Efficient Encoding of Temporal XML Documents

Mohamed-Amine Baazizi, Nicole Bidoit, Dario Colazzo
Univ. Paris-Sud, Leo Team, & INRIA Saclay
last-name@lri.fr

***Abstract*—The management of temporal data is a crucial issue in many applications. Recently, XML has become the standard for data exchange and representation. Consequently, important efforts have been made on the development of temporal extensions for XML. This paper investigates how to generate or maintain space-efficient time-stamped documents. We formally define a notion of compactness which allows for comparing documents. Then, we present two methods. For the first one, called general method, no restriction is made on the evolution of the XML documents whereas for the second one, called update-based method, changes are assumed to be specified by updates. For both methods, the issue is to enable processing very large documents, to use existing engines and to comply to XQuery Update Facility. The two methods are compared in terms of space-efficiency. The update-based method produces time-stamped XML documents that are more satisfactory wrt space-efficiency than the general method. This goes to show that the update-based method effectively takes advantage of the updates.**

I. Introduction

The management of temporal data is a crucial issue in many applications such as finance, banking, travel reservations, geographical information systems etc. With the increasing use of XML for data exchange and representation, the issue of developing temporal extensions for XML is gaining importance.

Temporal extensions have been extensively studied in the relational framework. Chomiki et al. [1] pointed out the necessity of separating the abstract model from the concrete one. The abstract model views temporal data as a sequence of instances. Although this model facilitates the formal development of query, update, and constraint languages, from a practical point, storing a sequence of database instances requires a very large amount of space. The concrete model provides a space-efficient format to store temporal data.

Current work on temporal XML concentrate on time-stamp XML documents, a concrete model. Although many proposals have addressed the issue of querying time-stamp XML documents, there has been less in-depth investigation of how to efficiently build or maintain temporal XML documents, keeping track of data evolution over time.

In this paper, we follow the approach of [1] and provide both a notion of abstract temporal XML document and a notion of concrete temporal XML document. This allows to study the link between abstract temporal documents and their concrete encodings. Obviously, a given abstract temporal document may have several concrete encodings, some being more space-saving than others. Comparing concrete encodings wrt compactness is formally captured by introducing a partial order.

In this paper, we study ways to generate concrete encodings of an abstract temporal document with the following requirements. The first goal is of course to build or maintain time-stamp document as compact as possible. The second goal is to provide methods enabling to process very large documents. This requirement is particularly important as the size of temporal XML documents is expected to be much larger than the size of static ones and as we assume the documents to be processed by in-memory engines.

We develop two methods. The first one makes no assumption on the abstract temporal document for which an encoding is built whereas the second one assumes that the abstract temporal document is produced by a sequence of updates $u_1, \ldots, u_n$ from an initial document. The update language considered is the XQuery Update Facility (XUF) update language as described in [2], [3]. The main feature of the first method, next called the general method, is that it can be implemented in a streaming manner based on a SAX parsing [4] overcoming main memory space limitations. This feature unfortunately entails that, in some cases, the obtained encodings are not as compact as they could be.

The second method, next called the update-based method, is much more sophisticated. It is based on the type projection paradigm developed for XQuery [5], [6] and XUF [7] in order to overcome the main memory limitation of *in-memory* engines like Galax [8], Saxon [9], QizX [10], [11], and eXist [12]. The update-based method relies on pruning the time-stamp document over which an update should be integrated in order to load only part of the document that are touched or needed by the update. The pruning phase is very similar to that introduced in [7] and makes use of the schema (DTD) typing the document.

The benefit of extending the update mechanism of [7] is manifold. First, it enables for processing large time-stamp documents which would not be processed by in-memory engines due to their size. Second, any update engine can be targeted by such a scenario. Another advantage is that no rewriting of the updates is necessary. Last but not least, the encodings produced using the update-based method are much more satisfactory from the point of view of

space-efficiency than the encodings produced by the general method. This goes to show that the update-based method takes advantage of the information given by update in an efficient way.

Related work Temporal relational databases were extensively studied. Different data models and query languages were proposed (see [1] for a detailed survey). Chomicki et al. [1] were the first to consider the abstract and the concrete models of temporal databases. They also studied the way to encode temporal relational databases in a compact format and addressed efficiency issues wrt temporal queries [13].

Recently, important efforts have focussed on studying temporal extensions for semi-structured data and XML in particular. For instance, Chawathe et al. [14] extend the OEM model in order to keep track of updates. In the context of XML, several proposals have been developed for managing the temporal dimension (see [15] for a survey). Next, we focus more specifically on those addressing the issue of building or maintaining encodings of temporal XML documents.

Many proposals store the last version of the data together with *deltas* that represent the changes between the different versions. These deltas are often given by edit scripts. They are used for retrieving past versions starting from the document that is stored. Buneman et al. [16] pointed out that delta-based approaches raise practical and semantic problems. First, the cost for recovering past versions, which is required for query evaluation, is considerable and increases when new versions are added. Second, the information about changes provided by the edit-scripts is syntactical and quite often not meaningful. Time-stamp approaches are more effective than delta-based approach wrt to querying simply because temporal queries can be directly evaluated on time-stamp documents without the need to retrieve the different versions.

Buneman et al. [16] investigate data structures for managing historical information about scientific data. They develop a technique for merging versions of scientific documents into a single temporal document. Their technique is tailored to a special kind of data which has the following characteristics: the node order is not taken into account; each node is uniquely identified on the basis of its content (notion of key); and finally, insertion of new elements is the only kind of update which occurs frequently.

In our study, as opposed to [16], we do not focus on a particular type of data, node order is preserved which has a price to pay. Wrt changes, we cover both the case where no information is available wrt the evolution of the documents (it may not be the result of structured updates but rather be produced by editing or any other treatment over the documents) and the case where the changes are specified by XUF.

Our approach has some similarities with that of Rizzolo et al. [15]. They model temporal data by a DAG and provide two mappings from their DAG structure to XML. A specific update language is provided for specifying temporal evolution and this language is translated into operations over the graph-based representation. The space-efficiency issue is addressed although indirectly, as the authors study two mapping strategies: a non-replicating strategy which uses references to optimize the storage of subtrees shared by several nodes and a replicating strategy. The authors also develop several algorithms to eliminate inconsistencies that may be introduced by the use of references. The work presented in [15] does not consider changes other than those generated by updates. They consider a specific update language while we cover XUF and while our method is compatible with any XUF query engine. Although DAG may be more space-saving than time-stamp XML trees, the approach [15] does not address maintenance of very large temporal XML documents.

Other proposals like [17], [18] address the issue of managing and querying historical XML databases. They present a technique for computing documents annotated with time-stamps (called H-documents) starting from a sequence of versions. However, preserving node document-order in H-documents is not considered and updating H-documents is not fully addressed wrt update language and scalability.

Our presentation is organized as follows. After a short preliminary section providing the main notations, the abstract and concrete model for capturing temporal XML evolution is provided in Section III together with the compactness order. Section IV introduces the two methods for generating time-stamp documents from a sequence of static XML documents and compares the two methods wrt space-efficiency. Section V reports the results of a first bunch of experiments for validating the update based approach. We finally conclude and discuss future work in section VI.

II. Preliminaries

As in [3], XML documents are represented as stores. Next, I, J, K may designate sets (id-sets) or a list (id-seq) of store-identifiers denoted $\mathbf{i}$, $\mathbf{j}$ ...; $()$ denotes the empty id-seq; $I{\cdot}I'$ denotes id-seq composition, and the intersection of I and J preserving the order of the id-seq I is denoted by $I_{|J}$.

A store σ is a mapping from the set of store-identifiers I to constructors k defined as follows: $k ::= text[s] \mid a[J]$, where s is a string, a a label and J an id-seq such that $J{\subseteq}I$.

We only consider stores that correspond to XML trees and forests. An XML forest f over I is given by (J, σ, γ) where σ is a forest over I whose tree roots are given by J. The mapping γ over I is optionally used, in some context, to associate explicit identifiers to document nodes. The following notations are used:

– $dom(\sigma){=}I$,
– *lab*$(\mathbf{i}){=}a$ if $\sigma(\mathbf{i}){=}a[I']$, *lab*$(\mathbf{i})$=*String* if $\sigma(\mathbf{i}){=}text[st]$,
– *child*$(\sigma, J){=}\{\mathbf{i}' \mid \exists \mathbf{i}{\in}J, \sigma(\mathbf{i}){=}a[I]\ and\ \mathbf{i}'{\in}I\}$,

– $desc(\sigma, J)$=$\{\mathbf{i}' \mid \mathbf{i}' \in child(\sigma, J)$ or $\mathbf{i}' \in desc(\sigma, child(\sigma, J))\}$,
– $roots(\sigma)$=$\{\mathbf{i} \mid \neg\exists \mathbf{i}', \mathbf{i} \in child(\sigma, \{\mathbf{i}'\})\}$, and
– $f \circ f'$ is the concatenation of two disjoint forests f and f'.
An XML document d over I is given by (r, σ, γ) such that $roots(\sigma)$=r and σ is a tree.

Given a store σ over I, the *projection* on $J \subseteq I$ of σ, is a store over J, denoted $\Pi_J(\sigma)$, defined by: for each $\mathbf{j} \in J$, if $\sigma(\mathbf{j})$=$a[K]$ then $\Pi_J(\sigma)(\mathbf{j})$=$a[K_{|J}]$ otherwise $\Pi_J(\sigma)(\mathbf{j})$=$\sigma(\mathbf{j})$. The reader should pay attention to the fact that the domain and the "co-domain" of the $\Pi_J(\sigma)$ are equal to J and that, even if σ is a tree, $\Pi_J(\sigma)$ may not be a tree.

Finally, for the purpose of capturing time, we use time-stamps of the form $[t, t'[$ with $t < t'$ and $[t, Now[$ where t and t' are positive integers capturing instants of time and Now is a variable indicating the current instant.

III. Temporal XML Models

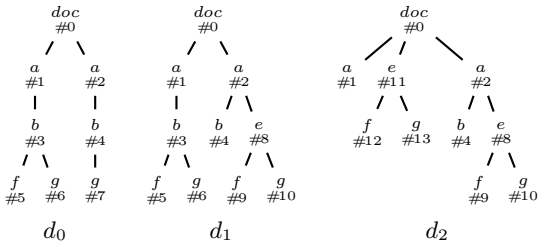


Figure 1. A temporal abstract document $\mathbf{d}$

The first part of this section follows the lines of [1] in providing two alternative models for capturing the evolution of XML documents.

Definition 3.1 (Abstract temporal document): An abstract temporal document $\mathbf{d}$ over the *temporal domain* $[0, n]$ is a sequence $d_0, \ldots, d_n$ of (static) documents such that, for each document d_t=(r, σ_t, γ_t), it is assumed that the mapping γ_t satisfies: $\forall \mathbf{i}, \mathbf{i}' \in dom(\sigma_t), \mathbf{i} \neq \mathbf{i}', \gamma_t(\mathbf{i}) \neq \gamma_t(\mathbf{i}')$.

The condition on γ_t enforces $\gamma_t(\mathbf{i})$ to behave as an explicit identifier for the node $\mathbf{i}$. These explicit identifiers are used to trace node evolution over time in the temporal document $\mathbf{d}$.

Fig. 1 presents an abstract document d_0, d_1, d_2. Explicit node identifiers are prefixed with #.

As already said, in practice, storing an abstract temporal document $\mathbf{d}$ may be quite inefficient because of replication of unchanged parts of the document. The concrete model aims at coping with this by introducing time-stamps (interval labeling) over document elements, providing the validity period of the elements. Changes are then encoded within a single document.

Definition 3.2 (Time-stamped Document):
A time-stamp XML document Δ=$(r, \sigma, \gamma, \tau)$ is an XML document enriched with a mapping τ:$dom(\sigma) \rightarrow Int$, satisfying the following properties:

– $\forall \mathbf{i}, \mathbf{j} \in dom(\sigma) \setminus \{r\}$, $\mathbf{j} \in child(\sigma, \{\mathbf{i}\})$ implies $\tau(\mathbf{j}) \subseteq \tau(\mathbf{i})$;
– $\forall \mathbf{i}, \mathbf{j} \in dom(\sigma)$, $\mathbf{i} \neq \mathbf{j}$ and $\gamma(\mathbf{i})$=$\gamma(\mathbf{j})$ implies $\tau(\mathbf{i}) \cap \tau(\mathbf{j})$=$\emptyset$.

The first condition is classical and ensures that time stamps are hierarchically consistent. The second one ensures that, two nodes $\mathbf{i}, \mathbf{j}$ representing the evolution of an element, cannot share the same validity intervals. $\gamma(\mathbf{i})$ are abusively called explicit identifiers, although for time-stamp document, γ is no more a bijection.

Fig. 2 depicts two time-stamp documents Δ and Δ'. For the purpose of making easier the manipulation of a time-stamp document, the explicit value of Now is registered at its root.

Temporal projection is now defined by extending XML projection already introduced for (static) XML documents in Section II.

Definition 3.3 (Temporal Projection): The temporal projection of a time-stamp document Δ=$(r, \sigma, \gamma, \tau)$ on a time point t, denoted $Snap(\Delta, t)$, is the (static) XML document d=$(r, \Pi_{T(\Delta,t)}(\sigma),\ \gamma_{|T(\Delta,t)})$ where $T(\Delta, t)$=$\{\mathbf{i} | t \in \tau(\mathbf{i})\}$ if $t \in \tau(r)$ and the undefined document $\perp$ otherwise.

The following definition links abstract and concrete representations in a natural manner.

Definition 3.4 (Sound & Complete Encoding):
Given an abstract document $\mathbf{d}$=$d_0, \ldots, d_n$, a time-stamp document Δ=$(r, \sigma, \gamma, \tau)$ is a sound, resp. complete, encoding of $\mathbf{d}$ if $\forall t \in \tau(r)$, $Snap(\Delta, t)$=d_t, resp. if $\forall t \in [0, n]$, $Snap(\Delta, t)$=d_t.

The two time-stamp documents Δ and Δ' of Figure 2 are encodings of the abstract document of Figure 1.

In the remainder, we consider sound and complete concrete encodings of abstract temporal documents and the term "concrete encoding" implicitly includes "sound and complete".

The following table summarizes the notation used throughout the article.

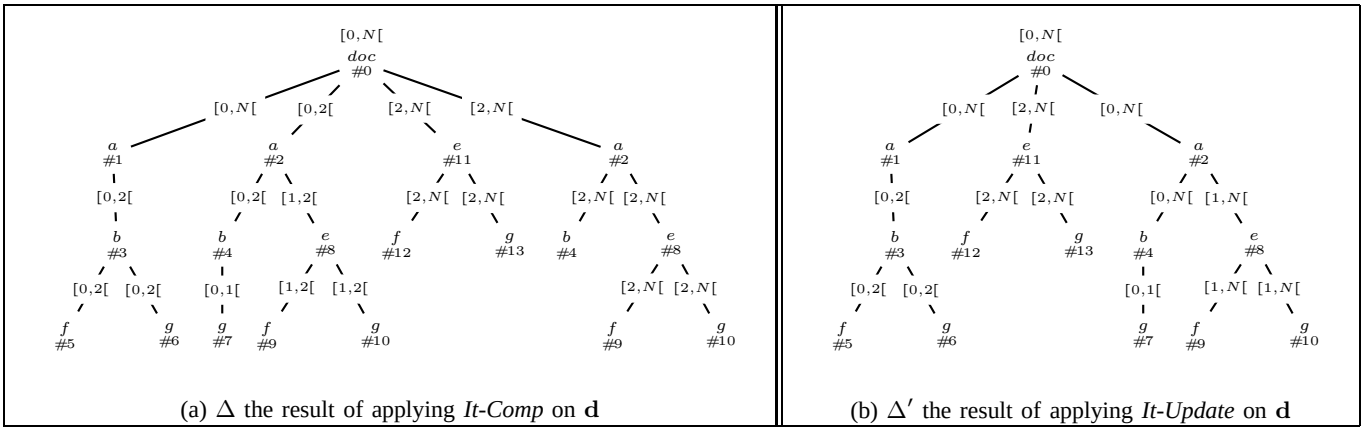
static XML doc.	d	DTD	D
abstract temporal XML doc.	$\mathbf{d}$	time-stamp XML doc.	Δ

Space-efficient concrete encodings

Obviously, an abstract document may have several concrete encodings. Some of them may be more space-saving than others because they have less nodes. Next, we formalize such a notion by introducing a pre-order $\preceq$ that enables to compare time-stamp documents.

Given a timestamp forest f=$(I, \sigma, \gamma, \tau)$ over J, a label l and an explicit identifier x (meaning x=$\gamma(\mathbf{i})$ for some $\mathbf{i} \in J$), $S_f(l, x)$ denotes the set of top-level store identifiers ($\in I$) associated with nodes labeled by l and whose explicit identifier is x: $S_f(l, x)$=$\{\mathbf{i} \in I \mid \sigma(\mathbf{i})$=$l[K]$ and $\gamma(\mathbf{i})$=$x\}$.

Definition 3.5 (Compactness Order): Consider two time-stamp forests f_1=$(I_1, \sigma_1, \tau_1, \gamma_1)$ and f_2=$(I_2, \sigma_2, \tau_2, \gamma_2)$. Below, l is a label in f_1 or f_2 and x is an explicit identifier i.e. $x \in \gamma_1(dom(f_1)) \cup \gamma_2(dom(f_2))$. $f_1$is *more compact* than f_2



(a) Δ the result of applying *It-Comp* on $\mathbf{d}$

(b) Δ' the result of applying *It-Update* on $\mathbf{d}$

Figure 2. Two encodings of the abstract temporal document $\mathbf{d}$

if:
(1) for each l and x, we have: $|S_{f_1}(l,x)|\leq|S_{f_2}(l,x)|$ and
(2) for each l and for each x, we have: $\Pi_{K_1}(f_1)\leq\Pi_{K_2}(f_2)$
where K_i=$S_{f_i}(l,x)\cup$*desc*$(\sigma_i, S_{f_i}(l,x))$

Our definition of compactness order is tree-based and relies on comparing the number of subtrees rooted at matching nodes. Let us go back to the example of Figure 2 presenting encodings of the abstract document of Figure 1: it can be checked that Δ' is more compact than Δ.

Comparing the compactness of two time-stamp documents that differ by their labels and temporal domains does not convey much information. Next, we will be comparing pairs of time-stamp documents that are concrete encoding of the same abstract temporal XML document. Under such a restriction, we have:

Property 3.6: $\preceq$ is a partial order.

Next, given an abstract temporal document $\mathbf{d}$=$d_0,\ldots,d_n$, we denote by $\top(\mathbf{d})$ the least compact concrete encoding of $\mathbf{d}$. Obviously, given a concrete encoding Δ of $\mathbf{d}$, we have: $\Delta \preceq \top(\mathbf{d})$.

IV. Building space-efficient encodings

This section focuses on building or maintaining a concrete encoding for an abstract temporal document. The issue is to ensure compactness of the encoding and, at the same time, to treat documents as large as possible. Two methods are investigated. The first one is developed without making any hypothesis on the abstract documents, whereas the second one makes the assumption that the abstract document is associated with a given sequence of updates. Although the methods are not comparable in general, we show that, histories can be encoded into time-stamp document in a significantly more compact manner.

A. Encoding general abstract documents

Let us consider an abstract document $\mathbf{d}$=$d_0,\ldots,d_n$. The method, called *It-Comp*, used to build or maintain a time-stamp encoding of $\mathbf{d}$, relies on an iterative process. The initial document is trivially transformed into a time-stamp document Δ_0=$d_0^{[0,Now[}$ and then assuming that Δ_{t-1} is the time-stamp document encoding $d_0,\ldots,d_{t-1}$, the document d_t is added to Δ_{t-1} in a specific manner to produce Δ_t=*Comp*(Δ_{t-1}, d_t, t).

Informally, *Comp* proceeds to a parallel and synchronized parsing of Δ_{t-1} and d_t and attempts to merge nodes in d_t with nodes in Δ_{t-1}.

The formal specification of *Comp* is given in Figure 3. Its inputs are: a timestamp forest F_s (some sub-forest of Δ_{t-1}), a XML forest F (some sub-forest of the static XML document d_t), and t referring to the validity time of d_t.
The time-stamp forest f^{int} is obtained by time-stamping each node of the forest f by the interval int. The time-stamp forest $\Phi^{a\leftarrow b}$ is obtained by replacing, in each occurrence of the time-stamp $[t,a[$ in Φ, the upper bound a by the value b.

Let us now explain the behavior of *Comp*.

Line 1 is the terminal case: the forest F_s is empty (its root set is empty). For Line 2, 3 and 4, the nodes parsed by *Comp* are the root node r_Δ of Δ for the Δ_{t-1} side and the root node r_d of d for the d_t side. Line 2 deals with the case where the parsed nodes r_Δ and r_d "match" an unchanged element node; then the function *TComp*(Δ, d, t) builds the tree whose root is r_Δ and whose sub-forest is recursively built by a call to *Comp* involving the sub-forest of Δ (for Δ_{t-1} side) and the sub-forest d (for d_t side). Line 3 deals with the case where both parsed nodes are the same text node. Line 4 captures the case where changes are identified: either r_Δ has been removed between the instants t–1 and t or it has moved modifying the child order; r_Δ is output first (together with its sub-forest) while closing its time-stamp, followed by the node r_d and its sub-forest time-stamped by $[t, Now]$ as expected.
It can be shown that:

Property 4.1: *It-Comp*$(\mathbf{d})$ is a concrete encoding for the abstract document $\mathbf{d}$, and *It-Comp*$(\mathbf{d})\leq\top(\mathbf{d})$.

Note that the way *Comp* is designed allows for a streaming

Comp(F_s, F, t) =		
$l.1$	$F^{[t,Now]}$	if $roots(F_s) = \varnothing$ otherwise
$l.1bis$	$F_s^{Now \leftarrow t}$	if $roots(F) = \varnothing$ otherwise assume F_s=$\Delta \circ \Phi$ and F=$d \circ f$
$l.2$	*TComp*$(\Delta, d, t) \circ$ *Comp*(Φ, f, t)	if *lab*(r_Δ)=*lab*(r_d) and $\gamma_\Delta(r_\Delta)$=$\gamma_d(r_d)$ and $\tau^+_\Delta(r_\Delta)$=Now
$l.3$	$\Delta \circ$ *Comp*(Φ, f, t)	if $\sigma_\Delta(r_\Delta)$=$\sigma_d(r_d)$=$text[st]$ and $\tau^+_\Delta(r_\Delta)$=Now
$l.4$	$\Delta^{Now \leftarrow t} \circ d^{[t,Now]} \circ$ *Comp*(Φ, f, t)	$otherwise$

Figure 3. Definition of *Comp*

implementation which ensures that large documents can be processed. Unfortunately, this also explains why *Comp* is unable to reduce replication of elements, which would require to buffer (in some case a large amount of) information. This is incompatible with our space-efficiency target.

B. Encoding Histories

This section considers a class of abstract temporal documents usually called histories. An abstract temporal document $\mathbf{d}$=$d_0, \ldots, d_n$ is an history when each document d_t is obtained from d_{t-1} by some update u_t. Indeed, the history $\mathbf{d}$ is equivalently specified by an initial XML document d_0 and a sequence of updates $u_1, \ldots, u_n$. We consider the XQuery Update Facility (XUF) update language as described in [3]. Although renaming is part of our study, we do not present the technical aspects of its treatment for the sake of simplicity. We also assume that each document d_i is valid wrt to some known DTD D_i, leaving space for schema evolution. For the sake of simplicity, we suppose next that $\forall i, D_i$=D.

Generating a time-stamp encoding of an history $\mathbf{d}$ specified by d_0 and $u_1, \ldots, u_n$ is an iterative process, called *It-Update*. The initial document d_0 is trivially transformed into the time-stamp document Δ_0=$d_0^{[0,Now[}$, and then assuming that Δ_{t-1} is the time-stamp document encoding the history specified by d_0 and $u_1, \ldots, u_{t-1}$, the update u_t is propagated, in a specific manner, over Δ_{t-1} to produce Δ_t=$Update(\Delta_{t-1}, u_t)$.

Recall here that our goal is to provide a compact encoding of the history $\mathbf{d}$, and also to handle time-stamp documents of very large size as well as we want to avoid for devising a new update engine. This is the reason why we have been investigating a method based on XML projection for specifying $Update(\Delta_{t-1}, u_t)$ because such method allows one for reducing main memory space consumption and for being compatible with any XUP query engine.

In [7], a type-based method has been proposed for optimizing main memory XML update processing where optimization should be understood as space optimization[1]. Given an update u over a document d valid wrt a DTD D, the idea is to determine, in a static manner, which fragments of the document are necessary for and touched by the update u. More precisely, from u and the DTD D, a static analysis infers a type-projector π specified by sets of labels. Then, the update scenario is as follows. At loading time, the document d is pruned wrt π in a streaming manner: roughly, nodes whose labels are in π are projected. The update u is evaluated over the pruned document $\pi(d)$. The document $u(\pi(d))$ is of course missing the pruned out nodes and thus a last step is necessary in order to reinstate the update of the projected document $u(\pi(d))$ in the document d. This is done by merging the documents d and $u(\pi(d))$ at writing-serializing time. The update scenario has been designed in order to avoid any rewriting of the update and in order to be independent of any main-memory engine.

The interested reader will find in [7] a full description of the type-projector, its extraction, the projection and merge algorithms as well as experiments validating the space and time optimization. Below, we proceed to a short introduction of the type-projector. Type-projector for updates have been devised in order to capture update expression features and also to ensure correctness of the merge phase. The type-projector π for an update u is specified by 3 sets of labels: $\pi_{\mathbf{no}}$ (the 'node only' component) is meant to capture labels of nodes that are traversed by u or target of deletion or existential condition; $\pi_{\mathbf{olb}}$ (the 'one level below' component) is meant to capture labels of mixed-content nodes or labels of nodes that are targets of insertion or replacement; $\pi_{\mathbf{eb}}$ (the 'everything below' component) is meant to capture labels of nodes that are roots of extracted subtrees. Given a document d valid wrt the DTD D, the behavior of the type-projector is as follows. If the label of a node (of d) belongs to $\pi_{\mathbf{no}}$, then it is projected. If the label of a node belongs to $\pi_{\mathbf{olb}}$, then it is projected together with all its children, even though their labels do not belong to the projector. If the label of a node belongs to $\pi_{\mathbf{eb}}$, then it is projected together with all its descendants, even though their labels do not belong to the projector. When a node is projected because its label belongs to π, its children are examined as candidate for projection, according to the above rules.

We are now ready to define $Update(\Delta_{t-1}, u_t)$. We use the example of Figure 4 to illustrate the presentation. The DTD D considered is specified by the rules *doc*→$(a|e)*$, *a*→*b*,c*,e?*, *c*→*String*$|f$, and *b,d*→$(f|g)*$. It is assumed that the document Δ_1 is an encoding of d_0, d_1 of Fig. 1 and that the update u_2 applied on d_1 to produce d_2 is specified by the XUF expression:

[1]Experiments of this method show that execution time is also improved for some engines.

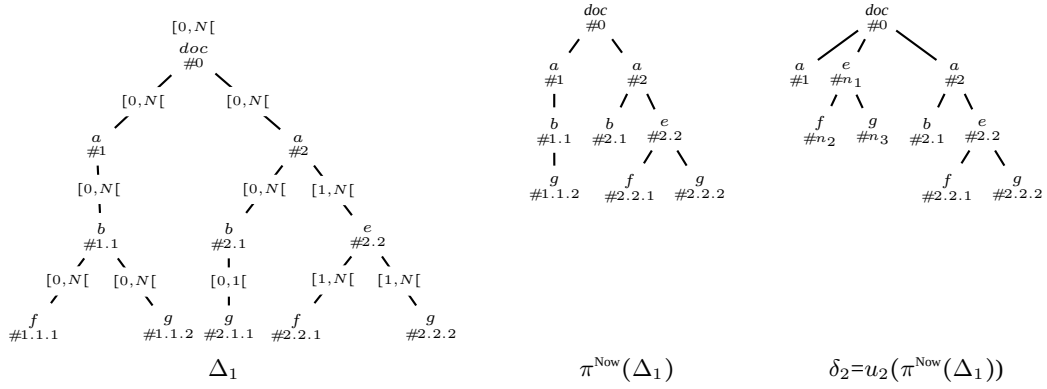


Figure 4. Illustrating the update-based encoding

for $\$x$ *in* $/doc/a$ *where* $\$x/b/g$
return { *insert node* $/doc/a/e$ *after* $\$x$; *delete* $\$x/b$}.

Propagating an update u_t on the time-stamp document Δ_{t-1} relies on the update scenario of [7] and proceeds as follows:

(1) the type-projector π for the update u_t is extracted as specified in [7]; for the update u_1 of our running example, the type-projector is specified by: π_{no}=$\{a, b, g\}$, π_{olb}=$\{doc\}$ and π_{eb}=$\{e\}$.

(2) the time-stamp document Δ_{t-1} is projected at loading time wrt the temporal type-projector π^{Now}, which combines the temporal projection over the time instant $t-1$ (or Now) and the type-projection π i.e. $\pi^{\text{Now}}(\Delta_{t-1}) \stackrel{def}{=} \pi(Snap(\Delta_{t-1}, Now))$; the projected document is equivalent to the type-projection of the document d_{t-1}=$u_{t-1}(u_0(d_0))$, i.e. we have[2]: $\pi^{\text{Now}}(\Delta_{t-1})$~$\pi(d_{t-1})$; for our example, only the node $\#2.1.1$ is pruned out by temporal projection wrt to Now; the type-projection prunes out the node $\#1.1.1$ because its label f does not belong to π; note that the subtree rooted at $\#2.2$ is projected because its label e belongs to π_{eb}.

(3) the update u_t is evaluated over the projected document $\pi^{\text{Now}}(\Delta_{t-1})$ producing $u_t(\pi^{\text{Now}}(\Delta_{t-1}))$; this can be performed by any update engine and update rewriting is not required; Fig. 4 shows the document $u_2(\pi^{\text{Now}}(\Delta_1))$ for our running example.

(4) the last step integrates the document $u_t(\pi^{\text{Now}}(\Delta_{t-1}))$ into the time-stamp document Δ_{t-1}; this phase called *UMerge* differs from [7] as not only it has to propagate the updates executed over $\pi^{\text{Now}}(\Delta_{t-1})$ but it also needs to maintain time-stamps. Wrt our example, the result of *UMerge* applied over Δ_1 and $u_2(\pi^{\text{Now}}(\Delta_1))$ is the time-stamp document[3] Δ' of Fig. 2(b).

The information used by *UMerge* are: the type-projector π, time-stamps and node identifiers as explained below. The *UMerge* phase proceeds by parsing both documents in a synchronized manner. At each step, the parsed node in Δ_{t-1} and the parsed node in $u_t(\pi^{\text{Now}}(\Delta_{t-1}))$ are examined to decide which one should be output and also to maintain time-stamps. We stress that *UMerge* does not perform changes on nodes other than time-stamp maintenance. Clearly, during parsing, the "old" nodes of Δ_{t-1} time-stamped by $[k_1, k_2[$ where $k_2 \neq Now$ are output directly: they have been pruned out by the temporal projection and were not involved in the update. For our example, this case applies to node #2.1.1. More attention should be paid to nodes of Δ_{t-1} time-stamped by $[k_1, Now[$ and thus corresponding to nodes in $Snap(\Delta_{t-1}, Now)$. Some of these nodes may have been pruned out although others may have been projected and modified. Detecting if a node n in $u_t(\pi^{\text{Now}}(\Delta_{t-1}))$ corresponds to a node m in Δ_{t-1} is based on node identifiers comparison with the following setting: (i) node identifiers for Δ_{t-1} are node positions; they are not stored within the document which would be quite space and time consuming but rather generated on the fly during the time-projection and the *UMerge* phases and only for the fragment of Δ_{t-1} corresponding to $Snap(\Delta_{t-1}, Now)$; however node positions are stored in the projection $\pi^{\text{Now}}(\Delta_{t-1})$; moreover, in practice, full position is not required and space is saved in a large extend by storing child order of nodes only; for the sake of the example, the position on the document Δ_1 have been given although, in practice, they are not part of the document; (ii) new nodes introduced by the update u_t have no explicit identifiers (positions) as we use the initial update u_t without rewriting it; for the sake of the example, special identifiers n_i have been used, in the document $u_2(\pi^{\text{Now}}(\Delta_1))$, for each node of the second subtree of the root because it is a subtree inserted by u_2.

Items (i) and (ii) entail that nodes n and m are identified to be the same (although the label of n may have been changed by the update u_t) just by verifying position equality.

Formally

$Update(\Delta_{t-1}, u_t)$=*UMerge(*$\Delta_{t-1}, u_t(\pi^{\text{Now}}(\Delta_{t-1})), t$*)*.

[2] ~ denotes value equivalence of XML documents

[3] The reader should not pay attention to the explicit identifiers which are different from those used in Fig 4.

Let us comment on *UMerge* with our running example. In order to do that, a node of a document X whose position is $\#i$ is denoted by $X\#i$; the document $u_2(\pi^{\text{Now}}(\Delta_1))$ is denoted with δ_2. While merging Δ_1 and δ_2, nothing special happens until after parsing $\Delta_1\#1$ and $\delta_2\#1$. Then, the next node to be parsed in Δ_1 is $\Delta_1\#1.1$, child of $\Delta_1\#1$, while $\delta_2\#1$ has no subtree. Because the time-stamp of $\Delta_1\#1$ is $[0, N[$ and its label b belongs to π, the function *UMerge* detects that the subtree rooted at $\Delta_1\#1.1$ has been deleted by u and thus this subtree is output after replacing the time-stamp $[0, N[$ by $[0, 2[$. The two next nodes examined are $\Delta_1\#2$ and the new node $\delta_2\#n_1$: the new subtree of δ_2 is then output with the appropriate time-stamp $[2, Now[$.

Property 4.2: Given an history $\mathbf{d}$, we have that:
(i) *It-Update*$(\mathbf{d})$ is a concrete encoding for $\mathbf{d}$, and,
(ii) *It-Update*$(\mathbf{d})\leq$*It-Comp*$(\mathbf{d})$.

Space limitation does not allow us to present proofs. Notice that (ii) expresses that when an abstract document is an history, the projection based method is better than the simple one in terms of compactness. This fact is validated by the experiments.

V. Experiments

To validate the effectiveness of our approach, we implemented both the *It-Comp* and *It-Update* algorithms in Java, and made experiments on a 2.53 Ghz Intel Core 2 Duo machine (2 GB main memory) running Mac OSX 10.6.4. The main-memory engine used for running the *It-Update* is QizX [10], [11].

We generated four abstract documents, starting from four initial XMark [19] documents of growing size, from 45MB to 1126MB. As the goal is to compare *It-Comp* and *It-Update*, each abstract document is an history d_0, d_1, d_2, d_3 where d_0 is the initial document, and d_i is obtained from d_{i-1} by means of an update u_i. The updates used for the experiments are given below. It should be clear that, for the purpose of evaluating *It-Comp*, we generated each d_i with adequate explicit identifiers, while evaluating *It-Update* only requires the initial document and the updates.

u_1. *delete node* /*site*/*regions*/*australia*

u_2. *for* \$*x in* /*site*/*closed_auctions*/*closed_auction*
where \$*x*/*annotation*
return insert node
<*amount*>*to be determined*</*amount*>
after \$*x*/*price*

u_3. *for* \$*x in* /*site*/*open_auctions*/*open_auction*
where \$*x*/*privacy return delete* \$*x*

Test results are reported in Table I. In this table, Δ_i is the result of compacting $d_0 \dots d_i$ documents applying the *It-Comp* method while Δ'_i is the encoding obtained after executing the u_i update starting from Δ'_{i-1} according the *It-Update* method. We compared the two methods in terms of sizes of Δ_i and Δ'_i with i ranging from 0 to 3. Obviously the difference in size between Δ_0 and Δ'_0 is explained by the presence of explicit identifiers in Δ_0.

d_0	45.4MB	112.4MB	454.8MB	1126.8MB
$\mathbf{\Delta_0}$	55.7	138.5	563.5	1351.7
$\mathbf{\Delta'_0}$	45.4	112.4	454.8	1126.8
$\mathbf{\Delta_1}$	76.5	190.5	774.7	1833.0
$\mathbf{\Delta'_1}$	45.4	112.4	454.8	1126.8
gain	40.7%	41.0%	41.3%	38.5%
$\mathbf{\Delta_2}$	84.5	210.1	853.6	2027.5
$\mathbf{\Delta'_2}$	45.6	112.9	456.4	1130.7
gain	46.0%	46.3%	46.5%	44.2%
$\mathbf{\Delta_3}$	91.7	228.6	929.0	2207.5
$\mathbf{\Delta'_3}$	45.6	112.9	456.4	1130.7
gain	50.3%	50.6%	50.9%	48.8%

Table I
COMPARISON BETWEEN *It-Update* AND *It-Comp*

Test results in Table I show the *It-Comp* method is more space consuming than the *It-Update* method. While the size is increasing in the Δ_i's sequence, the size is almost constant in the Δ'_i's sequence. This testifies that the *It-Update* method succeeds in avoiding node replication under updates. The method *It-Comp* instead entails a sensible amount of node replication since information coming from the update is not used.

We can also see that after each update the improvements in terms of size is sensible, going from 38.5%, after the first update, to 50.9%, after the last update.

In these experiments we considered abstract documents of length 4. Improvements in terms of size are likely to remain sensible in other realistic scenarios, since as the number of documents increases the probability that node duplication arises in the *It-Comp* method increases as well. Of course this holds for *It-Update* method as well, but in a much smaller extend, since update information is used. We postpone to future works experiments on more complex scenarios.

To conclude, we would like to highlight that both methods are able to process temporal documents of large size. Recall that the memory limitation of QizX, the update engine used for the experiments of *It-Update*, makes impossible to process documents whose size exceeds 580MB. We plan to run our experiments with other engines.

VI. Conclusion

In this paper we developed two techniques for generating and maintaining encodings of abstract temporal documents. The first technique addresses the case where no information is available on the abstract temporal document. Although this technique is not fully satisfactory from the point of view of space efficiency, it allows to maintain large temporal documents. The second one, called update-based, is designed

to manipulate document history. It takes advantage of a prior technique developed in [7] and enabling the update of large XML files using in-memory engines. The first experiments validate the effectiveness of both methods wrt to our first goal (processing large documents) and show that our update-based method is efficient wrt space-saving.

In the future, we plan to improve both methods and develop further experiments. Concerning the first one, we might investigate how to take advantage of a description of the changes (recall that changes are not updates) in order to develop projection-based method. For both method, we also plan to investigate parallelization in order to improve time execution.

VII. Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work has been partially funded by the Codex project, Agence Nationale de la Recherche, decision ANR-08- DEFIS-004.

References

[1] J. Chomicki and D. Toman, *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005, ch. Time in Database Systems.

[2] "Xquery update facility 1.0," http://www.w3.org/TR/xquery-update-10.

[3] M. Benedikt and J. Cheney, "Semantics, types and effects for xml updates," in *DBPL*, 2009, pp. 1–17.

[4] "SAX," http://www.saxproject.org/.

[5] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen, "Type-based xml projection," in *VLDB*, 2006, pp. 271–282.

[6] A. Marian and J. Siméon, "Projecting xml documents," in *VLDB*, 2003, pp. 213–224.

[7] M. A. Baazizi, N. Bidoit, D. Colazzo, N. Malla, and M. Sahakyan, "Projection for xml update optimization," in *EDBT*, 2011, pp. 307–318.

[8] "Galax," http://www.galaxquery.org.

[9] "Saxon-ee," http://www.saxonica.com/.

[10] "QizX/open," http://www.xmlmind.com/qizxopen.

[11] "QizX Free-Engine-3.0," http://www.xmlmind.com/qizx/free_engine.html.

[12] "eXist," http://exist.sourceforge.net/.

[13] D. Toman, "Point-based temporal extension of temporal sql," in *DOOD*, 1997, pp. 103–121.

[14] S. S. Chawathe, S. Abiteboul, and J. Widom, "Managing historical semistructured data," *TAPOS*, pp. 143–162, 1999.

[15] F. Rizzolo and A. A. Vaisman, "Temporal XML: modeling, indexing, and query processing," *VLDB Journal: Very Large Data Bases*, pp. 1179–1212, Aug. 2008.

[16] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan, "Archiving scientific data," *ACM Trans. Database Syst*, pp. 2–42, 2004.

[17] F. Wang and C. Zaniolo, "Temporal queries and version management in XML-based document archives," *Data Knowl. Eng*, pp. 304–324, 2008.

[18] F. Wang, C. Zaniolo, and X. Zhou, "Temporal xml? sql strikes back!" in *TIME*. IEEE Computer Society, 2005, pp. 47–55.

[19] A. S. 0002, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Xmark: A benchmark for xml data management," in *VLDB*, 2002, pp. 974–985.