Maintenance efficace de documents XML volumineux

Mohamed-Amine Baazizi, Nicole Bidoit, Dario Colazzo

Maintenance efficace de documents XML volumineux

Mohamed-Amine Baazizi [*1], Nicole Bidoit [*1], Dario Colazzo [*1]

[*] *Univ. Paris-Sud, Equipe Bases de Données & Leo, & INRIA Saclay*
[1]{first.last}@lri.fr

Résumé

La gestion des données temporelles est un problème critique pour diverses applications. Avec l'avènement de XML comme nouveau standard de représentation et d'échange des données, d'importants efforts ont été fournis pour développer des extensions temporelles de XML. Le but de ce travail est d'explorer comment générer ou maintenir des documents éstampillés garantissant un stockage efficace. Pour ce faire, nous définissons formellement la notion de compaction qui sert à comparer les documents éstampillés. Nous présentons ensuite deux méthodes pour construire les documents éstampillés. La première méthode est générale, elle ne fait aucune restriction sur l'évolution des documents XML. La deuxième méthode suppose que l'évolution des documents XML est le résultat de mises à jour. L'objectif des deux méthodes est le traitement de documents volumineux, l'utilisation des moteurs existants ainsi que la conformité avec le standard XQuery Update Facility. Nous comparons les deux méthodes en terme de taille des documents obtenus (compaction). La méthode basée sur les mises à jour produit des documents qui sont plus satisfaisants que ceux obtenus par la méthode générale. Ceci montre que la méthode basée sur les mises à jour que nous proposons exploite réellement celles-ci pour générer des documents plus compacts en même temps qu'elle permet de traiter des documents volumineux.

1 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. [13] 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 timestamped XML documents, a concrete model. Although many proposals have addressed the issue of querying timestamped 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 [13] 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 timestamped documents 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 [7, 9]. 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 [5] 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 [10, 14] and XUF [8] in order to overcome the main memory limitation of *in-memory* engines like Galax [2], Saxon [6], QizX [4, 3], and eXist [1]. The update-based method relies on pruning the timestamped document over which an update should be integrated in order to load only the fragment of the document touched or needed by the update. The pruning phase is very similar to that introduced in [8] and makes use of the schema (DTD) typing the document.

The benefit of extending the update mechanism of [8] is manifold. First, it enables to process large timestamped 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 [13] for a detailed survey). Chomicki et al. [13] 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 [16].

Recently, important efforts have focussed on studying temporal extensions for semi-structured data and XML in particular. For instance, Chawathe et al. [12] 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.

In many proposals, the last version of the document is stored together with *deltas* that encode the changes between the different versions of the document. 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. [11] 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. Timestamp approaches are more effective than delta-based approach wrt to querying simply because temporal queries can be directly evaluated on timestamped documents without the need to retrieve the different versions.

Buneman et al. [11] investigate data structures for managing 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 is considered as the most frequently applied update.

In our study, as opposed to [11], we do not focus on a particular application and 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 timestamped XML trees, [15] does not address maintenance of very large temporal XML documents.

Other proposals like [18, 17] address the issue of managing and querying historical XML databases. They present a technique for computing documents annotated with timestamps (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 models capturing temporal XML evolution are provided in Section 3 together with the compactness order. Section 4 introduces the two methods for generating timestamped documents from a sequence of static XML documents and compares the two methods wrt space-efficiency. Section 5 reports the results of a first bunch of experiments for validating the update based approach. We finally conclude and discuss future work in section 6.

2 Preliminaries

As in [9], 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: $k ::= \mathit{text}[s] \mid a[J]$, 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:
– $\mathit{dom}(\sigma){=}I$,
– $\mathit{lab}(\mathbf{i}){=}a$ if $\sigma(\mathbf{i}){=}a[I']$, $\mathit{lab}(\mathbf{i}){=}\mathit{String}$ if $\sigma(\mathbf{i}){=}\mathit{text}[s]$,
– $\mathit{child}(\sigma, K){=}\{\mathbf{i}' \mid \exists \mathbf{i}{\in}K, \sigma(\mathbf{i}){=}a[I']\ \mathit{and}\ \mathbf{i}'{\in}I'\}$,
– $\mathit{desc}(\sigma, K){=}\{\mathbf{i}' \mid \mathbf{i}'{\in}\mathit{child}(\sigma, K)$ or $\mathbf{i}'{\in}\mathit{desc}(\sigma, \mathit{child}(\sigma, K))\}$,
– $\mathit{roots}(\sigma){=}\{\mathbf{i} \mid \neg\exists \mathbf{i}', \mathbf{i}{\in}\mathit{child}(\sigma, \{\mathbf{i}'\})\}$, 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 $\mathit{roots}(\sigma){=}r$ and σ is a tree.

Let σ and σ' be two stores over I and I' resp. Let J and J' be two id-seqs

such that $J \subseteq I$ and $J' \subseteq I'$. The value equivalence $(J,\sigma) \sim (J',\sigma')$ is recursively defined by:
- $((),\sigma) \sim ((),\sigma')$ always holds,
- $(\mathbf{i}\cdot J,\sigma) \sim (\mathbf{i}'\cdot J',\sigma')$ iff $(J,\sigma) \sim (J',\sigma')$ and
  - $\sigma(\mathbf{i}) = a[K]$ implies $\sigma'(\mathbf{i}') = a[K']$ and $(K,\sigma) \sim (K',\sigma')$,
  - $\sigma(\mathbf{i}) = text[s]$ implies $\sigma'(\mathbf{i}') = text[s]$.

Value equivalence can be extended to a pair of forests f and f'. We write $f \sim f'$ for $(roots(f),\sigma_f) \sim (roots(f'),\sigma_{f'})$.

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 timestamps 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.

3 Temporal XML Models

The first part of this section follows the lines of [13] 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* $d_t = (r,\sigma_t,\gamma_t)$*, it is assumed that the mapping* γ_t *satisfies:* $\forall \mathbf{i},\mathbf{i}' \in dom(\sigma_t), \mathbf{i} \neq \mathbf{i}' \Rightarrow \gamma_t(\mathbf{i}) \neq \gamma_t(\mathbf{i}')$.

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 timestamps over document elements, providing their validity period. Changes are then encoded within a single document.

Definition 3.2 (Timestamped Document)
A timestamped XML document $\Delta = (r,\sigma,\gamma,\tau)$ *is an XML document enriched with a mapping* τ*:* $dom(\sigma) \to Int$*, satisfying the following properties*[1]*:*
- $\forall \mathbf{i},\mathbf{j} \in dom(\sigma) \setminus \{r\}$, $\mathbf{j} \in child(\sigma,\{\mathbf{i}\})$ *implies* $\tau(\mathbf{j}) \subseteq \tau(\mathbf{i})$*, and*
- $\forall \mathbf{i},\mathbf{j} \in dom(\sigma)$, $\mathbf{i} \neq \mathbf{j}$ *and* $\gamma(\mathbf{i}) = \gamma(\mathbf{j})$ *implies* $\tau(\mathbf{i}) \cap \tau(\mathbf{j}) = \emptyset$.

[1] Int is the set of positive integers.

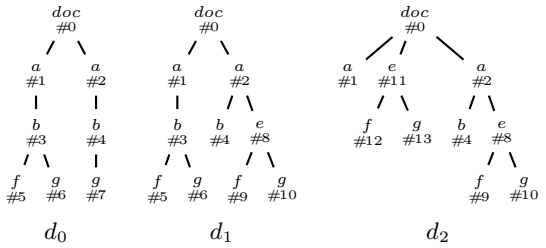


Figure 1: A temporal abstract document **d**

The first condition is classical and ensures that timestamps 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 timestamped document, γ is no more a bijection. Fig. 2 depicts two timestamped documents Δ and Δ'. For the purpose of making easier the manipulation of a timestamped document, the explicit value of *Now* is registered at its root.

Temporal projection is now defined by extending XML projection already introduced in Section 2 for (static) XML documents.

Definition 3.3 (Temporal Projection)
The temporal projection of a timestamped document $\Delta{=}(r, \sigma, \gamma, \tau)$ *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 timestamped document* $\Delta{=}(r, \sigma, \gamma, \tau)$ *is a sound, resp. complete, encoding of* **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 timestamped documents Δ and Δ' of Figure 2 are encodings of the abstract document of Figure 1.

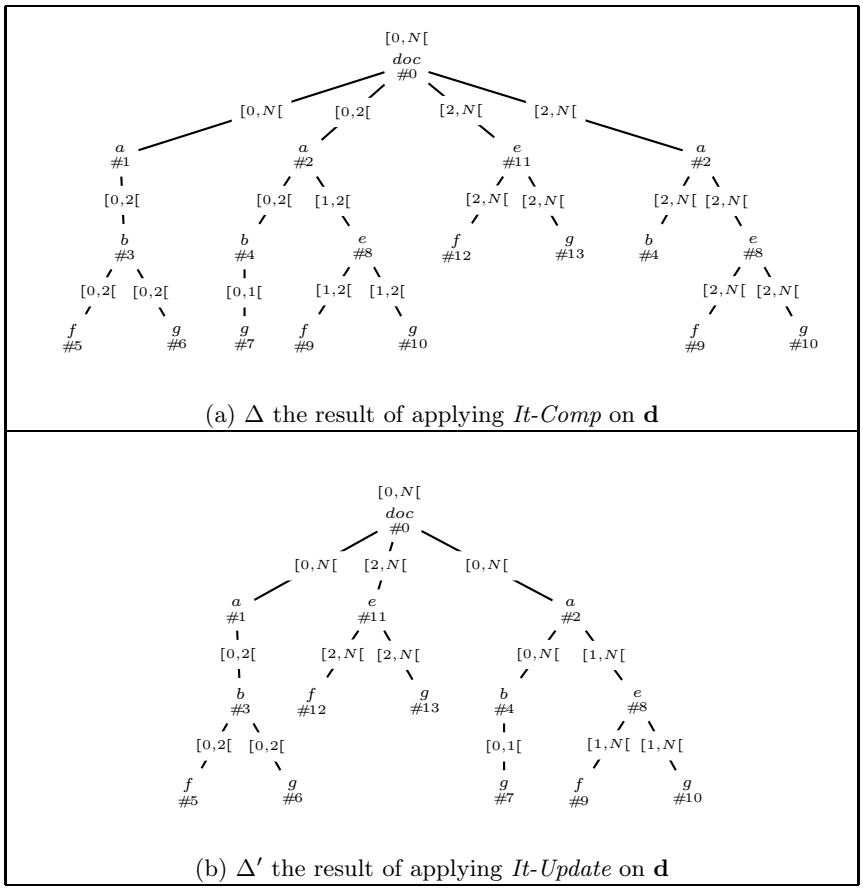
In the reminder, we consider sound and complete concrete encodings of abstract temporal documents and the term "concrete encoding" implicitly includes "sound and complete".

The following table summarizes the notation used throughout the article.

static XML doc.	d	DTD	D
abstract temporal XML doc.	**d**	timestamped 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.



(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 $\mathbf{d}$

Next, we formalize such a notion by introducing a pre-order $\preceq$ that enables to compare timestamped documents.

Given a timestamped 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: $S_f(l,x){=}\{\mathbf{i}{\in}I \mid \sigma(\mathbf{i}){=}l[K] \text{ and } \gamma(\mathbf{i}){=}x\}$.

Definition 3.5 (Compactness Order)
Consider two timestamped 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 *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:* $\Pi_{K_1}(f_1){\preceq}\Pi_{K_2}(f_2)$ *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 since the subtree rooted at the node whose label is a and explicit identifier is 2 is replicated in Δ.

It may happen that two timestamp documents are not comparable wrt compactness. For instance, it is the case of Δ_1 and Δ'_1 in Fig. 3. In the general case, $\preceq$ is a pre-order: it is reflexive, transitive but is not antisymmetric. Proving reflexivity and transitivity is straightforward. In order to prove that $\preceq$ is not antisymmetric, we exhibit a counter example. Let us consider the timestamped documents in Figure 3: we have that $\Delta'_1 \preceq \Delta_2$ and $\Delta_2 \preceq \Delta'_1$ but $\Delta'_1 \neq \Delta_2$.

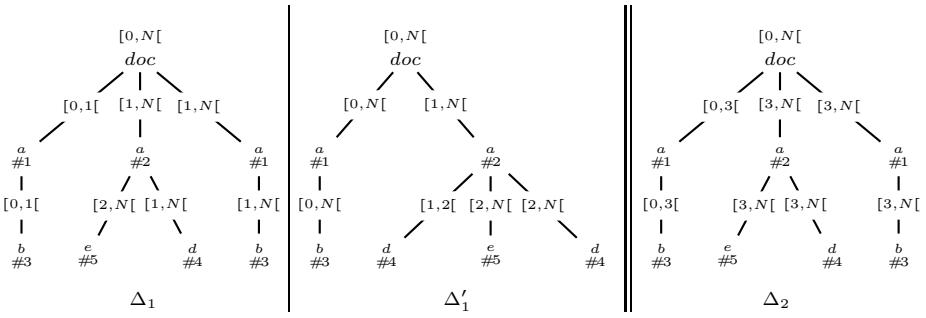


Figure 3: Compactness order

Note that Δ'_1 and Δ_2 do not encode the same abstract temporal document. Indeed, comparing the compactness of two timestamped documents that differ by their labels and temporal domains does not convey much information. Next, we will be comparing pairs of timestamped documents that are concrete encoding of the same abstract temporal XML document. Under such a restriction[2], we have:

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

Sketch of proof. Reflexivity and transitivity can be proved in a straightforward manner by using Definition 3.5. In order to prove antisymmetry, we need to prove that given two timestamped documents Δ_1 and Δ_2 encoding the same abstract document $\mathbf{d}$, if $\Delta_1 \preceq \Delta_2$ and $\Delta_2 \preceq \Delta_1$ then necessarily Δ_1 and Δ_2 are value-equivalent.[3] This is done by induction on Δ_1 and Δ_2 using the definitions 3.5 and 3.4.

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

[2]For sake of simplicity, we do not formally introduce this restriction in the property below.

[3]Value-equivalence for timestamped documents is defined in a straightforward manner by extending value-equivalence for static documents (see Section2).

4 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 very large documents. 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 timestamped document in a significantly more compact manner.

4.1 Encoding general abstract documents

$Comp(F_s, F, t)$ =		
$l.1$	$F^{[t,Now]}$	if $roots(F_s) = \varnothing$
$l'.1$	$F_s^{Now \leftarrow t}$	if $roots(F) = \varnothing$
		otherwise assume F_s=$\Delta \circ \Phi$ and F=$d \circ f$
$l.2$	$\Delta \circ Comp(\Phi, f, t)$	if $\tau^+_\Delta(r_\Delta) \neq Now$
		otherwise assume $\tau^+_\Delta(r_\Delta)$=Now
$l.3$	$TComp(\Delta, d, t) \circ Comp(\Phi, f, t)$	if $lab(r_\Delta)$=$lab(r_d)$ and $\gamma_\Delta(r_\Delta)$=$\gamma_d(r_d)$
$l.4$	$\Delta \circ Comp(\Phi, f, t)$	if $\sigma_\Delta(r_\Delta)$=$\sigma_d(r_d)$=$text[st]$
$l.5$	$\Delta^{Now \leftarrow t} \circ d^{[t,Now]} \circ Comp(\Phi, f, t)$	$otherwise$

Figure 4: Definition of *Comp*

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

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 4. Its inputs are: a timestamped forest F_s (a sub-forest of Δ_{t-1}), a XML forest F (a sub-forest of the static XML document d_t), and t referring to the validity time of d_t.
Given an interval int, the timestamped forest f^{int} is obtained by time stamping each node of the forest f with int. The timestamped forest $\Phi^{t_1 \leftarrow t_2}$ is obtained by upper bound substitution: each occurrence of the timestamp $[t, t_1[$ in Φ is replaced by $[t, t_2[$.

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, 4 and 5, 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 node r_Δ is not valid at the current time. In this case, the subtree rooted at r_Δ is simply output. Line 3 deals with the case where the parsed nodes r_Δ and r_d "match" an unchanged element node; then the function $\mathit{TComp}(\Delta, d, t)$ 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 4 deals with the case where both parsed nodes are the same text node. Line 5 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 timestamp, followed by the node r_d and its sub-forest timestamped 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*$(\mathbf{d}) \preceq \mathtt{Top}(\mathbf{d})$.

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

4.2 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 [9]. 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 timestamped 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 timestamped document $\Delta_0{=}d_0^{[0,Now[}$, and then assuming that Δ_{t-1} is the timestamped 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 $\Delta_t{=}\mathit{Update}(\Delta_{t-1}, u_t)$.

Recall here that our goal is to provide a compact encoding of the history $\mathbf{d}$, and also to handle timestamped documents of very large size as well as we want to avoid for devising a new update engine. This motivates the investigation of a method based on XML projection for specifying $\mathit{Update}(\Delta_{t-1}, u_t)$ because such method allows one for reducing main memory space consumption and for being compatible with any XUF [7] query engine.

In [8], a type-based method has been proposed for optimizing main memory XML update processing where optimization should be understood as space optimization[4]. 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 [8] 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-projectors 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 5 to illustrate the presentation. The DTD D considered is depicted in Figure 5.

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:

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 timestamped document Δ_{t-1} relies on the

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

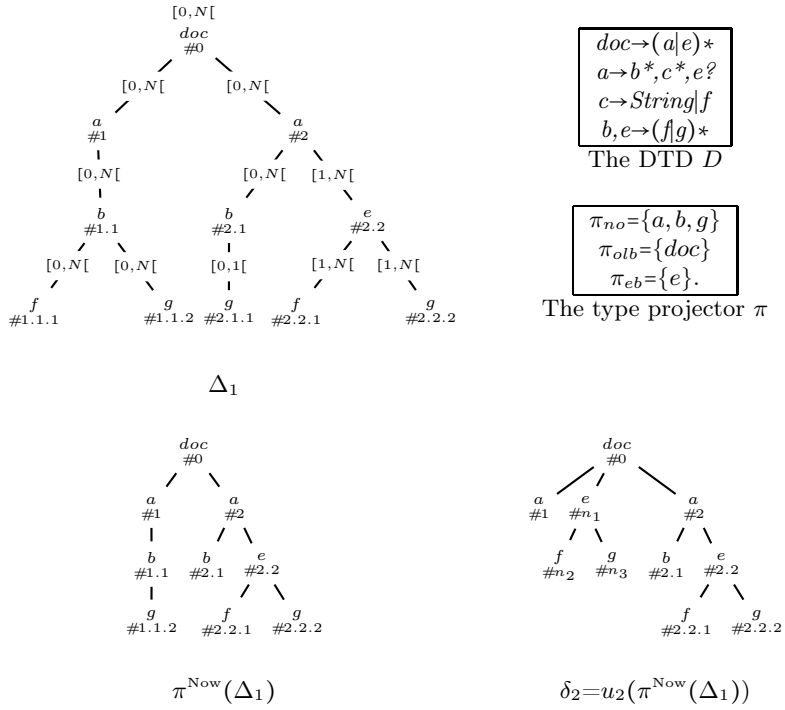


Figure 5: Illustrating the update-based encoding

update scenario of [8] and proceeds as follows:
(1) the type-projector π for the update u_t is extracted as specified in [8]; for the update u_2 of our running example, the type-projector is the one of Figure 5.
(2) the timestamped 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 $\pi^{\text{Now}}(\Delta_{t-1})$~$\pi(d_{t-1})$; 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. 5 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 timestamped document Δ_{t-1}; this phase called $UMerge$ differs from [8] as not only it has

to propagate the updates executed over $\pi^{\text{Now}}(\Delta_{t-1})$ but it also needs to maintain timestamps. Wrt our example, the result of $UMerge$ applied over Δ_1 and $u_2(\pi^{\text{Now}}(\Delta_1))$ is (value equivalent to) the timestamped document Δ' of Fig. 2(b).

The information used by $UMerge$ are: the type-projector π, timestamps 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 timestamps. We stress that $UMerge$ does not perform changes on nodes other than timestamp maintenance. Clearly, during parsing, the "old" nodes of Δ_{t-1} timestamped 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. Nodes in Δ_{t-1} timestamped by $[k_1, Now[$, corresponding to nodes in $Snap(\Delta_{t-1}, Now)$, requires a careful analysis: some of these nodes may have been pruned out although others may have been projected and modified. Checking whether a node n in $u_t(\pi^{\text{Now}}(\Delta_{t-1}))$ corresponds to a node m in Δ_{t-1} (although the label of n may have been changed by the update u_t) is done by checking identifier equality based on the the following setting: (i) node identifiers for Δ_{t-1} are node positions (see additional remarks below); (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.

Remark on node position. The reader should pay attention to the fact that, for the sake of the example, node positions for the document Δ_1 are given although, in practice, they are not stored within the timestamed document. This would uselessly increase space and time consumption. In practice, node positions are generated on the fly during the time-projection and the $UMerge$ phases. As positions are used only for nodes in $Snap(\Delta_{t-1}, Now)$, they are generated only for these nodes rather than for the whole timestamped document Δ_{t-1}. Node positions are stored, in memory, for the projection $\pi^{\text{Now}}(\Delta_{t-1})$. Moreover, in practice, full positions are not required and child order is sufficient with the advantage of reducing significantly space requirements.
For the sake of the example presentation, special identifiers n_i have been used, in the document $u_2(\pi^{\text{Now}}(\Delta_1))$, for nodes inserted by u_2.

Formally:

$$Update(\Delta_{t-1}, u_t) = UMerge_{\mathbf{no}}(\Delta_{t-1}, u_t(\pi^{\text{Now}}(\Delta_{t-1})), t).$$

The definition of the function $UMerge_{\mathbf{no}}$ is given in Figure 6(a). The functions $UMerge_{\mathbf{olb}}$ and $UMerge_{\mathbf{eb}}$ are specified through $UMerge_{\alpha}$ in Figure 6(b) where α refers either to **olb** or to **eb**. The inputs of these three functions are: a sub-forest F_s of Δ_{t-1}, a sub-forest F_u of $u_t(\pi^{\text{Now}}(\Delta_{t-1}))$ and t referring to the validity time of u_t.

The three functions are distinguished based on the following pre-conditions:
– $UMerge_{\mathbf{no}}$ assumes that (†) the parent node n of the forest F_s is of cat-

egory 'node only' which implies that, because of synchronization, i) none of the top level trees in F_u is of type *String*, and ii) root identifiers of top-level trees in F_u belong to root identifiers of top-level trees in F_s restricted at the time point $t-1$ that is: $roots(F_u) \subseteq roots^{Now}(F_s)$ where $roots^{Now}(F_s) = \{\mathbf{i} \mid \mathbf{i} \in roots(F_s) \text{ and } \tau^{+}(\mathbf{i}) = Now\}$.

– $UMerge_{\mathbf{olb}}$ assumes that (‡) the parent node n of F_s is in category 'one level below' which implies that, each node in $roots^{Now}(F_s)$ has been projected.

– $UMerge_{\mathbf{eb}}$ assumes that (‡‡) the parent node n of F_s is in category 'everything below' which implies that, each subtree whose root node is in $roots^{Now}(F_s)$ has been projected.

The function $UMerge_{\mathbf{no}}$ proceeds as follows:

Line 2 takes care of the case where the parsed node of Δ is either: (i) a node that has been pruned out by temporal projection or (ii) a *String* node not projected because of assumption (†). In both cases, the timestamped document Δ has to be output.

Line 3 deals with the case where Δ has been deleted by the update. This fact is identified by the fact that the label a of the root r_Δ of Δ belongs to π (thus at least r_Δ has been projected) and r_Δ does not occur in F_u: either F_u is empty or by comparing the identifiers of the currently parsed nodes (which are positions in F_s) we find that $r_{d_u} > r_\Delta$. Therefore, r_Δ is together with its sub-forest are output while closing their timestamps.

Line 4 takes care of the case where the parsing is synchronized over the "same" subtrees: the positions of r_Δ and of r_{d_u} are equal and only their labels may differ because of some renaming update. In this case, the output is $TreeUMerge_{\mathbf{no}}(\Delta, d_u, t)$ which is the subtree whose root is labelled with $lab(r_\Delta)$ and whose sub-forest F is defined as follows:

$$F = \begin{array}{ll} UMerge_{\mathbf{no}}(subfor(\Delta), subfor(d_u), t) & \text{if } lab(r_\Delta) \in \pi_{\mathbf{no}} \\ UMerge_{\mathbf{olb}}(subfor(\Delta), subfor(d_u), t) & \text{if } lab(r_\Delta) \in \pi_{\mathbf{olb}} \\ UMerge_{\mathbf{eb}}(subfor(\Delta), subfor(d_u), t) & \text{if } lab(r_\Delta) \in \pi_{\mathbf{eb}} \end{array}$$

where $subfor(d)$ is used for denoting the sub-forest of d.

We would like to stress that, for the case where $lab(r_\Delta) \in \pi_{\mathbf{eb}}$, unlike the scenario of static documents developed in [8], $UMerge$ performs additional processing over $subfor(\Delta)$ and $subfor(d_u)$ in order to propagate the effect of the update in a temporal fashion. Indeed, in [8] the result of the $Merge$ phase for the case where $lab(r_\Delta) \in \pi_{\mathbf{eb}}$ is simply the sub-forest of the updated partial document $subfor(d_u)$. In our case, the output is built by $UMerge_{\mathbf{eb}}$ whose explanation is given below.

Finally, line 5 deals with the case where the label of r_Δ does not belong to the projector π entailing that Δ has not been projected. Thus, Δ is output.

The function $UMerge_\alpha$ in Fig. 6(b) is a shorthand for the functions $UMerge_{\mathbf{olb}}$ and $UMerge_{\mathbf{eb}}$. Recall that $UMerge_{\mathbf{olb}}$ and $UMerge_{\mathbf{eb}}$ are distinguished based of the pre-conditions (‡) and (‡‡) respectively.

Line b2 deals with the case where the parsed node of the timestamped document Δ is an "old" node that has to be output together with its sub-tree.

Line b3 and b4 deal with the case where the current parsed tree of Δ is a *String* s which has been projected by temporal projection (‡ and ‡‡ entail that *String* nodes are projected). Line b3 deals with the case where the *String* s has not been modified by the update and has, thus, to be output. Line b4 deals with two cases: the case where the *String* s has been either modified by the update, indicated by $\sigma_{d_u}(r_{d_u})$=$text[s']$, or deleted. In both cases, the *String* s is output after closing its timestamp. One should pay attention to the fact that the subtree d_u will be considered by the application of the forthcoming lines.

Line b5 deals with the case where the current parsed tree d_u of F_u is either of type *String* or a newly inserted element. This latter case is identified by checking that the identifier of r_{d_u} is new ($\notin dom(\sigma_\Delta)$). Thus, the tree d_u is output with the timestamp $[t, Now[$.

Line b6 is similar to line 3 of $UMerge_{\mathbf{no}}$. It does not require further comments.

Line b7 is similar to line 4. The output is $TreeUMerge_\alpha(\Delta, d_u, t)$ where
– $TreeUMerge_{\mathbf{olb}}(\Delta, d_u, t)$=$TreeUMerge_{\mathbf{no}}(\Delta, d_u, t)$ and
– $TreeUMerge_{\mathbf{eb}}(\Delta, d_u, t)$ is the subtree whose root is labelled with $lab(r_\Delta)$ and whose sub-forest F=$UMerge_{\mathbf{eb}}(subfor(\Delta), subfor(d_u), t)$.
In the latter case, $TreeUMerge_{\mathbf{eb}}$ enforces the parsing of $subfor(\Delta)$ and $subfor(d_u)$ to be performed by $UMerge_{\mathbf{eb}}$. Recall that, because of the assumption (‡‡), the tree Δ has been projected entirely. $UMerge_{\mathbf{eb}}$ is designed such that the updates performed over the projection d_u are propagated into Δ while maintaining its timestamps.

Line b8 is deals with the case where the root node n of Δ does not belong to the projector π. In this case, although implicit, the equality r_Δ=r_{d_u} holds since, because of (‡‡) and thanks to synchronization, the node identified by r_Δ=r_{d_u} is in both forests F_s and F_u.

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 timestamp 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 timestamp $[0, N[$ 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 timestamp $[2, Now[$.

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*($\mathbf{d}$)≤*It-Comp*($\mathbf{d}$)*, and thus*
(iii) *It-Update*($\mathbf{d}$)≤$\mathtt{Top}(t)$

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.

$UMerge_{\mathbf{no}}(F_s, F_u, t)$ =		
1	$()$	if $roots(F_s)$=$\varnothing$ **otherwise** assume F_s=$\Delta \circ \Phi$
2	$\Delta \circ\ UMerge_{\mathbf{no}}(\Phi, F_u, t)$	if either $\sigma_\Delta(r_\Delta)$=$text[s]$ or $\tau^+_\Delta(r_\Delta) \neq Now$, **otherwise** assume $\sigma_\Delta(r_\Delta)$=$a[I_\Delta]$ and $\tau^+_\Delta(r_\Delta)$=Now
3	$\Delta^{Now \leftarrow t} \circ\ UMerge_{\mathbf{no}}(\Phi, F_u, t)$	if $a \in \pi_\cup$ and either $roots(F_u)$=$\varnothing$ or F_u=$d_u \circ f_u$ with $r_{d_u} > r_\Delta$
4	$TreeUMerge_{\mathbf{no}}(\Delta, d_u, t) \circ UMerge_{\mathbf{no}}(\Phi, f_u, t)$	if $a \in \pi$, F_u=$d_u \circ f_u$ and r_Δ=r_{d_u}
5	$\Delta \circ\ UMerge_{\mathbf{no}}(\Phi, F_u, t)$	if $a \notin \pi$

(a) The definition of $UMerge_{\mathbf{no}}$

$UMerge_\alpha(F_s, F_u, t)$ =		
$b1$	$F_u^{[t,Now[}$	if $roots(F_s)$=$\varnothing$ **otherwise** assume F_s=$\Delta \circ \Phi, F_u$=$d_u \circ f_u$
$b2$	$\Delta \circ UMerge_\alpha(\Phi, F_u, t)$	if $\tau^+_\Delta(r_\Delta) \neq Now$ **otherwise** assume $\sigma_\Delta(r_\Delta)$=$text[s]$, $\tau^+_\Delta(r_\Delta)$=Now
$b3$	$\Delta \circ UMerge_\alpha(\Phi, f_u, t)$	if $\sigma_{d_u}(r_{d_u})$=$text[s]$
$b4$	$\Delta^{Now \leftarrow t} \circ\ UMerge_\alpha(\Phi, F_u, t)$	if either $\sigma_{d_u}(r_{d_u})$=$text[s'], s \neq s'$ or $\sigma_{d_u}(r_{d_u})$=$b[I_{d_u}]$ **otherwise** assume $\sigma_\Delta(r_\Delta)$=$a[I_\Delta]$,$\tau^+_\Delta(r_\Delta)$=Now
$b5$	$d_u^{[t,Now[} \circ UMerge_\alpha(F_s, f_u, t)$	if either $\sigma_{d_u}(r_{d_u})$=$text[s]$ or $\sigma_{d_u}(r_{d_u})$=$b[I_{d_u}], new(r_{d_u})$=$true$
$b6$	$\Delta^{Now \leftarrow t} \circ\ UMerge_\alpha(\Phi, F_u, t)$	if $a \in \pi$ and either $roots(F_u)$=$\varnothing$ or $r_{d_u} > r_\Delta$
$b7$	$TreeUMerge_\alpha(\Delta, d_u, t) \circ UMerge_\alpha(\Phi, f_u, t)$	if $a \in \pi$, r_Δ=r_{d_u}
$b8$	$\Delta \circ\ UMerge_\alpha(\Phi, f_u, t)$	if $a \notin \pi$

(b) The definition of $UMerge_{\mathbf{olb}}$ and $UMerge_{\mathbf{eb}}$

Figure 6: The specification of $UMerge$

5 Experiments

To validate the effecticiency 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 [4, 3].

We generated four abstract documents, starting from four initial XMark 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*

d_0	**45.4MB**	**112.4MB**	**454.8MB**	**1126.8MB**
Δ_0	55.7	138.5	563.5	1351.7
Δ'_0	45.4	112.4	454.8	1126.8
Δ_1	76.5	190.5	774.7	1833.0
Δ'_1	45.4	112.4	454.8	1126.8
gain	40.7%	41.0%	41.3%	38.5%
Δ_2	84.5	210.1	853.6	2027.5
Δ'_2	45.6	112.9	456.4	1130.7
gain	46.0%	46.3%	46.5%	44.2%
Δ_3	91.7	228.6	929.0	2207.5
Δ'_3	45.6	112.9	456.4	1130.7
gain	50.3%	50.6%	50.9%	48.8%

Table 1: Comparison between *It-Update* and *It-Comp*

Test results are reported in Table 1. 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.

Test results in Table 1 show that 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 replication 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.

6 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 [8] 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 execution time.

References

[1] eXist. `http://exist.sourceforge.net/`.

[2] Galax. `http://www.galaxquery.org`.

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

[4] QizX/open.
http://www.xmlmind.com/qizx/qizxopen.shtml.

[5] SAX. http://www.saxproject.org/.

[6] Saxon-ee. http://www.saxonica.com/.

[7] Xquery update facility 1.0.
http://www.w3.org/TR/2008/CR-xquery-update-10-20080801.

[8] Mohamed-Amine Baazizi, Nicole Bidoit, Dario Colazzo, Noor Malla, and Marina Sahakyan. Projection for XML update optimization. In *EDBT'11*, 2011.

[9] Michael Benedikt and James Cheney. Semantics, types and effects for XML updates. In *DBPL*. Springer, 2009.

[10] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen. Type-based XML projection. In *VLDB*, 2006.

[11] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. *ACM Transactions on Database Systems*, 2004.

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

[13] Jan Chomicki and David Toman. *Time in Database Systems*. Elsevier, 2005.

[14] A. Marian and J. Siméon. Projecting XML documents. In *VLDB '03*, pages 213–224, 2003.

[15] Flavio Rizzolo and Alejandro A. Vaisman. Temporal XML: modeling, indexing, and query processing. *VLDB Journal: Very Large Data Bases*, 17(5):1179–1212, August 2008.

[16] David Toman. Point-based temporal extensions of SQL and their efficient implementation. In *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[17] Fusheng Wang and Carlo Zaniolo. Temporal queries and version management in XML-based document archives. *Data Knowl. Eng*, 2008.

[18] Fusheng Wang, Carlo Zaniolo, and Xin Zhou. Temporal XML? SQL strikes back! IEEE Computer Society, 2005.