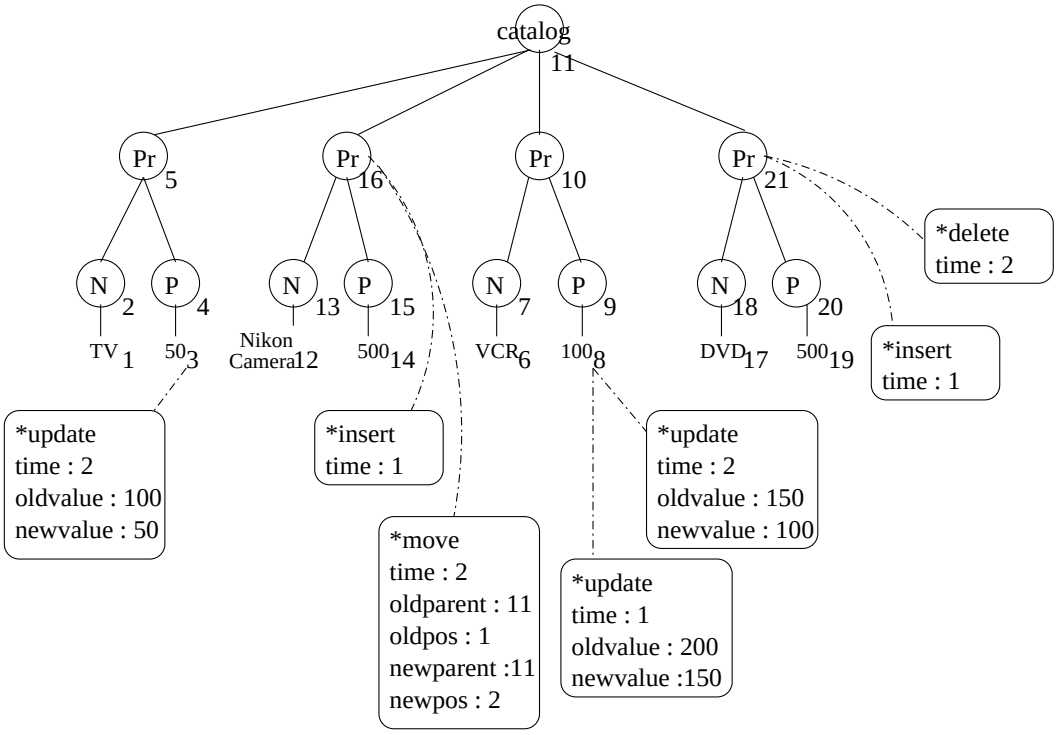


Figure 5: DOEM-style history of the document from Figure 1

Delta resemble the traditional logs of database systems and, even in the context of databases, one finds many applications that require access to the logs. A main difference here is that we view the delta as a logical notion, and more precisely, as data that may be externalized. In this section, we briefly discuss various possible uses of deltas in our context.

Mobile database and refreshing: A Xyleme client may request some documents, collections of documents, or query result, say V_i at time i and store this on its local computer. The client may then request $\Delta_{i,now}$ to refresh its data or see what changed. We use an application at the client site to perform the (small) task of combining $\Delta_{i,now}$ with V_i and displaying changes to the user, if requested. The same technology may be used to mirror a site or more generally, some portion of the web.

In this spirit, in the context of electronic commerce, the Information and Content Exchange, ICE [28, 17] is a standard that supports exchanging information about changes of a *package*, typically a set of pages. It is also based on snapshots. The deltas we use are more sophisticate since ICE deals with changes at the document levels whereas we address changes also at the level of portions of documents. Commercial offers of ICE are available, e.g., [19].

Monitoring changes: This is the first facet of a query subscription system we are implementing. At the time we compute the *delta*, we verify for each atomic change (by simple lookups in an index), whether this change is monitored by some subscription. Examples of changes that can be monitored include, for instance, any modification of a document or the insertion of a new item in a catalog.

ICE also provides a protocol for notifications. New services such as Mind-it [23] allow

to monitor changes at the page level in the web. We plan to support monitoring at the node level as well.

Querying changes: Deltas are XML documents and thus may be queried like any other documents. For instance, one will be able to ask for the list of all items ever introduced in a catalog, or for the new movies played in Paris this week. For this, we can use the standard indexes of the system. These queries will have the same syntax as regular queries over documents. Queries to changes form a second facet of the query subscription system. Continuous queries [12] are asked regularly to the warehouse. We believe that continuous queries over change often are what the users are interested in, e.g., notify me once a week of the top 20 new documents about XML.

Versioning: we will be able to obtain an old version of a document or part of one.

Temporal Queries: This is another way to query changes using a logical representation of the history of data and a Temporal Query syntax. More complex queries can be performed on the history of a document or a set of documents. This is in the spirit of temporal query languages and more precisely of DOEM-QL. Example of such queries are: what are the products which prices increased since time i? When did Nikon Camera F10 disappeared from the catalog? Note that, in our context, the history of the document(s) exists only virtually. A query over this history has to be compiled into a query over the deltas and the current version.

We briefly examine next the issue of queries over change vs. queries over the history and conclude by a brief discussion on change monitoring.

Recall that alternatives for querying the history is to reconstruct it (at least partially) or compile the temporal queries into queries over deltas. Suppose we are interested in the changes of prices of a product, say *VCR*, since 1998. This can be expressed as the following query over the history (using a Lorel [3] and DOEM-QL style):

```
select <change time=U.time value=U.newvalue/>
from   C is "www.zcat.com", C.Product P,
       P.Price.*update U
where  P.Name = "VCR" and U.time >= "1998/01/01"
```

An example of result yield by this query is:

```
<change time=2000/01/01 value=150/>
```

In the deltas, we need to use some "joins" based on XIDs to obtain the same information as in:

```
select <change time=D.time.to value=U.newvalue/>
from   C is "www.zcat.com", C.product P,
       C.delta.unit_delta D, D.update U,
where  P.Name = "VCR" and P.XID = U.XID
       and D.time.to >= "1998/01/01"
```

To conclude, let us considering monitoring. Suppose we want to monitor the price of the *VCR*. We may support such a mechanism via continuous queries. We might prefer to detect at the time we compute *diff* if a change of the price of such a product occurred, e.g., to send a notification. One may have to monitor in Xyleme huge amounts of such changes.

The detections have to be performed at very low cost while loading the new version, e.g., immediately after computing the *diff*. We are working on data structures that would support such detections [7]. Observe that such detection is particularly useful when the document is *not* versioned. Because in that case, we will not be able to evaluate the changes with a continuous query since the old state is not available. Observe also that we would in any case perform some kind of *diff* to update the warehouse indexes, so computing *diff* for a non-versioned data is not an overhead.

4 Physical Representation

In this section, we discuss the physical representation of versioned data in Xyleme. We first present some general requirements. Then we compare the particular storage strategy we chose to alternatives, based on these requirements. Finally, we discuss in more details some implementation aspects of the storage.

We decided to store the last version (the current one) in the repository as well as the *forward completed unit deltas*, i.e., ($\overline{\Delta}_{i,i+1}$)section, we know that this suffices to reconstruct the sequence of snapshots. (Since we use completed deltas ($\overline{\Delta}$), no information is lost.) This choice was motivated by our general requirements. (Recall, for instance, that it considered more critical to be able to provide the current version or the changes between the data at time t_i and its current value, rather than rebuilding the document as it was at some time t_i.)

We use the NATIX repository [18] developed at the U. of Mannheim that is adapted to the storage of tree data. We represent changes as XML trees which facilitates their transmission to clients. Changes described in XML may seem quite verbose storage-wise. However, a lot of the redundancy is introduced by the tags, such as *insert, delete* etc. NATIX represents such tags as integers, so they are not repeated in the store. In NATIX (and this is not specific to it), appending of new data is fast whereas updates to data in place is more expensive. This is an important aspect in the choice of a storage policy.

4.1 Advantages over other policies

We discuss our particular storage strategy (storing the last version and unit forward completed deltas) last. Alternatives we considered are as follows:

Storing first and current versions + forward deltas It is easy to see that no information is missing in that approach. The main advantage is for the computation of $\Delta_{i,now}$. It is slightly better in this respect than with completed delta's. This is because simple deltas use less space. So, typically, a little 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 V_i may become very costly for recent versions since we have to start from V_0.

Storing current version + backwards deltas In this method, we store the current version and simple deltas but backwards. These are deltas that give the modifications from a version to the previous one. Clearly, this solution preserves all the information. In particular, one can compute V_i for each i starting from V_{now} and applying the appropriate sequence of delta's.

This solution 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 that completed ones. Getting V_{now} is immediate and V_i for some i is rather efficient.

The main drawback is for the computation of $\Delta_{i,now}$. Since simple $\Delta's$ cannot be reversed (see above), the information missing from the delta has 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. See Figure 5. It is relatively easy to code such a tree in XML, so for Xyleme, to store and process it. This is more in the spirit of typical storage of versioned object databases [6]. In such a representation, an object contains the entire history of an XML node.

This is clearly the best 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. We preferred an approach that avoids updating objects in the store.

Our policy We believe it is a good compromise. The most recent version is available. Forward deltas (by pruning of the completed delta) 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.

To conclude this section, it should be noted that the choice of one policy depends on the pattern of use of the warehouse. In our choice, we assumed implicitly the pattern discussed at the beginning of this section.

4.2 Xyleme Version Storage Policy

The $\overline{\Delta}$'s are stored as an XML document. See Figure 6 for an XML representation of the unit completed delta from Table 1. The XML delta keeps all the information needed for reversing deltas: old values, parent and positions are kept as well as the XIDs of the inserted and deleted subtree (in the form of an XID-map of the subtree).

When a new version arrives from the web, (i) the previous version is loaded from disk, (ii) *diff* is run on the two versions, (iii) the completed delta is computed, (iv) the new version and the completed delta are stored, and (v) the old version is deleted.

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

Let us analyze the problem more precisely. If a node has not changed since V_0, we have imperatively to keep its old state in the delta when it changes. Otherwise, we loose information. On the other hand, once it has changed, keeping the old state is redundant. By state, we mean its value in case of an update or the subtree in case of inserts. Note that the situation is in fact more complex since the "state of a node" may be partially known, e.g., some of its children have not changed since V_0 - so are not known in the delta - and some have been inserted or updated - so are in the delta.

Several storage policies may be considered:

```
<delta>
   <unit_delta>
   ...
   </unit_delta>
   <unit_delta>
      <time from=''1'' to=''2''/>
      <delete parent=''11'' position=''4'' xid-map=''(17-21)/>
         <Product>
            <Name>DVD</Name>
            <Price>500</Price>
         </Product>
      </delete>
      <move xid=''16'' new_parent=''11'' new_position=''2''
                       old_parent=''11'' old_position=''1''/>
      <update xid=''3'' new_value=''50'' oldvalue=''100''/>
      <update xid=''8'' new_value=''100'' oldvalue=''150''/>
   </unit_delta>
</delta>
```

Figure 6: XML $\overline{\Delta}$ from Figure 1

- Redundant storage: we store the old and new values for updates and the deleted subtrees for deletes.
- Reduced storage: we keep track of the objects that are already described in the delta (with a sequence of m bits where m is the number of nodes in the document) and do not redundantly store them.
- Backward pointers: if a value/tree in $\overline{\Delta}_{now-1,now}$ is already present in a previous delta we replace it by a backward pointer to the corresponding object in the store.
- Forward pointers: if a value/tree in $\overline{\Delta}_{now-1,now}$ is already present in an older delta, we replace its value in $\{\overline{\Delta}_{0,1}, ..., \overline{\Delta}_{now-2,now-1}\}$ by a forward pointer to the corresponding object in $\overline{\Delta}_{now-1,now}$.

Each of these storage policy has advantages and drawbacks. Redundant Storage is wasting space since redundant information is stored. On the other hand, its processing is fast. Also, query processing is rather easy with such a storage policy. Reduced Storage is efficient in terms of storage and delta processing. But it is costly in terms of query processing. Using pointers saves storage space but computing $\overline{\Delta}_{now-1,now}$ when a new version is checked in may be considerably costly in terms of processing time since we have to load older deltas to compute the pointers. The Forward Pointers is better than Backward Pointer in the sense that it privileges recent data vs. staler one. On the other hand, Forward Pointers requires expensive updates to the delta already in the store, a drawback avoided by Backward Pointers.

Compression I We decided to use the Redundant Storage technique with, periodically, a Compression Phase to remove the redundancy. We briefly describe the semantics of the Compression Phase. For pointers, we use physical IDs of the store. 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, we prefer 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 can use two different strategies:

- Backward Pointers: we write the sequence of deltas starting from the most stale one (i.e. the first one: $\overline{\Delta}_{0,1}$) While doing that, we keep the physical IDs of the objects that we know are repeated and use these instead of the object themselves in more recent deltas.
- Forward Pointers: we write the sequence of deltas starting from the most recent one, $\overline{\Delta}_{now-1,now}$. While doing that, we keep the physical IDs of the most recent objects when these are repeated and use these instead of the object themselves in older deltas.

These two variations of the Compression I strategy allow us to diminish the main drawback of the Redundant Storage technique: storage space. The result in the store will be like the results of a Backwards (or a Forward) Pointers storage policy. But the processing of a new version, i.e., computing $\overline{\Delta}_{now-1,now}$ will be much faster. Clearly, optimization techniques notably based on indexes may be used to avoid having to read the entire history. These are rather standard but quite interesting 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:

Intermediate Versions It may sometimes be useful to store intermediate versions from time to time. An intermediate version is a complete version of the document at time i. These can be used to save space (in some case the delta may be bigger than the new version), or to save processing (for some very old i, it may be much easier to recompute V_i for an intermediary version that from V_{now}). Intermediate versions complicates change queries and monitoring. In the current implementation, intermediate versions are not supported.

Compression II 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 some loss of information. Suppose we aggregate over a year. Then an element inserted and deleted that same year disappears.

A subtlety is that a user may have stored a version, say of June 15th 1989 although this version does not exist anymore from the warehouse viewpoint. A careful (but not complicated) management of XIDs will allow us to handle this situation and indeed be able to send a delta, e.g., $\Delta_{01/01/89,now}$, that will permit refreshing the old version of the user (and detecting the changes in the newer version). Due to space limitations, we will not explain here how this is achieved.

Archiving In our approach, archiving is straightforward. It suffices to archive the sequence of deltas before a certain date. Some additional processing is needed if we use the backward pointers method for compression.

4.3 Management of XIDs

In this section, we consider a critical issue in our method, namely the management of XIDs. XIDs are persistent identifiers given to element and text nodes of a document. They are only used by the Versioning Module. Some other Xyleme modules also need identification of elements (e.g. for indexing and query processing) but their requirements for identifiers differs from Xyleme Version requirements.

Allocating Xyleme IDs Since each XML node of a versioned document has a Xyleme ID, it is necessary to have a convenient and efficient way to manage these XIDs. Logically, an identifier is assigned to each element. One needs also to use an *XIDnext* number which keeps the value of the next XID that can be allocated for the document. This *XIDnext* number is needed to avoid reassigning the ID of a node that has been deleted. An issue is the storage of the assignment of XIDs to elements. One might want to store the XIDs inside the XML document, i.e., add one extra XID attribute per element. The main drawback of this method is space. This adds 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 processing when sending it to the users.

It is more space efficient to keep the assignment of XIDs to XML nodes separately as a list of integers. More precisely, we store the list of XIDs of the nodes in some particular traversal of the tree. Indeed, we next see how a simple trick often produces large space savings.

Our management of XIDs is based on the utilization of *XID-maps* that provide a mapping between an element using its postorder position in the tree and its XID. See an example of a tree using an XID-map in Figure 7.

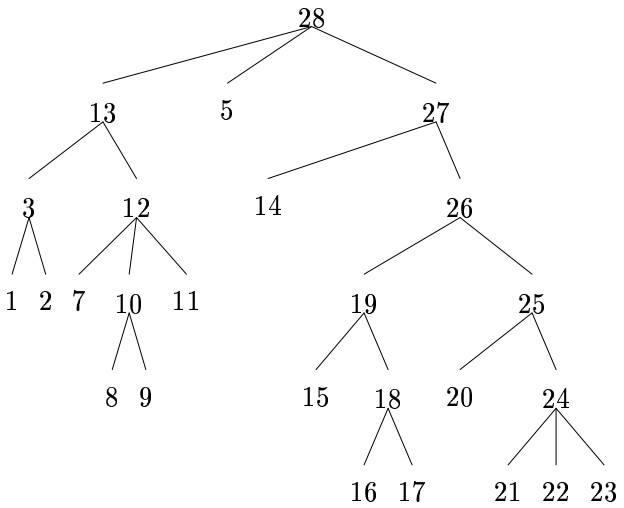


Figure 7: Tree with XID-map *(1-3,7-13,5,14-28)|29*

This example gives the XID-map for a tree at time t. It provides the next available XID to be used if a new node is introduced (here 29). It specifies that we should traverse the XML 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. Such a mapping could have been obtained, for instance, if nodes 4 and 6 were deleted between time 0 and time t and some nodes were moved. It assigns a unique (persistent)

integer to each node. We will argue that it does it in a compact (storage-wise) manner.

We present next our method for creating and managing XID-maps. We consider the initial allocation of XIDs, then their maintenance.

Initialization At the initialization, the identification of the nodes - that we call the *XID-map* - can be described by $\Xi_1 = 1..n \mid n+1$, which states that the tree should be visited in postorder assigning integers from 1 to n and that $n+1$ is the next available integer.

Note that XIDs are persistent names. In particular, an original node that is never deleted will always keep this initial identification.

Evolution When there are insertions, deletions and moves, the structure of the tree changes. For insertions, we use new integers and a similar assignment for inserted subtrees (using again postorder traversal for the inserted subtrees).

Consider Ξ_1. Suppose a subtree is deleted. It is easy to see that at this stage, node XIDs in a subtree consist in consecutive integers, say $i..j$. The XID-map is now $\Xi_2 = 1..i-1, j+1..n \mid n+1$. Suppose that we now insert a new subtree with l nodes just before node $k+1$ in post-order traversal. The resulting XID-map may now look like

$$\Xi_3 = 1..i-1, j+1..k, n+1..n+l, k+1..n \mid n+l+1$$

Such an XID-map does not provide the structure of the XML-tree. It provides, together with the tree, identifiers for all the nodes.

Observe the use of the XIDnext to keep the next XID that can be allocated and avoid allocating twice the same XID for two distinct elements.

What is the XID-map good for? It allows to identify the nodes in the current version and in the older ones in a unique manner. It is stored separately from the current version. When a client requests the current version (and is interested in changes and not only snapshots), the client is sent the current version together with the XID-map. Future changes will always use XIDs based on the assignment specified by the XID-map.

Observe that for a document of n nodes, the length of the list may grow in the worst case to[1] $n+1$. However, observe that, in general, the list is much smaller than n. To start, the list is small, size 3. Updates do not change the structure and so, do not modify the XID-map. Consider the other operations:

- a delete may cut a sublist in two.
- an insert may cut a sublist in two and add a sublist between the two parts.
- a move can be considered as composing a delete then an insert. It may thus lead to two cuts and an addition.

Thus the number of cuts grows linearly in the number of changes. However, insertions will have a tendency to reintroduce long sublists (for large inserts). Deletion may result in reducing the total number of sublists.

[1]Each element in the list is an integer. Since we know the highest integer λ (*XIDnext*), we can use, inside a unit delta, a coding of the integers that uses a fixed number of bits, e.g. order of $log(\lambda)$ bits, to gain some storage space.

Attributes We handle attributes in a particular manner. 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 nodes. Besides attributes are not ordered in XML, so it is not obvious to extend the notion of XIDs that are essentially based on order to attributes that are by definition unordered in XML.

Some elements could be handled like attributes. Suppose the DTD states that a *product* consists of a sequence of four elements, a *name*, a *description*, a *price*, and a *picture*. We could also limit the size of the XID-map by not assigning XIDs to such elements but treating them as components of identified elements. In the prototype, to simplify, we assign XIDs to all elements. Observe that an element such as the name of a product is unlikely to be inserted or deleted – only updated, or an entire product will be inserted/deleted. Thus products will always be represented as sublists in the XID-map and will therefore not much increase the size of the map. For these reasons, we believe that except for pathological cases, the length of the XID-map will be much smaller than the number of nodes in the DOM representation of the document, i.e., much smaller that the document.

Remark 4.1 (Resetting the XID-map) If in a particular application, the size of the XID-map is considered to be too large, we could "reset" it. This means that we will use two different XID-maps, one before a certain date and one after. We, of course, have to store the value of the XID-map at the time of the resetting. Since this complicates dramatically query processing (a node may now have a time dependent XID), we decided not to use such an optimization. □

4.4 Other techniques for managing XIDs

In this section, we discuss various techniques that we decided not to use.

Semantic IDs: Using document data as XIDs We assign Xyleme IDs to all nodes independently of the document content. In some cases, the data itself may contain meaningful identifiers. When this can be detected, the use of such (meaningful) identifiers may allow to save storage.

The best way to find that a particular attribute (or value) is an identifier for some element is via *publication*, i.e., some data source explicitly specifies such a knowledge. Such kind of publication will be provided in Xyleme but is not the topic of the present paper. We may also try to "learn" such information. I.e., by examining the delta of a document, we may try to find data that may be used as (meaningful) XIDs. In particular, in the XML world, IDs (together with IDREFs) are typically used to denote elements and are therefore primary candidate for serving as XIDs.

Note that the system will have to be flexible since nothing prevents one or more data values that were selected to form the persistent key for an element to change.

Identifying elements by their current position This is a completely different approach. The "position" of an element is used to identify it. They are many ways to specify compactly positions in a tree. For instance, Xyleme uses for indexing purposes, a *postfix/prefix coding* that allows to identify a node with a pair (i, j) where i is its ranking in preorder traversal of the document and j that in postorder. 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. It is an interesting research issue to obtain 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. To do that, one may use the fact that often only some nodes 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 and description are not. So, we may decide to consider changes only to certain nodes. So, to continue with the catalog example, a change in the characteristics of a product will be considered as the deletion of the product and the insertion of a new one. We may accept such loss of flexibility to save in XID processing. The choice of "updateable" elements may again be provided via publication or computed in a learning phase.

5 Conclusion

We briefly discuss the status of the work, some experimentation and future work.

The management of versions has been implemented with the exception of the compaction techniques. For *diff*, we have implemented a simple *XMLdiff* [8]. We are working on a more elaborate version of it. The management of version is being integrated to Xyleme. (Communications are via Corba.) The change GUI (briefly described further) has been implemented.

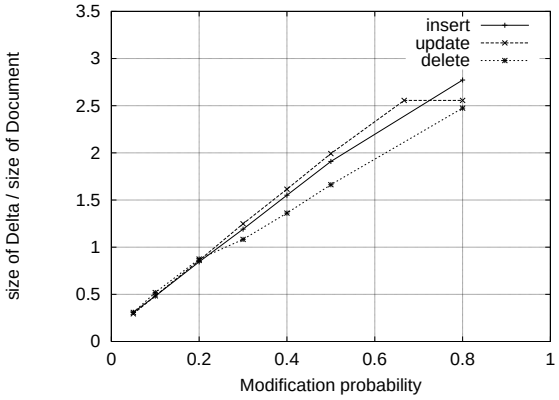
The change module of Xyleme is, like the entire system, programmed in C++. Documents and deltas in XML are stored in a repository tailored to the management of trees [18] developed at the U. Mannheim. Each time a new version of a versioned document or a continuous query is obtained, the Version Manager (VM) receives it from a web interface and loads the previous version from the stores. We use fingerprint techniques to avoid doing this when the data has not changed. The *treediff* is run on those two versions and the resulting *edit script* is converted to an XML "forward completed unit delta" with appropriate management of the XID-map. The new version, the XID map and the unit delta are then stored in the repository. The old version is deleted. As previously mentioned, only appends and deletes to the store are performed and the existing delta is not read, which provides a very fast management of new version.

Since deltas are stored as standard XML documents, we can use Xyleme standard query processor (under construction) on them. To implement sophisticate temporal queries, we plan to use a compilation into standard queries over the delta followed by a "de-compilation" of the result. The use of the XIDs together with indexes over the delta (full-text as well as time-based) will allow to optimize such queries. Design of this part of the system is still at a very early stage.

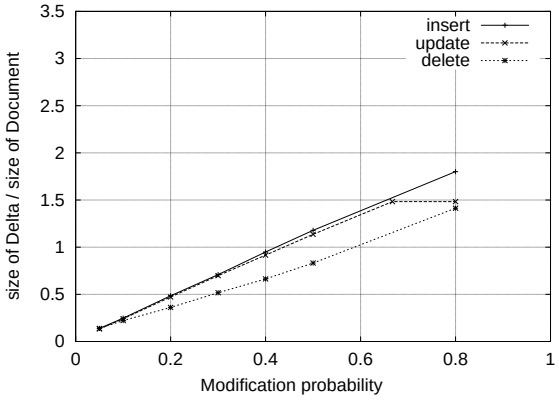
As expected, a critical aspect is the size of the delta. We briefly discuss some experimental results on this size.

On the size of deltas We measured deltas generated by a change simulator. The change simulator takes as input a document and produces a new document and the delta between the old and the new. The simulator uses 3 parameter, α, β, γ, respectively the delete, update and insert rates. The simulator randomly deletes $N \times \alpha$ nodes where N is the total number of nodes. Similarly, it inserts $N \times \gamma$ nodes and updates up to $N \times \beta$ text nodes (If there

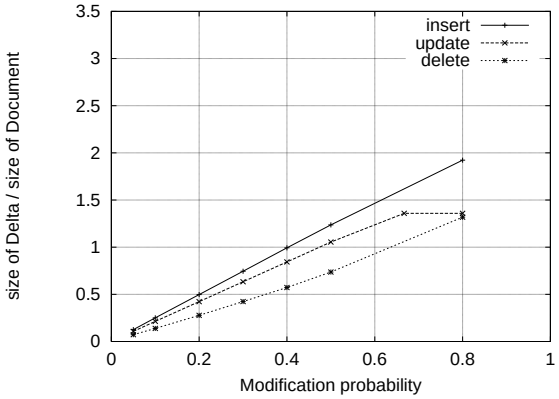
are fewer text nodes, it updates all text nodes.) To simplify, the simulator ignores attribute updates here. They roughly behave like element updates.



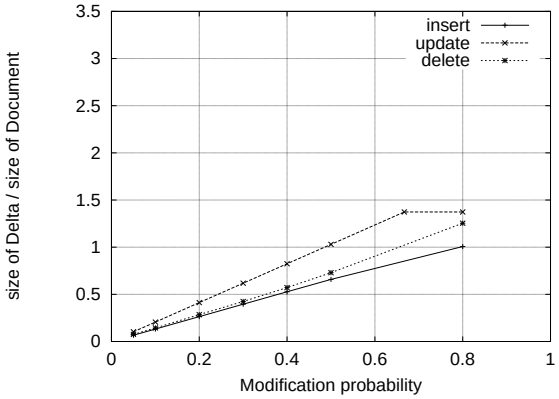
(a) Modifications on a very small document (0.5K)



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



(c) Modifications on a medium document (45K)



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

Figure 8: Rate between the size of the XML delta and the size of the XML Document

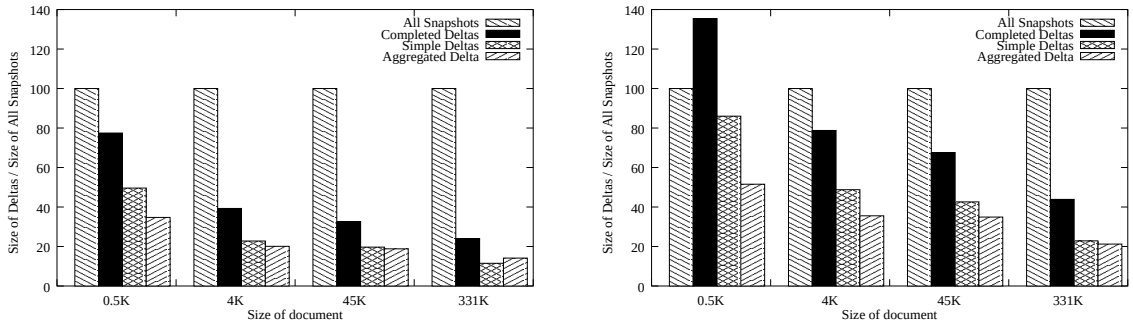
Figure 8 gives the rate between the delta documents and the original documents for documents of 4 increasing sizes. Each point was computed from 1000 test results. The good news is that the size grows linearly, reasonably in the rate of changes. For small documents, it reaches faster the size of the document. Clearly, when the document is small, any overhead costs a lot as a fraction of the document size. The measures given here are limited in several ways:

1. Changes are made by the simulator. This allows a fine controle over changes. However, we plan to now conduct experiments with real data gathered on the web.

2. We measure here the size of deltas as text files. It would also be interesting to know the size of the changes in the physical representation.

3. These results do not show the impact of the structure of the document (e.g., deep vs bushy, regular vs. irregular) on the size of the delta. We plan further study in that direction.

When the size of the delta is important compared to that of the document itself, it may be cheaper to simply save the versions. However, note that the delta is more informative since it also keeps change information that would have to be recomputed if we store simply the versions. Finally, Figure 9 illustrates the relative sizes of the different storage policies for a



(a) 15% Modification on the Document

(b) 30% Modification on the Document

Figure 9: Relative sizes of deltas on a sequence of 10 snapshots of a document

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 rate of 15% (5% update, 5% insert, 5% delete), the second one for a rate of 30% (10%, 10%, 10%). The different storage policies are:

storing all the versions: All the snapshots of the document are stored in the repository.

completed deltas: The last version and the sequence of completed deltas are stored.

simple deltas: The last version and the sequence of simple deltas are stored.

aggregated delta: The last version and the aggregated delta are stored. The aggregated delta corresponds to the aggregation of the 10 completed deltas computed over the sequence of 10 changes. (The aggregated delta is a completed delta.)

Simple deltas are obviouly less costly in terms of space than completed ones (about 40% in the test example) but they do not keep as much information. When recomputing old version is not needed, simple deltas should be prefered. Aggregated deltas can provide enormous saving in space (up to 50% in the test example) at the cost of loosing some granularity in versioning.

More experiments are planed for the near future. First, we need to gather statistics on XML data such as average number of pages in a site, average page size, average change frequency, average change size, etc. We plan also to study the use of XML IDs and other data from the document as XIDs. This is potentially the source of large space and processing savings. Also, from a user viewpoint, they carry more meaning that system-generated IDs. An interesting issue is the choice of such identifiers in a learning phase.

Change GUI A graphical user interface to manage changes has been designed [24]. This GUI works in the spirit of change-management interfaces such as [16]. An old version of a document and its XID-map are stored locally. The delta is obtained via HTTP from Xyleme server. Based on that, the GUI builds the new version of the document. If requested, the GUI can present the old and new version in a change presentation window.

Consider Figure 10. The interface proposes 4 main areas:

- a split pane, in the center, with the two documents in text mode. (Colors are used to indicate insertions, deletions, changes). the GUI use color to show evolution
- a tree viewer, on the left, that presents the structure of the document. (Colors are also used to identify the changes.)
- a radar view, on the right, to see localisation of modifications in the whole document as well as the current location.
- a magnifying glass ("Loupe") to see the modifications only in the text mode view.

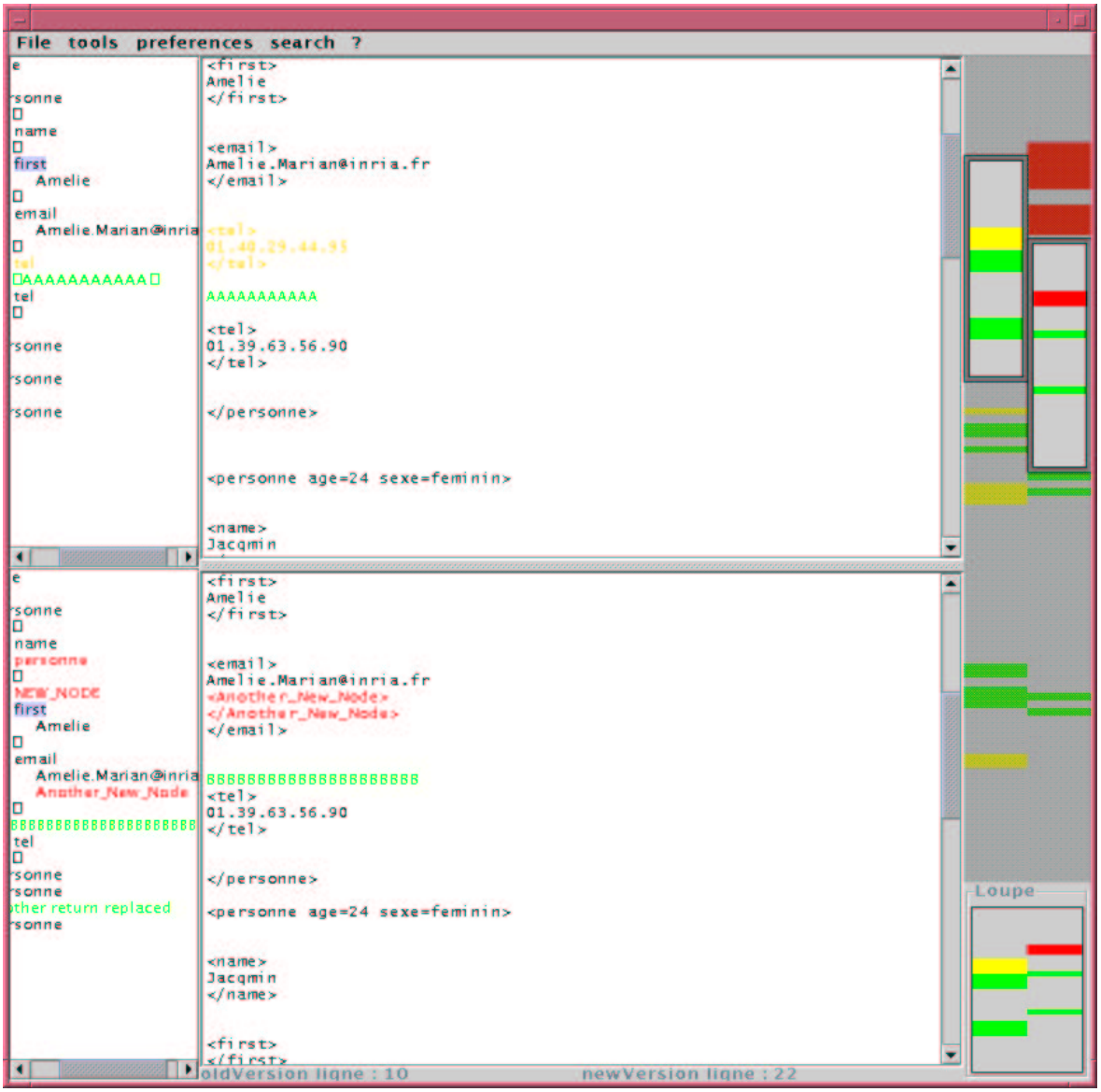


Figure 10: The Change-Management Graphic User Interface

Main functionalities of the GUI are the possibility to (de)synchronize those components while investigating the changes to some data.

We are now using this versioning technology to monitor document changes in a query subscription system. The work presented here is preliminary. In particular, the foundations of change combination (perhaps an algebra of deltas) should be studied in more depth. Furthermore, we believe that there is a lot of room for optimization in the storage of delta, e.g., using compression techniques [21]. Finally, 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 archiving strategies for it.

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

References

[1] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. In *Proc. Int. Conf. on Very Large Data Bases*, 1999.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publisher, October 1999.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1, 1997.

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

[5] Vincent Aguiléra, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, and Fanny Wattez. Querying xml documents in xyleme. In *to appear in the proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.

[6] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In *Proceedings of the 16th Internationnal Conference on Very Large Databases*, Brisbane, Australia, 1990.

[7] Xyleme Change Group. Query Subscription in an XML WebHouse, 2000.

[8] Xyleme Change Group. XML Diff, 2000. in preparation.

[9] S. Chawathe. Diff on trees. http://www.cs.umd.edu/ chaw/stdiff/.

[10] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

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

[12] J. Chen, D. DeWitt, F. Tian, and Y Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for xml. http://www.w3.org/TR/NOTE-xml-ql/.

[14] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *Proceedings of the ACM SIGMOD Intl Conference on Management of Data*, 1996.

[15] Doubletwist. http://www.doubletwist.com.

[16] Code Warrior : File Compare. `http://www.metrowerks.com/`.

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

[18] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. Technical Report 8/99, University of Mannheim, 1999. available at `http://pi3.informatik.uni-mannheim.de/publications/techrep899.ps`.

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

[20] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proceedings of the International Conference on Very Large Databases*, Bombay, India, September 1996.

[21] H. Liefke and D.Suciu. XMill : an Efficient Compressor for XML Data. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[22] Laurent Mignet, Mihai Preda, Serge Abiteboul, Sébastien Ailleret, Bernd Amann, and Amélie Marian. Acquiring XML pages for a WebHouse. In *proceedings of Base de Données Avancées conference*, 2000.

[23] Mind-it. http://mindit.netmind.com.

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

[25] J. Robie, J. Lapp, and D. Schach. Xml query language (xql). http://www.w3.org/TandS/QL/QL98/pp/xql.html.

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

[27] J.T.L. Wang, K. Zhang, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, 1994.

[28] 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.

[29] Xyleme Project. `http://www-rocq.inria.fr/verso/Xyleme.html`.