Managing Branch Versioning in Versioned/Temporal XML Documents*

Luis J. Arévalo Rosado, Antonio Polo Márquez, and Jorge Martínez Gil

University of Extremadura, Department of Computer Science
Avda. de la Universidad s/n 10071 Caceres (Spain)
{ljarevalo,polo,jmargil}@unex.es

Abstract. Due to the linear nature of time, XML timestamped solutions for the management of XML versions have difficulty in supporting non-lineal versioning. Following up on our previous work, which dealt with a new technique for the management of non-lineal versions of XML graph documents, called versionstamp, we have gone a step forward by adding temporal information to each version included in the document. Not only does it allow us to query the vDocuments on a temporal and version level but also we can manage branch versioning in the temporal axis. Moreover, to check its functionality, we have compared our technique to a timestamped XML solution and a set of Web services has been developed. The easy management of multiple versioning, the large number of queries in different XML standard query languages and its implementation by using only XML technology, are some of the advantages of the proposed technique.

1 Introduction

In this collaborative society information flows through all forms of computing, however nobody looks at it in a static way because it changes throughout time and its management becomes necessary to query past information, to retrieve documents belonging to a specific version and to monitor the changes, etc. Document management has been used for years in such environments like collaborative software development, file share resources, etc and more recently, with the appearance of XML [1], it has become necessary also to manage these documents.

Versions of an XML document can be managed through traditional procedures like CVS [2] or subversion [3], the traditional adapted procedures based on XML operations change (delta XML) [4,5] or integrate the different versions into a single XML file using temporal [8,11,12,13,14] or version [9,15] technique. We consider that whatever XML versioning system should have the following main features: it should be able to, validate all XML versions of the document to its schema (the first two solutions do not take into account this fact), support branch versioning (temporal solutions do not do this) and, have the possibility to

* This work has been financed by Spanish CICYT projects "TIN2005-09098-C05-05" and "TIN2005-25882-E".

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 107–121, 2007.

query the XML versioned documents using some XML standard query languages such as XQuery and XPath (the first solution does not do this).

To get these characteristics, we have used the technique shown in [9] that consists of marking the document with a versionstamp instead of using a timestamp. In this work we have gone even further by adding temporal information to each version allowing us to query the document either on temporal or/and version level. We have also defined the basic updated operations common to whatever XML document, describing them by means of an XML document called *XML transactional document* which allows us to manage changes for any markup language based on the XML specification. Moreover, to check its functionality, we have compared our technique to a timestamped XML solution as well as developing a set of Web Services.

The remainder of this paper is organized as follows: we begin by summarizing the current solutions for the management of XML versions. Then, we continue showing the foundations of this paper based on [9], extending it with temporal information and describing later the basic updated operations. We then follow up this by showing several queries made on a temporal and version level. After that, some implementation details and the achieved results are discussed and finally, we offer our conclusions and a look at our future work.

2 State-of-the-Art

The problem of XML document version management combines the issues of document version management [4,5,6,7] and temporal databases [22]. Document version management has been used for years mainly in collaborative environments. These traditional techniques [2,3] are based on diff lined-based algorithms to locate the differences between two versions of a text. For XML documents, where the organization in lines can be neglected, line-based approaches are inappropriate since the structure of the document is lost. The necessity to manage XML versions not only is important in XML databases but also in XML document management because nowadays more and more applications use it to store their configurations, data, etc, such as OpenOffice and Microsoft Office.

XML solutions have been centered mainly in some of the following ideas. *Delta XML management* is based on traditional change operation procedures adapted to XML [4,5]. It consists of obtaining and storing the XML differences between two versions (*delta XML*). An exhaustive study of XML diff approaches is made in [10] where the authors use an C++ implementation of [4] to manage XML OpenOffice document versions. However delta XML solutions have the same problems than traditional techniques, it means, neither XML validation nor XML query cannot be carried out in these solutions.

Multiversion XML [6,7] define an indexing technique for branched versioning which they called BT-Tree and BT-ElementList respectively, however they cannot be used in XML Standard Query languages (XQuery or XPath[2]).

Temporal XML Representation based on temporal database topics [21] representing and managing historical information in XML. In [11] a technique for

managing temporal web documents is shown using an XML/XSLT infrastructure. A data model is proposed for temporal XML documents [14] where leaf data nodes can have alternative values; however supporting different structures for non-leaf nodes is not discussed. Extensions of XPath data model are exposed in [13,12] to represent and query transactional and valid time respectively, by means of the addition of several temporal dimensions. A temporally-grouped data model is shown in [8] that gives us a way to represent the content database evolution using XML timestamps, however non-lineal versioning is not supported.

The integration of time and version concepts to manage dynamic information has been studied recently in [15,16] for XML and object-oriented databases respectively. In [15] the authors defined temporal delta (tDelta) and introduces version time in it, however query support is not discussed.

Due to the linear nature of time, XML timestamped solutions for the management of XML versions have difficulty in supporting non-lineal versioning. In collaborative scenario, due to the fact that users can update any version of the document generating a new version either from the current one or discard it and reuse an old version, branched versioning is necessary. Using our solution, called as *versionstamp* or *vstamp* [9], this feature can be modeled in a easy way.

3 XML Versioned Documents

In this section we present how to manage changes in XML document in a branch way. Firstly the foundations our work is based on [9] is shown. Then we extend it to incorporate temporal information and finally we describe a taxonomy of changes for XML documents.

3.1 Versionstamp Technique

An XML versioned graph data model, called as *V-XML* data model, was shown in [9] to represent versions in XML graph documents by means of adding versionstamp information in the graph document obtaining a new XML document which we called as *vXML Document* or *vDocument*. This is formed by two sections: The first one which stores all information about the included versions and the relationship between them and the second one being, each element in the document is transformed into a versioned element by means of defining its version validity, that is, for which version/s of the document it is valid.

In order to store the included versions, we decided to map by means of an XML document, which we called as *version_ tree,* how the versions have been made over time. Each included version is an element and represents the different snapshots of the document. If there is an parent-child relationship from V_i element to V_j element, it means that, V_j is created by updating V_i.

Once the included versions have been represented, it is necessary for each versioned element to represent its version validity. To do it, we use a *versionstamp technique*, which we called as *Version Region* [9], that is defined as a set of version identifiers from the version tree indicating for which versions of the tree it is valid

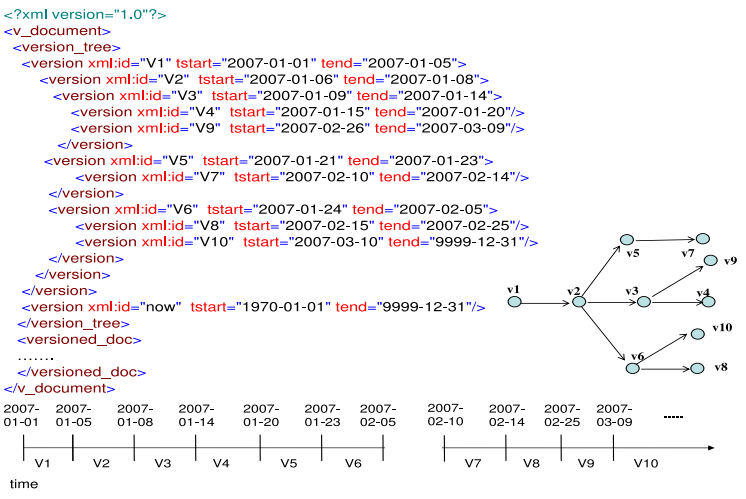

Fig. 1. XML and graphical representation of a version tree with temporal information

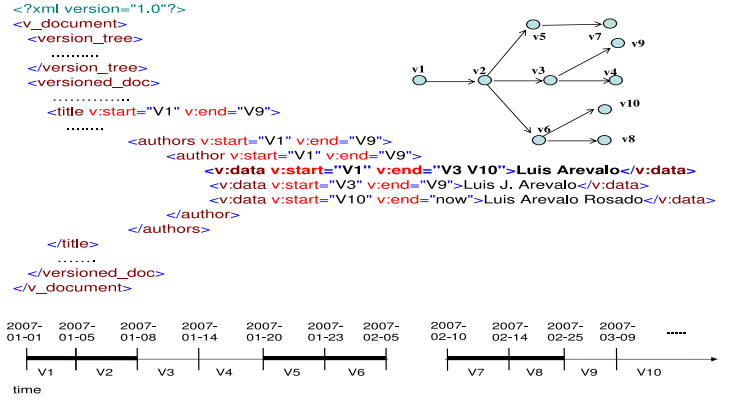

Fig. 2. Versioned elements with version region

(a sub-tree of the version tree). A version region is a [start-End] pair where the start value is a version identifier that represents the origin node of the valid area in the version tree and End is a *set* of version identifiers that indicate when each area has stopped being valid. In this way each element in the versioned document is formed by a version region that is converted into two attributes, v:start and v:end. The first one is defined as an IDREF datatype attribute which refers to a version identifier from the version tree and the second one defined as IDREFS datatype which allows us to represent a set of version identifiers from the version tree.

In figure 1 and 2 a vDocument is shown. On the one hand in figure 1 the version tree with several versions of an XML document is shown: i.e: from the version identified by V_2 several changes have been made (identified by V_3, V_5, V_6). On the other hand in figure 2 several versioned elements are shown. i.e: the first *author* element is valid from V_1 and stops being valid in V_9, this means that it is valid for all descendants-self of the V_1 version except all V_9 descendant-self versions. Another example is the first *v:data* child for *author* element which is valid from [V_1, {V_3,V_{10}}] so it is valid in the versions identified by V_1,V_2,V_5,V_7,V_6 and V_8 since all descendant-self of V_3 and V_{10} are not included meanwhile the second v:data of this *author* is only valid for all descendants versions of V_3 except descendants-self of V_9. The special value "now" in the attribute v:end indicates "no changes until now", in other words, the version region is formed by all version descendants from the v:start attribute. Obviously, we have to take into account that an element cannot exist without its ancestor elements.

3.2 Temporal Time in vDocuments

When a new version of the document is generated in a vDocument, these changes happen at some point in time. Until now, we have only represented the relationship between the versions in vDocuments without taking into account when these changes occurred, this means that, the temporal validity information associated to each version is lost. In this section we show how to integrate the valid-time axis in a vDocument calling as *VTstamp*.

Temporal database researchers have focused on three principal dimensions of time [22]: valid time, transactional time and user-defined time. In this work, we have decided to model the valid-time axis, although the other axes can be managed in the same way. The valid time of a fact is defined [22] as the time when the fact is true in the modeled reality, in our case, the valid time of a version is when the version is true. We have decided to include the valid time by means of a time interval, a pair of two time instants [t_1,t_2] that is turned into two attributes for each version defined in the document as shown in figure 1. The following restrictions must be carried out: 1) For each version defined in the version tree, the value of t_1 instant must always be less than t_2 2) Any two time intervals from the version tree cannot overlap and 3) We assume that time is bounded.

On the other hand it is also necessary to define the valid time for each tag included in the document, that is when this tag is valid. Using the version region used in our technique, we can define its temporal validity easily. Due to the fact that a specific tag is valid in a set of versions from the version tree, this means that, this tag will also be valid in each period of time for each valid version. For example in figure 2 the temporal validity of a specific v:data tag which is valid in the following version: [V_1, {V_3,V_{10}}] is shown, therefore it will be valid in the following time intervals {[01-01,01-05], [01-06,01-08], [01-21,01-23], [01-24,02-05], [02-10,02-14], [02-15,02-25]} (shown with a thick line above in the figure). Notice that some of these time intervals can be joined forming a continuous period of

time (coalesce) i.e: [02-10,02-25], however, this is not advisable since they are placed in different branches from the version tree.

3.3 Changing and Updating a VXML Document

As has been said, XML documents are not static, so it is necessary to manage inserts, deletes or updates that can modify them [20]. Beginning at the initial state of the document (version 0), new versions are then established by applying a number of changes to whatever version defined in the document. Once we know how to represent versions in XML documents, the following questions will be: what kind of change operations can generate a new version? And, how to update the XML versioned document from a change operation?.

In order to answer the first question, we have analyzed which items can be changed in an XML document and which operations can be performed on them. However, before this, it is necessary to identify thoroughly those elements which have been changed from the current version. Among the different possibilities shown in [4], we have decided to add an attribute *idf* to each element in the document in order to identify it in a vDocument, with the exception of v:data, v:attrib and v:isref because those elements are identified by its parent element. Thus, the basic structural XML operations, common in whatever document based on the XML specification, are shown in table 1.

Although *move operation* can be represented as a delete and an insert operation we have decided to include it as one of our basic operation since it is a very frequent change in XML documents. According to the consistency principle, to accept the execution of each primitive a restriction must be satisfied, that is, the document obtained must be well-formed, and each version of the document must be valid in accordance to the specifications of its XML-Schema. To guarantee this, a whole set

Table 1. XML changes primitives

Operations	Meaning
IE (idf,name,pos)	It adds a new element with the name *name* from the parent *idf* in the position *pos*.
DE(idf)	It removes the element identified by *idf*. All valid descendant elements are deleted too.
RE(idf, name)	It renames the name of the element identified by *idf*.
IA(idf, name, value)	It adds an attribute for the *idf* parent.
DA(idf, name)	It removes the attribute called *name* from the element identified by *idf*.
UA(idf, name,n_value)	It changes the value of the attribute called *name* for the parent *idf* for the *n_value*.
IC(idf,data)	It adds a PCDATA text for the *idf* parent.
DC(idf)	Removes the content for the element identified by *idf*.
UC(idf, data)	Changes the PCDATA value for *idf* to *data* value.
ME(idf,idf_from,pos)	Moves the element identified by *idf_from* and its descendants to element *idf* in the position *pos*.

of pre-conditions to be fulfilled have been defined for each single operation before producing a new version of the document. For example: 1) the "idf" parameter for all operations must exist for the version we want to update, 2) the name of the attribute in IA operation implies that another attribute for this element cannot exist from the current version (there cannot be two attributes with the same name) and 3) the DC operation cannot be carried out if there isn't any PCDATA information for the required identifier.

These basic updated operations can be obtained mainly by means of two techniques. On the one hand, obtaining the XML operational differences between two versions by means of several approaches such as [4,18,19] or on the other hand from a certain version specifying which changes we want to carry out. The technique proposed in this work is based on both solutions, needing, therefore, a mechanism to integrate them. This consists of representing each update operation exposed previously in an XML format.

In this way if an approach based on differences is chosen, then an XSLT stylesheet, which transforms this XML document with differences to our XML representation, is defined. From [10], where several XML diff approaches are analyzed, we have decided to choose JXydiff [25] which is a Java tool for detecting changes in XML documents based on Cobena's work shown in [4]. We chose this for the following reasons: 1) It has the main features to retrieve XML differences: can manage all kind of XML nodes, can detect move and update operations and is based on a tree oriented algorithm, 2) It is written in Java, so its integration in our implementation is immediate and 3) It is very easy to export its output XML differences to our XML representation by means of an XSLT stylesheet. As a future work, our idea is to use a relational-based approach [17] for detecting changes in XML documents due to scalability problem that suffers the main-memory Diff algorithms mainly in Java. On the other hand, if we decide to change the document manually, the change editor has only to generate a batch document with update operations in our XML representation.

In this way, the creation of a new version is defined by a set of the aforementioned operations represented in an XML document with changes, which we call an *XML transaction document*, as is with the concept of transaction in databases, the vDocument is updated if and only if all changes are executed. This transaction is carried out in the following three phases:

Phase 1) Retrieval of the version to modify. The document to work on will be the version of the XML document obtained from the vDocument, to which the XML change transaction will be applied.

Phase 2) Modification of the retrieved XML document.

Phase 3) Updating of the versioned document. a). Obtain the XML transaction document b). Execute each operation from this XML to the vDocument and c). The new version and its associated temporal information is added to the version tree.

In figure 3 the XML transaction schema is shown as well as a practical example. As we can see, an XML transaction document may be formed by several versions where each version may be formed either by a sole operation or by

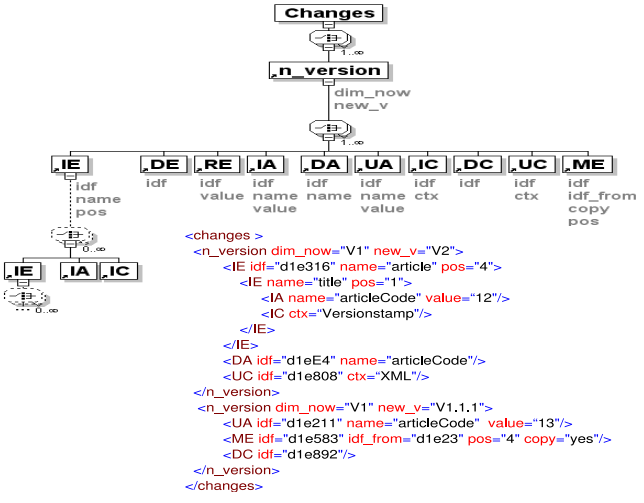

Fig. 3. Schema and an example of an XML transactional document

means of several of them (the parameters of each operation from table 1 are defined as attributes). For example, the first DA operation shown in figure 3 is formed by two attributes: *idf* that stores the parent identifier (d1eE4) and *name* (articleCode) that is the name attribute to delete. Another example is the IE operation, InsertElement, that can be formed by one or several IE/IA/IC operations as is shown in the same figure. In that case, the first IE operation inserts an element which has a child element which contains an attribute (IA) and a PCDATA content (IC).

Related to the second question about how to update a VXML document when a basic change operation is produced the following actions are carried out. When an insert operation is made, the new element/attribute/content is inserted in its position setting the v:start attribute to the new identifier version and the v:end attribute to "now" value. In the case of a delete operation, it is only necessary to change the v:end attribute of all affected items setting them to the new identifier version. For update operations the v:end attribute for the current item is set to the new identifier version and the new element/attribute/content is added and its version region attribute is set as in the insert operation. In the case of a move operation, the affected items are modified as in the update operation.

One of the most important advantages of using an XML document to define the update operations, is that it allows us to manage changes for any markup language based on the XML specification, since these update operations are common to all of them. Thus, to specify the changes of a certain XML language, it would be only necessary to define it by means of these primitives. In this way, as a future work we will use this technique to manage versions of XSLT and SVG document. Moreover, this technique can be also used to represent the version history of an XML schema document.

4 Retrieval in vDocuments

One of the main advantages of this proposal is the wide set of queries we can specify both using version and temporal axis. In this way, classical temporal XML queries can be made such as temporal projection, snapshot, etc and also version queries such as version projection, snapshot version, etc. Here, we will show some of them that are used in the following section to measure our technique.

Q1: *Version snapshot query*

In order to retrieve the valid labels for a given version it will be necessary to analyze which versions are included in a version region and check if the requested version belongs to them. This occurs only if 1) the given version is among the descendants in the "start" version identifier in the version tree or even is itself and 2) the given version is not among the descendants or is itself in all version identifiers for "end" attribute. To do this effectively, we have to obtain which versions are in a version region and check if they contain the requested version. We use the id() function provided by XPath to obtain the versions by means of dereference the version/s in the version tree which v:start and v:End refer to (they are defined as IDRef and IDRefs datatypes respectively) and thereby we can easily obtain their descendants and check the constraints said before.

We have defined a version operator called *Vmeets* as a user-defined function (line 1) that check (line 4) if the given version belongs only to the v:start attribute (line 2) and not to the v:End attribute (line 3). That query retrieves all nodes valid for V_8 version (line 6). In the same way, other version operators are able to been defined as: Vancestors, Vparent, Vcontains, etc.

```
1. declare function f:Vmeets($p,$v) as xs:boolean{
2.     let $start:=$p/id($p/@v:start)/descendant-or-self::version/@xml:id
3.     let $end:=$p/id($p/@v:end)/descendant-or-self::version/@xml:id
4.     return (($start=$v) and (not($end=$v))) };
5. <data>{
6. for $s in //versioned_ doc//*[f:Vmeets(.,'V8')]
7. return $s
8. }</data>
```

Q2: Count the number of the title element valid for version V8 using Xpath

Using the id() function, we can query the vDocument using another XML standard query language such as XPath. In the following query all title elements valid for version V_8 are counted.

```
count (//*title[not(id(./@v:end)/descendant-or-self::version/@xml:id='V8' ) and
(id(./@v:start)/descendant-or-self::version/@xml:id='V8')]
```

Q3: *Temporal snapshot query*

Since temporal information has been added to our vDocuments, we can retrieve it by means of the valid-time axis. To do this, it is necessary to find out in which version the given time belongs to. If a time instant is given, a user-defined

function called *tmeets* (line 1) retrieves which version contains this time. After that, the previous version snapshot is executed (line 5, 6). In the case of a time interval, a user-defined function called *tContain* is defined which verifies which version contains the requested time interval. Q1 query using the valid-time axis is shown below.

```
1. declare function f:tmeets($time) as xs:string{
2.    let $id:= //version[(./@tvstart<=$time) and (./@tvend>=$time)]/@xml:id
3.    return $id };
4. <data>{
5. let $version:=f:tmeets("2007-02-20")    //This instant belongs to V8
6. for $s in //versioned_ doc//*[f:Vmeets(.,$version)]
7.    return $s
8. }</data>
```

5 Experimentation and Implementation

In this section several experiments have been carried out in order to compare our technique to a timestamp XML approach and some details of its implementation are also shown.

5.1 Experimental

The testing machine is a Pentium Mobile 1,8GHz PC with Linux (Ubuntu), with 1024MB memory and a 120GB IDE hard drive. The data shown in the graphics are the performance average on 3 identical tests. We have developed a Java application to generate a large amount of version data where the operations from the table 1 are selected at random, assigning a higher probability to the insertion of elements. Once selected a primitive, the current version and the affected node are selected at random too. The tests have been carried out on cases of lineal versioning and branch versioning. In the latter case, we have selected at random the version we want to update according to the following probabilities a 20%, 50% and 80% possibility of choosing a different version from the current one.

The experiments were carried out on 5, 10 and 20 changes per version, for 100, 60 and 30 versions respectively thereby evaluating the behavior of our system in the following cases: a large number of versions with few changes (100 versions - 5 changes), a medium number of versions with some changes (60-10) and a small number of versions with many changes (30-20). In the experiment, we selected the ACM XML Sigmod Record supplied in [26] (November of 2002) where three different versions of this document were used: small, medium and large. All characteristics of these documents can be consulted in figure 4.

We have also developed a temporal timestamped XML solution (tstamp) in order to compare it with ours. In this way, we have chosen the technique shown in [8], based on adding a time interval to each label in the document, allowing the incorporation of temporal information in the XML document. All our versioned lineal XML documents have been converted to temporal ones. The resulting

		Size	Element		Attribute	Text
ACM Sigmod XML	Small	42028	687		411	417
	Medium	225487	3688		2285	2317
	Large	545368	8930		5677	5769
		Size	Element	Attribute	v:attrib	v:data
ACM Sigmod Vdocument	Small	95519	691	4545	411	417
	Medium	522093	3692	24820	2285	2317
	Large	1281000	8934	60976	5677	5769

Type. Versions/Changes		Tstamp	Lineal	20,00%	50,00%	80,00%
Vdocument	S. 100/5	191113	222245	229027	229992	236706
	S. 60/10	203406	228582	250637	240359	261261
	S. 30/20	197510	216406	219185	233103	233238
	M. 100/5	557512	640329	653778	669064	647238
	M 60/10	584815	660593	671610	658810	674334
	M. 30/20	583351	651263	659213	664908	658625
	L. 100/5	1216703	1386684	1406887	1391842	1395761
	L. 60/10	1250378	1415846	1424534	1403901	1405975
	L. 30/20	1245227	1407287	1422153	1409936	1412069

Fig. 4. a. Characteristics of the document. b. Resulting vDocument size.

version size document is shown in figure 4b where it can be seen that the size of our vDocuments are a bit higher than the timestamped solution.

The retrieval time obtained refers to the transformation time in a client application, regardless of the document loading time in memory or transmission where the retrieval time has been calculated on 3 performances. To do it, we have used the Saxon processor [27] where the following queries have been carried out:

* Q1: Version/Temporal Snapshot query using XSLT.
* Q2: Find the total number of *title* elements valid for a version in XPath.
* Q3: Retrieve those authors and their descendants valid for a version in Xquery.
* Q4: Snapshot query using an optimized XSLT.

In figure 5.a the retrieval time (measured in ms) obtained using an XSLT stylesheet is shown (query Q1). This figure shows the retrieval time using the timestamped solution (Tstamp), using the versionstamp solution (VStamp) and the versionstamp solution on a temporal level (VTstamp). As we can see, our solution here behaves less efficiently than the timestamp solution, since the time solution uses the operators $<=$ and $>=$ to verify if a time belongs to a time interval, meanwhile in our process we have to retrieve all descendant identifiers for the v:start and v:end attributes. In this way, both Vstamp and VTStamp greatly depend on the number of versions that the document has as well as the size of it. In some cases (short documents or documents with few changes) our performance is quite similar to the timestamped solution, however our solution in lineal versioning performance is poorer. This can be seen in figure 5.b and figure 6.a where the retrieval time for Q2 and Q3 query are shown.

To avoid this situation, we have developed an optimized solution that consist of storing within each version their descendants allowing us not to have to constantly recover this information in each query. Therefore, each version in the version tree will have a new attribute called *descen* that stores its descendants. In this way if we want to check if a requested version belongs to a version region, it is only necessary to verify if the *descen* attribute for the *v:start* version

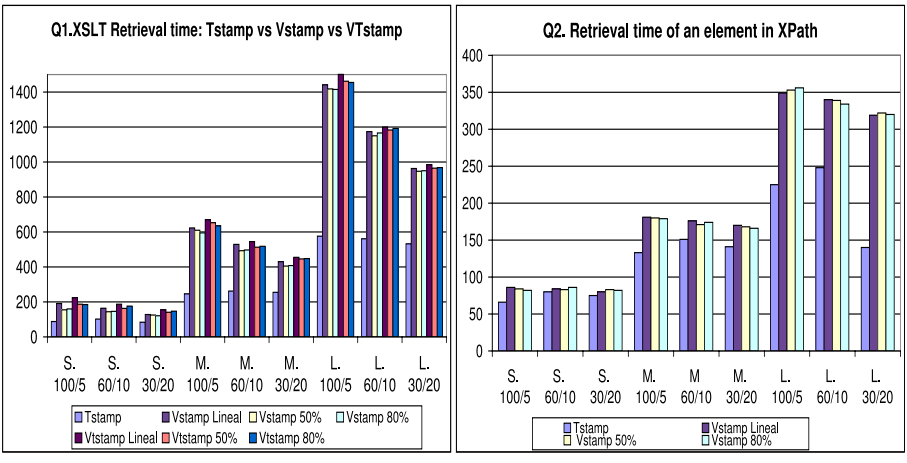

Fig. 5. Retrieval time a. Q1 query b. Q2 query

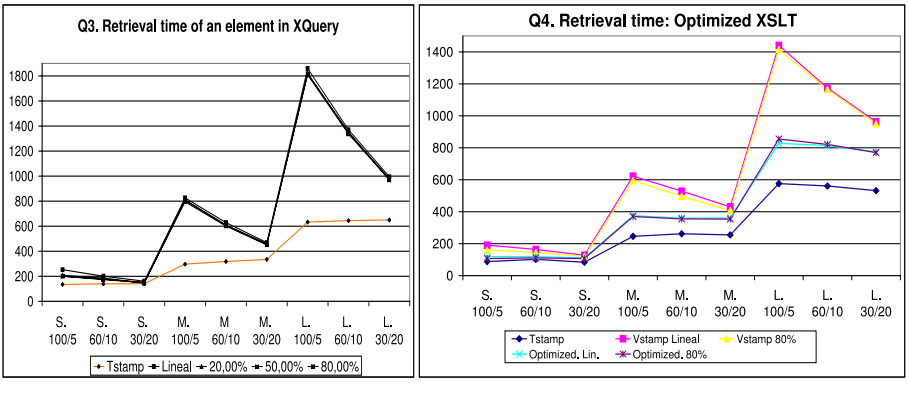

Fig. 6. Retrieval time a. Q3 query. Q4 query.

contains the given version and it is not present in the *v:end* attribute. We can see this improvement in figure 6 where we can verify that using it the retrieval time is reduced considerably and in some cases the retrieval time is quite close to the timestamped solution. In those cases that our solution had its poorest performance (large documents or several versions) this time has been reduced by up to 50%. Notice that, this solution is almost independent from the number of versions, since it is not necessary to retrieve the descendants of the v:start and v:end attribute.

Although it can be argued that our solution performs poorly in large documents, it offers many advantages that timestamped solutions cannot: we can query versioned documents both on a version and a temporal level, manage branch versioning that is not supported in timestamped solutions and extend the number of temporal/version queries that can be made.

5.2 Implementation

The system has completely developed using XML technology. To execute it we just need an XSL stylesheets, Xquery or XPath processor with support for id() XPath function (tested in Exist, Saxon and Xsltproc processor). To check its functionality, we have developed a set of Web Services to manage versions of XML documents. Our proposal is to develop a generic engine to store, manage and query the different versions from an XML document through Internet thanks to Web Services without to set any additional software. The most important advantages of this engine is the possibility to offer them to third-party clients to either version their data or to develop a more complex versioning system by means of invoking our Web Services.

Table 2. Versioning Web Services

Group	Web Service	Brief description
Conversion	doc2vdoc	Generates an XML versioned document from an XML document.
Conversion	vdoc2doc	Retrieves the reconstructed XML document from a specific version
Get	getDocs/VDocs	Retrieves a list of documents/versioned documents stored in the system for a specific user.
Get	getVersions	Retrieves the available versions for a vDocument
Get	getInfo	Gives information about each update operation (parameter, error, etc)
Query	getQuery	Executes XPath or XQuery in a vDocument.
Changes	Primitive	Used to run an update operation for a specific vDocument
Changes	execTrans	Executes an XML transaction document
Changes	exec_Randontrans	Executes an XML random transaction document
Manage	uploadXML/ VXML	Allows us to upload XML/VXML to the repository
Manage	deleteXML/ VXML	Allows us to delete a XML/versioned document from the repository
Diff.	GetDiff	Obtains the differences between two versions
Diff.	getXMLDiff	Retrieves a specific version of the document from the Vdocument.

Our Services have been developed using Java, more specifically the API called AXIS [24] from the Apache Software Foundation. AXIS has proven itself to be a reliable and stable base on which to implement Java Web services. Initially, we propose a set of 16 Web Services that can be classified in six groups as shown in table 2 (parameters of each service is omitted in this work due to lack space). In order to carry out some trials on these services we developed a client prototype too as it is shown in the following URL: http://exis.unex.es/versionado/.

6 Conclusions and Future Work

Document version management has been used for years mainly in collaborative environments by means of, on the one hand, diff lined-based approaches or delta XML, however these solutions are not recommended in XML documents since they can neither validate nor query the XML versioned document and, on the other hand, XML temporal document solutions, based on the timestamped technique, which have difficulty in supporting non-lineal versioning. To solve these problems we proposed a versionstamp technique in [9].

In this paper, we have extended it by means of adding temporal information to each version included in the vDocuments. Not only does it allow us to query the vDocuments on a temporal and version level but also we can manage branch versioning in temporal documents. Moreover we have also defined the basic updated operations common to whatever XML document, describing them by means of an XML document called *XML transactional document* which allows us to manage changes for any markup language based on the XML specification.

Finally we have compared our solution to a timestamped XML one. Although it performs poorly in some cases we have improved it by means of an optimized solution thereby offering us many advantages that timestamped solutions cannot achieve. Moreover, we have developed a set of Web services which do not have portability restrictions and allows us not only to manage the different versions of an XML document but also to validate, transform, store and query them in an easy way. Since our proposal is open, it can be used for third-party clients either to manage their documents or to extend them by incorporating new features.

As future work we propose these following steps:

* To analyze new queries in XML versioned documents as range queries, temporal/version queries, temporal overlapping queries, etc.
* Compare the results storing the documents in native XML databases and in relational databases.
* To define the version region by means of a set of sub-graph nodes allowing us to represent element temporal interval.
* To implement a versioning system based on XUpdate.
* To extend these services by incorporating some features of traditional version control systems such as security, lock files, indexing the document to run the queries faster, etc.
* To apply our versionstamp technique to other XML markup languages such as XSLT stylesheets, SVG graphics or even to XML office documents as OpenOffice or Microsoft Office.

References

1. W3C, `http://www.w3c.org`
2. CVS. Concurrent Versions System, `http://www.cvshome.org`
3. Subversion, `http://subversion.tigris.org/`

4. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in XML documents. In: Proceeding of the 18th International Conference on Data Engineering (2002)
5. Chien, S-Y., Tsotras, V.J., Zaniolo, C.: Efficient management of multiversion documents by object referencing. VLDB (2001)
6. Vagena, Z., Moro, M.M., Vassilis J.: Tsotras. Supporting Branched Versions on XML Documents. In: RIDE (2004)
7. Salzberg, B., Jiang, L., Lomet, D.B., Barrena, M., Shan, J., Kanoulas, E.: A Framework for Access Methods for Versioned Data. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, Springer, Heidelberg (2004)
8. Wang, F., Zaniolo, C.: XBiT: An XML-based Bitemporal Data Model. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 810–824. Springer, Heidelberg (2004)
9. Rosado, L.A., Márquez, A.P., González, J.M.F.: Representing versions in XML documents using versionstamp. ECDM (2006)
10. Ronnau, S., Scheffczyk, J., Borghoff, U.M.: Towards XML Version Control of Office Document. In: Proceedings of ACM DocEng. (2005)
11. Grandi, F., Mandreoli, F.: The valid web: An XML/XSL infrastructure for temporal management of web documents. In: ADVIS (2000)
12. Dyreson, C.E.: Observing transaction-time semantics with TTXPath. In: WISE (2001)
13. Zhang, S., Dyreson, C.E.: Adding valid time to XPath. In: Bhalla, S. (ed.) DNIS 2002. LNCS, vol. 2544, pp. 29–42. Springer, Heidelberg (2002)
14. Amagasa, T., Yoshikawa, M., Uemura, S.: A data model for temporal XML documents. In: Ibrahim, M., Küng, J., Revell, N. (eds.) DEXA 2000. LNCS, vol. 1873, Springer, Heidelberg (2000)
15. Wuwongse, V., Yoshikawa, M., Amagasa, T.: Temporal Versioning of XML Documents. In: Chen, Z., Chen, H., Miao, Q., Fu, Y., Fox, E., Lim, E.-p. (eds.) ICADL 2004. LNCS, vol. 3334, Springer, Heidelberg (2004)
16. Galante, R.M., Santos, C.S., Edelweiss, N., Moreira, A.S.: Temporal and Versioning Model for Schema Evolution in Object-Oriented Databases. In: Transactions on Data and Knowledge Engineering (2005)
17. Leonardi, E., Bhowmick, S.S., Madria, S.K.: Xandy: Detecting Changes on Large Unordered XML Documents Using Relational Databases. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, Springer, Heidelberg (2005)
18. Mouat, A.: XML diff and patch utilities. Master's thesis, Heriot-Watt University, Edinburgh, Scotland (2002)
19. Wang, Y., DeWitt, D.J., Cai, J.: X-Diff: An effective change detection algorithm for XML-documents. In: Conf. on Data Engineering, IEEE CS Press, India (2003)
20. Xquery Update. `http://www.w3.org/TR/xqupdate/`
21. Snodgrass, R.T.: The TSQL2 Temporal Query Language. Kluwer Academic Publishers, Dordrecht (1995)
22. Jensen, C.S., Dyreson, C.E., et al. (eds.): The Consensus Glossary of Temporal Database Concepts (February 1998)
23. Tatarinov, I., Ives, Z.G., Halevy, A.Y., Weld, D.S.: Updating XML. In: ACM Sigmod. (2001)
24. Apache AXIS. Retrieved From: `http://ws.apache.org/axis/`
25. JXydiff. `http://potiron.loria.fr/projects/jxydiff`
26. ACM XML Sigmod Record. `http://www.sigmod.org/record/xml`
27. Saxon. `http://www.saxonica.com`