Change-Centric Management of Versions in an XML Warehouse*

Amélie Marian   Serge Abiteboul†   Gregory Cobena†   Laurent Mignet†

ABSTRACT

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 deltas and unique identifiers. We present a novel way of allocating persistent identifiers to nodes. We analyze several implementation issues and compare different storage policies. Based on some requirements (certain functionalities that we want to support efficiently), we motivate our choice of a specific storage policy. We store the last (most current) version and a document containing a list of "completed" forward deltas with some information on the persistent identifiers.

This work is part of the Xyleme system. Xyleme is developed on a cluster of PCs under Linux with Corba communications. The part of the system described in this paper has been implemented. We also mention first experiments.

Keywords

XML, Version, Warehouse, Web, Persistent Identifiers, Delta, Temporal Query, 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 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 deltas and an efficient storage policy for it. We describe the XML versioning system we implemented.

XML is becoming the new standard for semistructured data exchange over the Internet [25, 1]. This work is part of the Xyleme project [28] that is studying and building a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing the XML data available on the web. Because of XML, it is becoming possible to deliver higher level services that are difficult or impossible to support with the current HTML technology. In particular, with XML, it is possible to offer more sophisticated query languages [11, 24], provide some automatic semantic integration and achieve monitoring at a finer grain than just the document level by using the structure of the document. In the present paper, we consider the issue of version management support for the Xyleme system.

Versions serve many purposes in such an XML warehouse environment. In a standard manner, we may want to version a particular document or (part of) a web site. Also, the Xyleme system supports continuous queries, i.e., queries that are evaluated at a regular frequency, and it may be useful to also version answers. Like in a standard versioning system, one would like to query versions (of documents or query answers) or reconstruct the value at some previous time. The following other two uses of versioning are also essential in our context:

- Deltas: we may want to obtain the modifications between two points in time, typically between some version at time t that a user may have stored and the current version. This may be used, for instance, by a user interested in understanding the changes since the last time

*Work supported in part by R.N.R.T.
†INRIA-Rocquencourt 78153 Le Chesnay France
email: firstname.lastname@inria.fr

she observed some particular data.

- Query subscriptions: The Xyleme system provides a subscription system [21] that allows monitoring at the element level, e.g., notify me when a new product is introduced in a particular catalog under category *electronic*. Such monitoring is based primarily on the analysis of consecutive versions.

The system acquires XML data from the web and maintains it up-to-date [19]. Thus, for a particular document, a sequence of snapshots [3] is obtained. Also, each continuous query is evaluated regularly and produces a sequence of query answers, i.e., a sequence of snapshots of an XML document as well. The modifications that occurred between time t_{i-1} and time t_i can be computed using a *diff* algorithm. There has been a lot of work on such algorithms on relational, e.g., [17], or tree data, e.g., [26, 8]. Intuitively, they focus on computing a minimum *edit script* between the versions at time t_{i-1} and t_i. We developed our own *diff* algorithm. (Its presentation is not within the scope of this paper.) 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 will 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 of a storage policy for versions adapted to the control of changes. Our logical representation is based on *deltas* in the style of [12]. Its two main components are:

1. Completed deltas. Our deltas are reminiscent of traditional ways of representing logs in database systems. Simple deltas are lossy and cannot be inverted. For instance, they typically include the new value of a string and ignore the old one. We introduce *completed deltas* which are deltas containing additional information so that they can be reversed.

2. 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.

Our physical storage policy is based on storing the current version, an *XID-map* to handle XIDs and a single XML document containing all *forward unit completed deltas*. A forward unit completed deltas 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 XID-map is a novel concept that allows to attach persistent XIDs to every node in a storage-efficient manner. The choice of this storage policy was motivated by an analysis of the requirements of the change management in Xyleme that 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 use a delta compression technique to periodically recover space.

As previously mentioned, the present work is a report on the on-going work about versioning performed in the context of the Xyleme project. The project is still in an early phase and we only start obtaining large quantities of XML data. We implemented the versioning system and started performing tests to validate our choices. We briefly present some experimentation.

In Section 2, we present the logical representation based on XIDs and (completed) deltas. In Section 3, we give a synthesis of the possible uses of our deltas. We discuss our storage policy in Section 4. In Section 5, we give a brief description of our implementation and present experimental results. Section 6 describes shortly the Graphical User Interface. The last section is a conclusion.

2. LOGICAL REPRESENTATION

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

The document model 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 such as the *VCR* in the example (see Figure 2), 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 (here $update(8, 150)$).
- Suppose we want to reconstruct the history of the description 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.

Observe that XIDs considered here form a *logical* concept that can be used to denote an XML node in a persistent manner. Any physical means of denoting, in a persistent manner, particular node can be used. We will discuss particular implementations of XIDs further. For now, we will assume that XIDs are from the set $\mathcal{N}$ of integers. Values and labels are from a set $\mathcal{Q}$, e.g. the set of strings.

To simplify the presentation, we use (without loss of generality) in the present paper, 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 *label mapping*, assigns a label 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 1. 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.

An important aspect of the model is that the same node in two versions of some document will be associated the same integer.

REMARK 2.1. *In the complete model, we need to distinguish between text nodes and element nodes because they behave differently for some of the change operations we will study. A text node cannot have children, has a fixed label and a value that can be updated. Element nodes on the opposite, have no value that can be updated, but their label may be updated and they may have children. We also need to consider attributes nodes that also present a specific behavior. We will mention later how attribute nodes are handled. The system we are implementing deals with the general XML model.* □

Deltas We next turn to the presentation of the logical representation of changes. As mentioned before, the starting point is a sequence $V_1, ..., V_n$ of snapshots, i.e., the versions of some data at time $t_1, \ldots, t_n$. See Figure 2 (ignore the node identifier for the moment and the *diff*). We use a *diff* algorithm that extracts the changes between two consecutive versions based on some heuristic, namely, minimizing[1] the *delta*. For instance, consider the first two snapshots in Figure 2. The result of the *diff* algorithm that we use [7] is shown in Figure 3 where **A** and **B** denote XML trees. The doted lines represent matchings between XML nodes of the two versions discovered by *diff*. Due to space limitations, we will not present our *diff* algorithm in this paper.

Let $\vec{V}$ be a sequence of snapshots. The*diff* algorithm first provides a matching between nodes in consecutive versions. Given V_i, V_{i+1}, it also provides the delta between V_i and V_{i+1}, $\Delta_{i,i+1}(\vec{V})$, that we will note $\Delta_{i,i+1}$ when $\vec{V}$ is understood. Finally, 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. *label(n,v)* that changes the value of the label of element node n to v

Changes are 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 V_i 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 assume 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 assume that all the deltas we deal with are consistent. Furthermore, we assume that the *diff*

[1]We will have operations such as subtree insertions. Clearly, such an operation may be quite expensive if the tree is large. The measure of deltas we use has to take this into account.

```
<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>
```

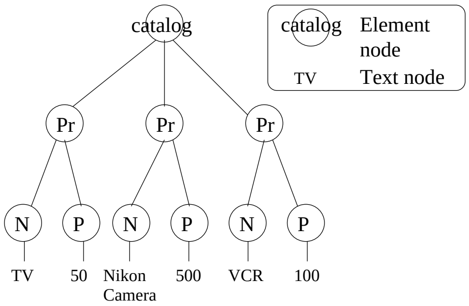
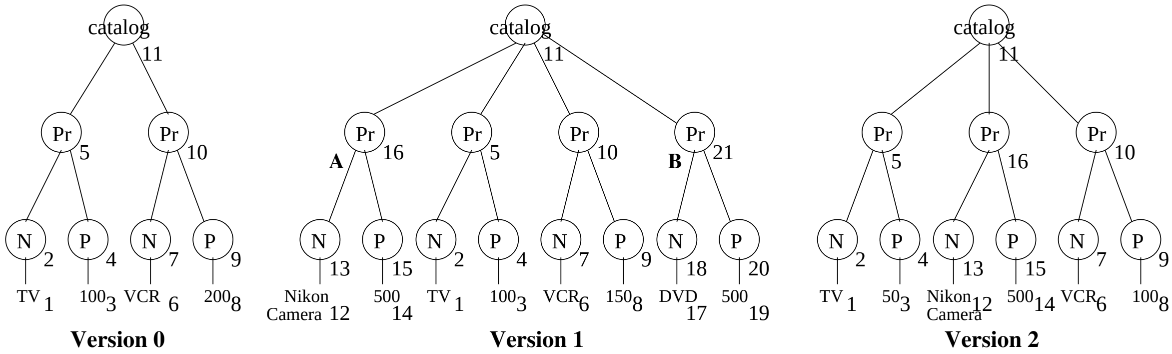


Figure 1: An XML tree representation



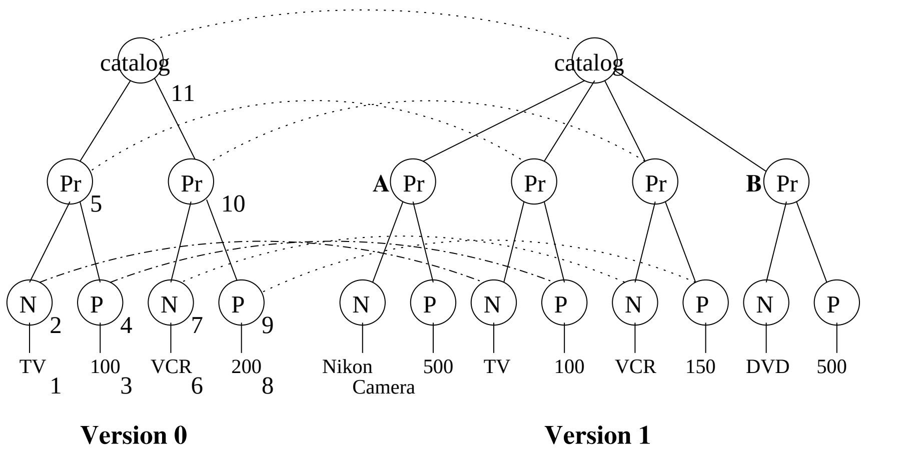
Diff (V0, V1)

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 2: Snapshots of the catalog document using XIDs



Diff (V0, V1)

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

Figure 3: Result of *diff* between Versions 0 and 1 of the catalog

algorithm produces deltas that are consistent as operators, i.e., for each i,

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

Remark 2.2. *This model of changes is rather simplistic. Our implementation does consider a larger set of operations that we briefly mention now:*

1. *A node may be at the same time involved in a move and an update or a move and a change of label. To simplify the processing the operations, we also consider* move_update *and* move_label *operations. With these new operations, one can see that every operation "belongs" to a single node.*

2. *We also consider operations on attributes: attribute insertion, modification and deletion but not move since an attribute is by definition attached to a single node.*

All the operations mentioned so far are used in our implementation. A limitation is that our current diff *algorithm ignores any* label *operation but the change of the root label that is needed to be able to transform an arbitrary document into any other document. Finally, note that one could consider a richer set of operations. For instance, one could introduce 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.* □

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 changes, it may be more complex because of positions, e.g.,
$\{insert(24, 1, X)\} \circ \{insert(24, 1, Y)\} \equiv$
$\{insert(24, 2, X), insert(24, 1, Y)\}$

Operations on the same node may be aggregated. For instance if a text node is inserted in $\Delta_{i-1,i}$, then updated in $\Delta_{i,i+1}$, the corresponding operations in the aggregated delta $\Delta_{i-1,i+1}$ is an insertion of a text node with its new value. To see another example, a text node updated in $\Delta_{i-1,i}$ and moved in $\Delta_{i-1,i+1}$ will lead to a *move_update* operation in $\Delta_{i-1,i+1}$. The aggregation of deltas is not trivial because of positions and because of the combinations of the different operators.

Based on composition, we can define:

$$\begin{array}{ll} \Delta_{i,j} = \Delta_{i,i+1} \circ \cdots \circ \Delta_{j-1,j} & \text{for } i < j \\ \Delta_{i,j} = \Delta_{i,i-1} \circ \cdots \circ \Delta_{j+1,j} & \text{for } i > j \end{array}$$

It can be shown that:

$$\begin{array}{ll} V_j = V_i \circ \Delta_{i,i+1} \circ \cdots \circ \Delta_{j-1,j} = V_i \circ \Delta_{i,j} & i < j \\ V_j = V_i \circ \Delta_{i,i-1} \circ \cdots \circ \Delta_{j+1,j} = V_i \circ \Delta_{i,j} & i > j \end{array}$$

Observe that we can aggregate modifications from several deltas on the same document to get a new delta involving a larger time interval. 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 $\Delta_{i,j}$ 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(8, "100")*. The delta ignores the old value of node 8 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) or for a change of label. To see why this is important in practice, consider the following four possible representations of a sequence of snapshots:

(a) $V_1, \Delta_{1,2}, \ldots \Delta_{now-1,now}$;

(b) $\Delta_{2,1}, \ldots \Delta_{now,now-1}, V_{now}$;

(c) $\Delta_{1,2}, \ldots \Delta_{now-1,now}, V_{now}$;

(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, we would appreciate to have available V_{now} and the forward deltas, i.e., representation (c).

This motivates the introduction of completed deltas.

Completed Deltas form 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 Section 4.

The operations in completed deltas are as follows:

1. $\overline{\textit{delete}}$*(n,k,T)* that deletes the XML tree T from a child of node n in position k.
2. $\overline{\textit{update}}$*(n,v,ov)* where ov is the old value.
3. $\overline{\textit{insert}}$*(n,k,T)* that inserts the XML tree T as a child of node n in position k.
4. $\overline{\textit{move}}$*(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. $\overline{\textit{label}}$*(n,v,ov)* where ov is the old value.

Again the complete model also includes operations for attributes and combined operations such as $\overline{\textit{move\_update}}$.

Table 1 gives the forward and backward *simple unit deltas* (Δ) from Version 1 to Version 2 from Figure 2 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:

$$\begin{array}{ll} (\overline{\Delta}_{i,j})^{-1} = \overline{\Delta}_{j,i} & ((\overline{\Delta}_{i,j})^{-1})^{-1} = \overline{\Delta}_{i,j} \\ \overline{\Delta}_{i,j} \circ \overline{\Delta}_{j,k} = \overline{\Delta}_{i,k} & \overline{\Delta}_{i,j} \circ \overline{\Delta}_{j,i} = \phi \\ (\overline{\Delta}_{i,j} \circ \overline{\Delta}_{j,k})^{-1} = \overline{\Delta}_{i,k}^{-1} = & \overline{\Delta}_{k,i} = \overline{\Delta}_{k,j} \circ \overline{\Delta}_{j,i} \end{array}$$

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

$$\begin{array}{ll} \overline{\Delta}_{i,j} = \overline{\Delta}_{i,i+1} \circ ... \circ \overline{\Delta}_{j-1,j} & i < j \\ \overline{\Delta}_{i,j} = \overline{\Delta}_{i-1,i}^{-1} \circ ... \circ \overline{\Delta}_{j,j+1}^{-1} & i > j \\ V_i = V_n \circ \overline{\Delta}_{n,i} & \end{array}$$

Thus from V_n and the list of forward unit deltas ($\overline{\Delta}_{i,i+1}$ with $0 \leq i \leq n-1$), we can reconstruct all the versions and all possible completed deltas.

REMARK 2.3. *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 [8, 9] using annotations to describe changes. To illustrate, the history $H_{0,2}$ for the running example is given in Figure 4 using a representation in the style of 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.* □

To conclude this section, observe that storing delta is expensive. One can imagine having various level of services for different data managed by the system. 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 and support some queries over changes without fully supporting version management. Finally, for a large quantity of data, no temporal support may be provided. In this last case, we can still provide some form of monitoring when the new version of a page is fetched (the old version is still available). But this done, the old version is discarded and no versioning information is kept or some minimum one such as the fingerprint of the page to be able to detect whether the page changed at the next fetch.

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 consider in this section how deltas may be used.

We consider the following uses of deltas:

Learning about changes: A Xyleme client may request some documents, collections of documents, or query result, say V_i at time i and store it on its local computer. The client may then request $\Delta_{i,now}$ to refresh its data and/or see what changed. We provide a GUI [22] 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. (See Section 6.) The same technology may be used to mirror or monitor 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 [27, 14] 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., [16].

Subscribing to changes: We are also implementing a subscription system. 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. Since documents are received one at a time, this monitoring can only be made on changes involving a single document. Related

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

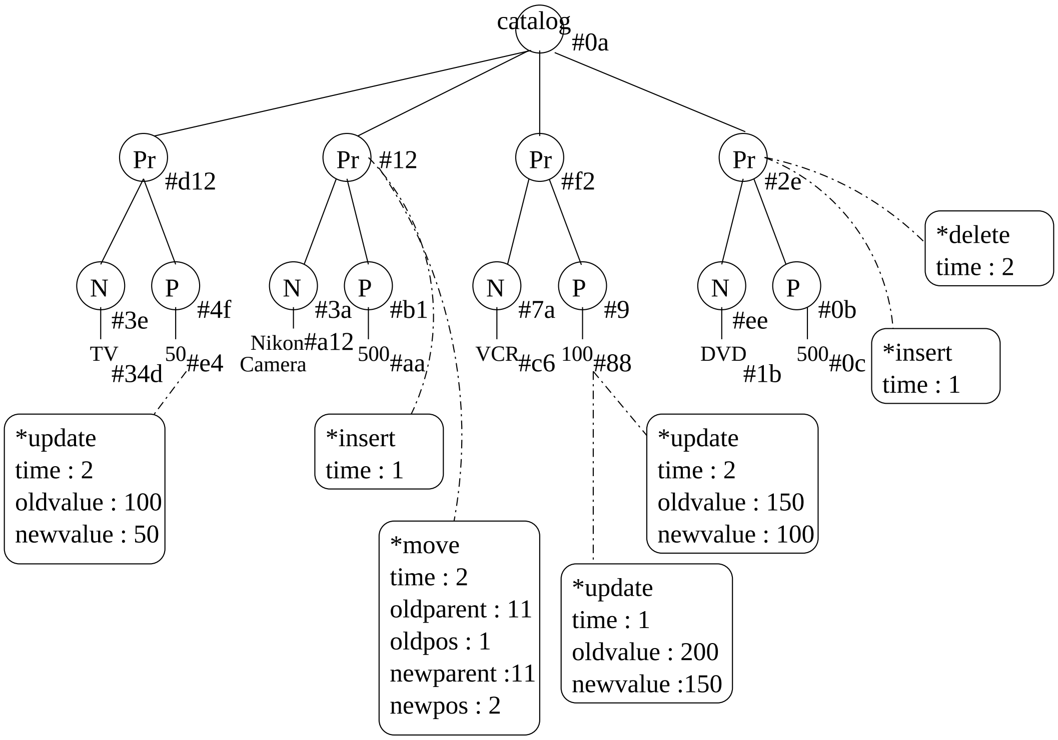


Figure 4: DOEM-style history of the document from Figure 2

works on subscription systems that use filtering tools for information dissemination have been presented in [29, 5].

ICE also provides a protocol for notifications. New services such as Netmind [20] or search engines such as Northernlight [23] allow to monitor changes at the page level in the web. We 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 may ask for the list of all items recently introduced in a catalog, or for the new movies showing in London this week. Since deltas are XML documents, such queries are regular queries over documents.

Versioning: we can obtain an old version of a document or part of one. Perhaps, more importantly, we believe that versioning a continuous query [10] (i.e., queries that are asked regularly) is a most useful feature in practice. A user may want to ask regularly the same query but may be mostly interested in what changed in the answer.

Temporal Queries: This is another way to query changes using a logical representation of the history of data and a Temporal Query syntax. The user does not have to know about the representation of changes or about XIDs and may query the data history directly. This allows to express more complex queries in a simpler manner. This is in the spirit of temporal query languages and more precisely of Chorel [9]. Example of such queries are: what are the products of which the 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.

To conclude this section, we examine next the issue of queries over change vs. queries over the history.

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 [2] and Chorel 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 yielded by this query is:

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

Using the delta representation, 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"
```

4. PHYSICAL ORGANIZATION

In this section, we first present the physical organization of versioned data in Xyleme. We then describe various alternatives and motivate the one we selected. Finally, we discuss in more details some implementation aspects of the storage and, in particular, the management of persistent identifiers.

4.1 Physical organization

We decided to store the last version (the current one) in the repository as well as a document containing the *forward completed unit deltas*, i.e., $(\overline{\Delta}_{i,i+1})$. We also store the *XID-map* of the current version that provides the means to obtain the XID of the current nodes. This suffices to reconstruct the sequence of snapshots. Since we use completed deltas $(\overline{\Delta})$, no information is lost. This choice was motivated by the general requirements of Xyleme. Recall, for instance, that it is 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 a native XML repository that stores XML data as tree [15]. 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. 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. This is typically faster (in the repository we use as well as in many repositories) than updates to data in place. This is an important aspect of the choice of a storage policy.

4.2 Advantages over other policies

We next discuss alternatives we considered and compare them to the organization we selected.

Storing 1st + last 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 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 backward deltas 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}, thus reconstruct the entire sequence. With this, getting V_{now} is immediate and the computation of V_i for some recent i is rather efficient. 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.

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 4. 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

As mentioned before, we store the last version and the completed forward deltas. 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 in the introduction and throughout the present section.

4.3 Implementation aspects

The $\overline{\Delta}$'s are stored as an XML document. See Figure 5 for an XML representation of the unit completed delta from Table 1. The XML delta keeps all the information needed for reversing deltas: for instance, for each update, we keep the XID of the node, the old and new values. We will describe in Section 4.4 the notion of XID-map that is used to associate persistent identifiers to the nodes.

```
<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 5: XML $\overline{\Delta}$ from Figure 2

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 on 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 their XIDs. The XID map of the new version is constructed.
4. The completed unit-delta is computed.
5. The new version and the new XID-map are stored. The unit 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 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 that has not changed since V_0 is updated, we have imperatively to keep its old state in the delta. Otherwise, we loose information. On the other hand, once it has changed, keeping the old state is redundant since it is already in the delta. By state, we mean here 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 already in the delta.

Several storage policies may be considered:

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

Each of these storage policies has advantages and drawbacks. Redundant Storage is wasting space since redundant information is stored. On the other hand, the processing of a new version 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. The use of pointers saves storage space but updating the delta when a new version is checked in may be considerably costly. 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 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 use the Forward Pointers strategy. We process 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. This allows us to diminish the main drawback of the Redundant Storage technique: storage space. The result in the store is like the results of a Backwards (or a Forward) Pointers storage policy. Moreover, the processing of a new version is very fast. 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 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.

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. 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 allows 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.

In the current implementation, intermediate versions and archiving are not supported. We did implement the aggregation algorithm.

4.4 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 element and text 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. (See Section 4.5.) Thus, XIDs are only used for change management and in particular by the versioning module.

Allocating XIDs

Since each XML node of a versioned document has an XID, 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 that 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 to a new node. 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 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 of processing queries.

In fact, it is more convenient and less space consuming to keep the assignment of XIDs to XML nodes separately as a list of integers. More precisely, we use the notion of XID-map that is based on the listing of XIDs of the nodes in some particular traversal of the tree. We will see how a simple trick often produces large space savings. See an example of a tree using an XID-map in Figure 6.

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. The XID-map provides the XIDs for every node in the tree. 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. 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 $\theta_1 = 1..n \mid n+1$, 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 will always keep this initial identification.

Evolution When there are insertions, deletions and moves, the structure of the tree changes. For insertions, we assign new integers to the nodes in the inserted subtrees using again a postorder traversal for the inserted subtrees.

Consider θ_1. Suppose a subtree is deleted. It is easy to see that at this stage, the XIDs in a subtree consist in consecutive integers, say $i..j$. The XID-map is now $\theta_2 = 1..i-1, j+1..n \mid n+1$. 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

$$\theta_3 = 1..i-1, j+1..k, n+1..n+l, k+1..n \mid n+l+1$$

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

What is an XID-map good for? It allows to identify the nodes in a 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 a document. (Portions of the XID-map for deleted subtrees are also stored in the delta.) 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 refer to the 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[2] $n+1$. However, observe that, in general, the list is much smaller than n. To start, the list is small, size 3.

[2]Each element in the list is an integer. Since we know the highest integer *XIDnext* considered so far, we can use, inside a unit delta, a fixed-length coding of integers, e.g. order of $log(XIDnext)$ bits.

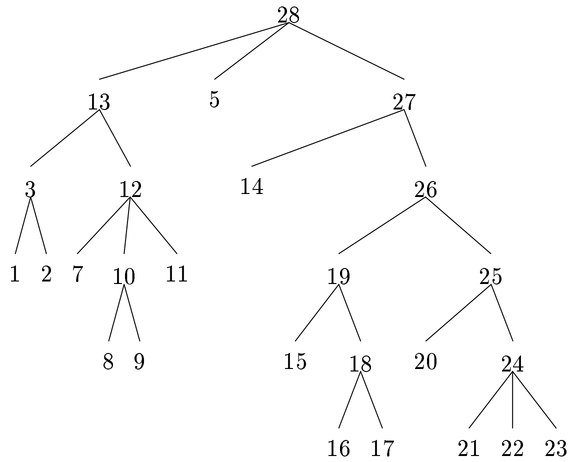


Figure 6: Tree with XID-map *(1-3,7-13,5,14-28)|29*

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 viewed as composed of a delete and 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. To conclude this argument on the size of the XID map, consider the example of a catalog. Typically, some text nodes will be updated, e.g., prices, descriptions. This would not modify the XID map. Products (reasonably large chunks of XML) will be inserted and deleted. Thus, one can expect the general XID-map to be reasonably small.

We next consider two ways to reduce the number of XIDs in a document. (Only the first one is used in our implementation.)

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 is essentially based on order to attributes that are by definition unordered in XML.

Using DTDs Some elements could be handled like attributes. Suppose for instance that the DTD states that a *product* consists of a sequence of four elements, a *name*, a *description*, a *price*, and a *picture*. We could limit the size of the XID-map by not assigning XIDs to such elements but treating them as components of identified elements. Information on such elements could be obtained by processing the DTD. In the prototype, to simplify, we assign XIDs to all elements and text nodes.

4.5 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. Observe that from a user viewpoint, they carry more meaning that system-generated IDs.

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. This 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 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 adoption of such IDs as XIDs relies on the assumption that the publisher of the document uses consistently the IDs in a persistent manner. The versioning system still has to be flexible since the XML standard does not enforce any persistency for ID attributes.

An interesting issue is the choice of such identifiers in a learning phase.

Identifying elements by their 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 *prefix/postfix 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 is less likely to change. So, we may decide to consider changes only to nodes with certain labels only. So, to continue with the catalog example, a change in the characteristics of a product would 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 elements that may be updated may again be provided via publication or computed in a learning phase.

5. IMPLEMENTATION AND EXPERIMENTATION

The management of versions has been implemented with the exception of the compression technique. For *diff*, we have implemented a simple *XMLdiff* [7]. We are working on a more elaborate version of it. The management of version is being integrated to Xyleme.

The Xyleme versioning module is, like the entire system, programmed in C++. XML Documents and deltas are stored in a repository tailored to the management of trees [15]. 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 *XMLdiff* algorithm 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 event read, which provides a very fast management of new versions.

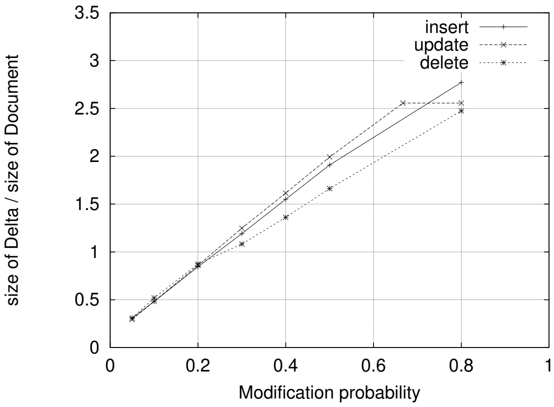
Since deltas are stored as standard XML documents, we can use Xyleme standard query processor [4] on them. To implement sophisticate temporal queries, we plan to use a compilation into standard queries over the delta followed by a "de-compilation" of the result. The use of the XIDs together with indexes 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 an early stage.

As expected, critical aspects are *diff* execution time and the size of the delta. We now briefly discuss some experimental results on the delta size. But first, it should be stated that the size of the delta depends critically on the quality of the *diff*. In particular, misses in matching large subtrees may result in important variations in the size of the delta. This is why we are still working on improving our *diff* algorithm besides the fact that a better *diff* brings better semantics to the users. A trade-off is that we also want a very efficient algorithm. We are considering using several *diff*s with various qualities according to the needs. For instance, for some documents, *diff* may simply be used to update the full text index. In such case, a mediocre quality is acceptable since the penalty is simply in terms of index updates. For versions or monitoring, we need a better quality since errors may impact the quality of services.

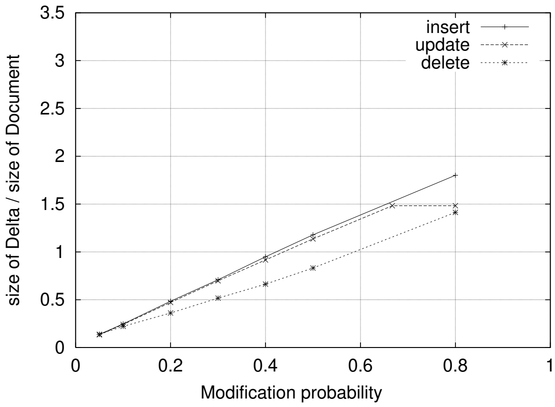
We next present some experimental result. Note that they use deltas generated using a change simulator, so they do not depend on the particular *diff* that is used.

On the size of deltas

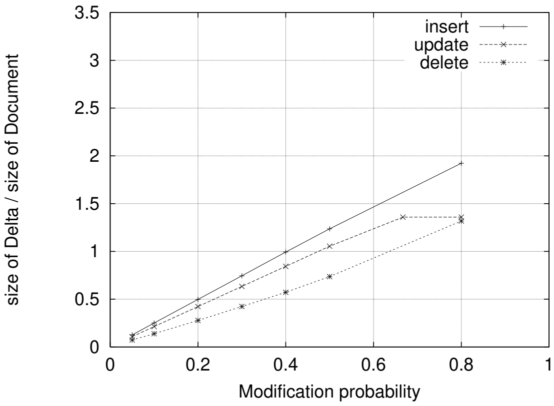
We measured deltas generated by a change simulator. The change simulator takes as input a document and produces a new document and the delta between the old and the new. The simulator uses 3 parameter, α, β, γ, respectively the delete, update and insert ratios. The simulator randomly deletes $N \times \alpha$ 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 modifications. They roughly behave like element modifications.



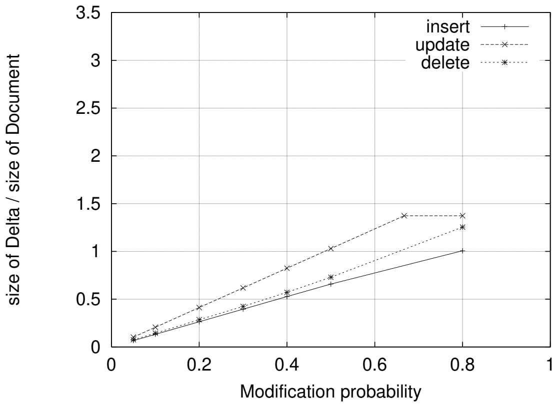
(a) Modifications on a very small document (0.5K)



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



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



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

Figure 7: Ratio between the size of the XML delta and the size of the XML Document

Figure 7 gives the ratio 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 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. The measures given here are limited in some ways:

1. Changes are made by the simulator. This allows a fine control over changes. However, the deltas produced by the simulator are in some sense minimum. We need to "measure" the quality of our *diff* algorithm to see the increase in size it would introduce.
2. It would be interesting to experiment with real data gathered on the web. In particular, this would allow to evaluate the pattern of changes for documents of the web.
3. We measured here the size of deltas as text files. It would also be interesting to measure the size of the changes in the physical representation. We estimated that the actual storage in the XML repository [15] requires a little less than the text size. Thus this factor does not have much impact on our measures.
4. Our results do not show 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.

Note that updates are applied to text nodes. In our test documents, approximately 66.6% of nodes were text nodes. Note also that 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, the delta is more informative since it also keeps change information that would have to be recomputed if we store simply the versions.

To conclude this analysis, consider Figure 8 that illustrates the relative sizes of different storage policies 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, the different storage policies are (from left to right):

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 obviously 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 preferred. 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.

6. CHANGE GUI

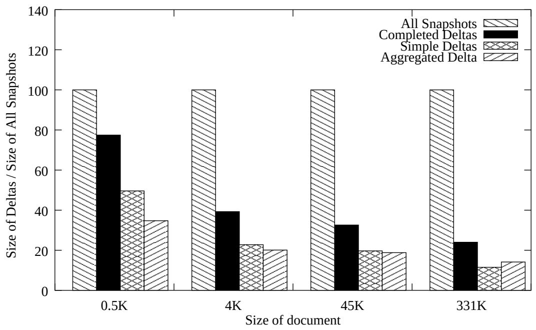
A graphical user interface to manage changes has been designed and implemented in Java [22]. This GUI works in the spirit of change-management interfaces such as [13]. An old version of a document and its XID-map are stored locally. The delta is obtained via HTTP from Xyleme server. Based on that, the 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 9. The interface proposes 4 main areas:

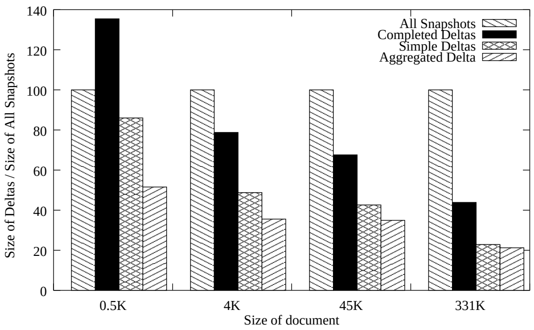
- a split pane, in the center, with the two documents in text mode. Colors are used to indicate insertions, deletions, changes.
- a tree viewer, on the left, that presents the structure of the document. Colors are also used to identify the changes.
- a radar view, on the right, to see localization of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

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

The text boxes have been added to improve the screenshot lisibility on paper; the operations are actually differentiated by coloring the corresponding text area (update in green, insert in red, ...).



(a) 15% Modification on the Document



(b) 30% Modification on the Document

Figure 8: Relative sizes of deltas on a sequence of 10 snapshots of a document

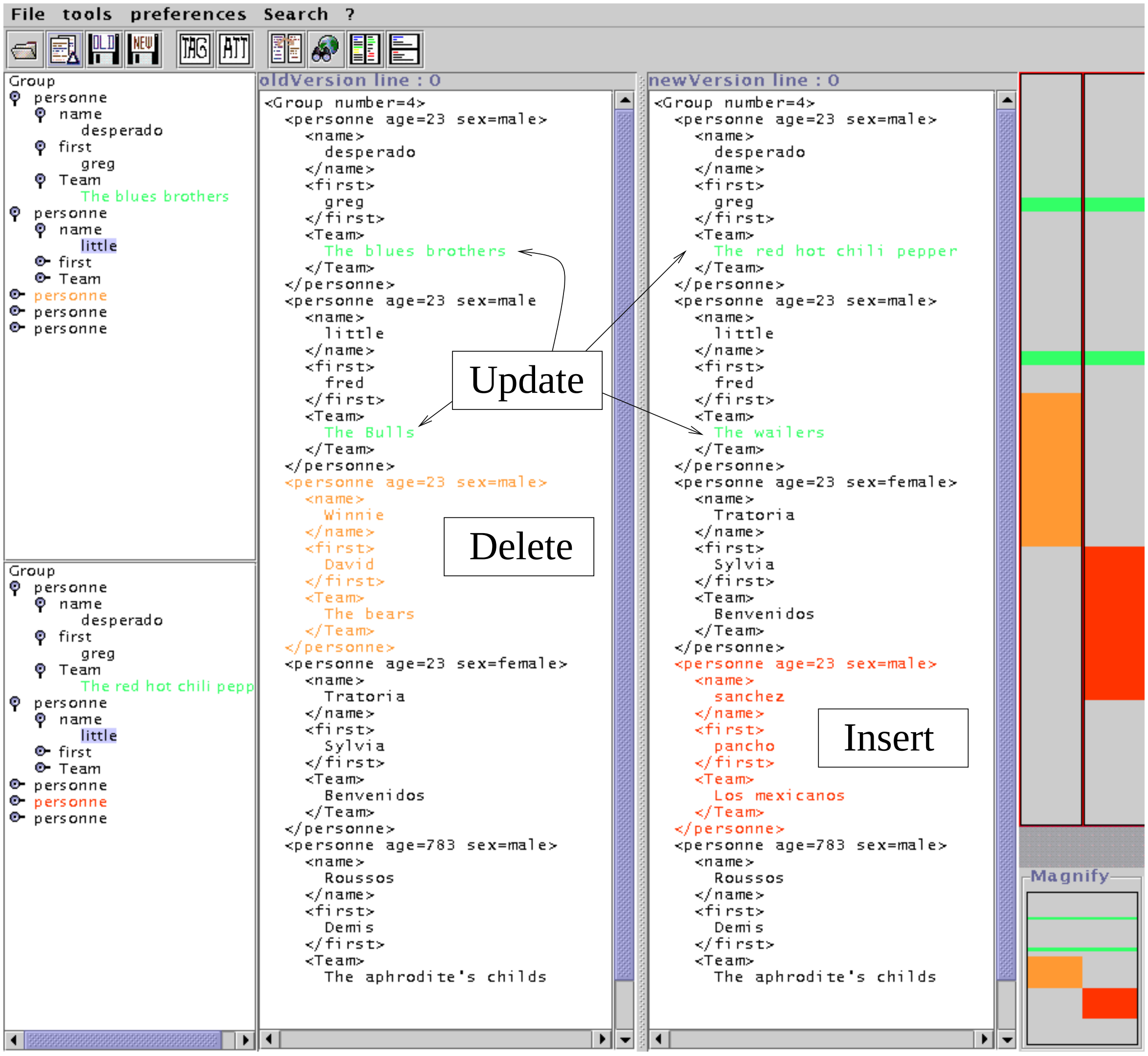


Figure 9: The Change-Management Graphic User Interface

7. CONCLUSION

The work we presented here is preliminary. The versioning technology has so far been mostly used in Xyleme system to monitor document changes for query subscriptions. Many questions remain open:

1. We plan to study the foundations of change composition (an algebra of deltas) in more depth.
2. The aggregation algorithm is not trivial. We need to understand better how it works and improve its performance.
3. We believe that there is a lot of room for optimization in the storage of delta, e.g., using compression techniques [18]. 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.
4. An issue is that of the processing of temporal queries based on the change-centric representation we proposed here.
5. It would be interesting to develop new strategies for allocating identifiers to nodes in an XML tree. We mentioned the issue of a strategy that would provide persistent identifiers that would also give indications on the structure of the data. We also discussed using semantic Ids from the document.
6. As already mentioned in the paper, we are working on *diff* algorithms adapted to XML data and we plan further experiments on the size of deltas.
7. We already mentioned the need to gather statistics about the size of deltas. To understand changes, we need to also gather statistics on change frequency, patterns of changes in a document, in a web site, etc.

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.

8. REFERENCES

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

[2] 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.

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

[4] 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.

[5] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 53–64, 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. XML Diff, 2000. in preparation.

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

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

[10] 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.

[11] 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/.

[12] 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.

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

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

[15] 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.`

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

[17] 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.

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

[19] 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.

[20] User-driven personalization and alerts. http://www.netmind.com.

[21] B. Nguyen, S. Abiteboul, G. Cobena, and L. Mignet. Query subscription in an xml webhouse, 2000. DELOS - to appear.

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

[23] Northern light news search. http://www.northernlight.com/news.html.

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

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

[26] 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.

[27] 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.

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

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