An XQuery-based version extension of an XML Native Database

Luis Arévalo Rosado
University of Extremadura
Avenida de la Universidad s/n
10071 Caceres (Spain)
ljarevalo@unex.es

Antonio Polo Márquez
University of Extremadura
Avenida de la Universidad s/n
10071 Caceres (Spain)
polo@unex.es

Miryam Salas Sánchez
University of Extremadura
Avenida de la Universidad s/n
10071 Caceres (Spain)
mirym@unex.es

ABSTRACT
The goal of this paper is to show our experience in the design of an extension to an XML native database in order to add in a native way version features. Due to the lineal nature of time, XML versions have difficulty in supporting non-lineal versioning, and their implementations are scarce and not very portable. From our technique described in [22, 23], which allows us to manage branch versioning, we have developed our versioning system focusing on its high portability and platform independence properties. To achieve this, our versioning system is based exclusively on XML technology. The system is composed by XQuery modules, so that this allows its use not only in an XQuery engine but also in an XQuery-support XML native database. In order to test it, the modules have been integrated in an eXist XML native database, an XML difference tool is used to observe the changes made between versions, and the eXist web interface has been extended to make the management of XML versions easier.

Categories and Subject Descriptors
I.7.1 [Document and Text Processing]: Document and Text Editing- Document Management, Version Control

General Terms
Algorithms, Management, Performance

Keywords
XML versions, Branch Versioning, Historical XML Information, XML Native Databases

1. INTRODUCTION
The XML standard [26] is a consolidated tool for representation and interchange of information, due to both its flexibility and the management and conversion facilities that the documents offer in this standard. Besides, nobody doubts it has remarkably influenced the increase of our digital society so that it is used in multiple facts such as information interchange (SOAP, WSDL, ...), configuration file, XML office documents (Open-Office, Microsoft Word,...) and mainly to store information, where different approaches can be used: in a flat file, in a relational / object-oriented database or more recently in an XML Native Database.

Furthermore, XML information changes throughout time and its management becomes necessary to query past information, to retrieve documents belonging to a specific version or to monitor the changes. For example, imagine a design based on SVG which obviously evolves over time until achieving the desired design. In this scenario users should be able to update any version of the document generating a new version either from the current one or discard it and reuse an old version.

Version management has been used for years in such environments like collaborative software development, file share resources, historical information retrieval, etc. More recently, with the appearance of XML [26], it has become necessary also to manage changes in these documents.

Versions of an XML document can be managed through traditional systems like CVS [1] or subversion [25], or traditional adapted procedures based on XML operations changes (delta XML) [8, 7] or by integrating the different versions into a single XML file using a temporal [27, 5, 30], a version [22] or a temporal-version [29, 11, 23] technique.

From an operational XML point of view, 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), to support branched versioning (temporal solutions do not) and, to have the possibility to query the XML versioned documents using some XML standard languages like XQuery and XPath (first and second solution do not).

From an execution point of view, the majority of the previous solutions require the setting up of a specific software, which often prevents its portability. Regarding temporal XML solution, several XML data models have been proposed [5, 15, 20, 27] but only a few implementations have been made and none of them is available to be re-used.

For these reasons, from our technique shown in [22] for managing and querying branched versioning of XML documents, we have developed our versioning system, much more portable, complete and operational than [23], which focuses on its high portability and platform independence. To achieve this aim, a set of XQuery modules have been developed which can be integrated either in any XQuery engine or XQuery-support XML native databases and, a difference
tool [4] has been used to obtain the differences between two versions. Integrating both, XQuery script and the diff-tool, eXist XML native database has been extended to support versioning features in a native way as well as the eXist web interface to help us in the management of XML versions.

The remainder of this paper is organized as follows. We begin by summarizing the current solutions for the management of XML versions. Then, we describe our versioning technique. Next, we show how we have extended an eXist XML Database by means of XQuery modules as well as its web interface to include version features and, finally, we offer our conclusions and our future works.

2. STATE-OF-THE-ART

Document version management has been used for years mainly in collaborative environments. Traditional techniques [1, 25], based on diff line-based algorithms, are inappropriate for XML documents since its structure is lost and XML queries are not supported. 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 for version management are mainly based on:

Delta XML management is based on traditional change operation procedures adapted to XML [8, 7]. It consists of obtaining and storing the XML differences between two versions (delta XML). However, neither XML validation nor XML query can be carried out in these solutions.

Temporal XML Representation, that is based on temporal database topics [16] for representing and managing historical information in XML. Several approaches have been proposed such as: [14] a technique for managing temporal web documents using an XML/XSLT infrastructure, [5, 9, 20] data models for temporal XML documents, extensions of XPath data model in [30, 10] to represent and query transactional and valid time respectively, [6] presented a timestamp-based approach to archive scientific data [4] where they focus on how to integrate different versions to one document with nodes timestamped, and a temporally-grouped data model [27] that gives us a way to represent the content database evolution using XML timestamps. However, since they use timestamped markup to represent the versioned tags, all of these solutions present difficulties in supporting non-linear versioning.

Integration Time-Version: The integration of time and version concepts to manage dynamic information has been studied recently in [29, 23, 11] for XML and object-oriented databases respectively. In [29] the authors defined temporal delta (tDelta) and introduced version time in it, however, query support is not discussed. Our versioning technique [22], based on this approach, integrates both axis (version and time) in an only XML document, being able to run queries in both levels [23].

Furthermore, not only have Temporal XML representation difficulties in supporting non-linear versioning, as has been said previously, but also in many of them, temporal XML queries are not discussed and their implementations are few and not very portable. Regarding temporal XML queries, they have been studied in some papers [27, 18, 13]. In [27], traditional temporal queries such as projection, snapshot, slicing, join, etc, have been re-adapted to XML by means of user defined functions and in [13] authors proposed an extension to XQuery for temporal support (rXQuery language) where queries are translated into XQuery. However they can only make queries in the temporal level since the version one has been obviated.

In addition to that, several temporal XML data models [5, 27, 15, 20] have been proposed, although only a few of them have been implemented, and those that have been, require the setting up of a specific software, which often prevents its portability. For example, in [28] XChronicler was proposed which is a versioning system to store historical XML documents in a ArchIs database; XML version systems have been also developed using indexing techniques in [20, 19], both being non-portable implementations.

In order to avoid portability restrictions we looked for a way of developing our system using XML standard languages. The system has been developed using XQuery scripts where they can be integrated in any XQuery engine or in XQuery-support XML native databases. It is based on our technique proposed in [22] which supports branched versioning and versioned documents can be queried using several XML standard query languages: XQuery, XPath and XSLT.

3. PRELIMINARIES

In this section we present the foundations our work is based on [22]. The aim of a versioned document is to be able to store different versions or states of a document. Beginning at the initial state of an empty document (\( V_0 \)), new versions may be established by applying a number of changes to a version. In order to manage versions of an XML document, mainly two solutions could be chosen: 1) a system that stores every snapshot of the document and maps the relationship between them such as backup system, 2) to use one of the techniques described in state of the art, for integrating versions. As shown in [28], where they analyze in details both solutions, the first solution shoots up greatly the storage capacity, therefore this leads us to choose a technique to integrate versions.

In the early techniques for managing versions of XML documents, it was assumed that it was not advisable to store all versions in a single document, mainly for capacity reasons [28]. The proposed solution was obtaining and storing the XML differences between two versions (delta XML) [8, 7, 24] which implies a high reconstruction cost for any version different from the current one. However, as shown in [28, 19], nowadays the fact of storing all versions into a single document has not so much impact of space and time, due to the capacity and speed features of the new devices. Therefore a good solution is to use a technique that integrates all versions into a single document, and knowing the difficulty of supporting branch versioning of XML temporal solutions, we will use our technique proposed in [22].

In [22] we defined an XML versioned graph data model, called a V-XML data model, in order to represent versions in XML graph documents by means of adding versionstamp information for each versioned item, so obtaining a new XML document which we called a vXML Document or vDocument. An XML versioned document has two sections: The first one which stores all information about the included versions and the relationship between them The second one, each element in the document is turned into a versioned element by means of defining its version validity, that is, for which version/s of the document it is valid.

The version_tree represents how the versions have been
A vXML document with several versions made over time (shown in figure 1.a). Each state of the document included in the graph is an XML element (version element) and represents different snapshots. The nesting of these version elements stands for the relationship that may exist between them.

In the second one, the vXMLElement, each element is transformed into a versioned element by defining its version validity. We used a version stamp technique, called as Version Region [22], that is defined by a set of version identifiers from the version tree (a sub-tree of the version tree). In this way each element is formed by two attributes, v:start and v:end that refers to the version tree (an IDREF and IDREFS datatype respectively). The v: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 has stopped being valid. There is a special value "now" in the attribute v:end indicates "no changes until now" from the start version.

A vXMLElement with several versions is shown in figure 1a which is formed by an XML version tree and some versioned elements. In this example, the text of the first author has changed in versions V3 and V10 and has been deleted in V9. This fact can be check since the author element has three v: data elements with different version validity. For example the first v: data child for author element, squared in the figure, is valid from [V1, {V3, V10}] so it is valid in the versions identified by V1, V2, V5, V7, V6 and V8 due to all descendants-self of V3 and V10 being not included (shown graphically the valid versions in the figure 1.b).

The most important advantages of this technique are: it is able to manage branched versioning that is not supported in timestamped solutions, vDocuments can be queried using several standard XML query languages such as XQuery and XPath as shown in [22] and allows us to integrate temporal information associated to each version as shown in [23].

4. VERSIONING SYSTEM

Once we have shown how to represent versions in XML documents, the next step deals with how to implement our versioning system. The proposed system works as shown in figure 1b. Beginning at the initial state of the document V0 or an existing XML document, where it is turned into a vDocument, the new versions are integrated in the vDocument with the rest of versions. To do this, firstly, we will use a difference tool to obtain the changes that have been made between two versions and then the elements or attributes inserted or deleted are added or removed logically in the document by using the guidelines described in [23].

Therefore, in this section we explain our versioning system to manage versions of XML documents. Firstly, we explain our XQuery modules which have been integrated into an XML native database, eXist[12]. Later we show how the eXist web interface has been extended to add versioning features and finally the achieved results are discussed. The proposed versioning system can be tested in the following url: http://exist.unex.es/versionado.

4.1 Versioning Modules

Our versioning system can be developed in two ways: 1) by using the same programming language used by the processor or the XML native database where our system will be run 2) by using a standard query language that allows us to integrate it into any system. For portability reasons we have decided to handle versions of XML documents by means of a set of XQuery modules that allows us to integrate them in any XQuery processor or in any XQuery-support XML native database (XNDB).

These scripts are divided into 7 modules which are composed of 1200 code source lines where the Version module has the public API used as a version manager. These modules are:

- vUtil Module: This module has plenty of basic operations on XML documents such as checking if the input/output file is well-formed, the existence of a specific node, etc, as well as basic operations on different types of data. It is very useful for the rest of modules.

- vStamp Module: It is formed by all version operators. Temporal solutions use predicates based on the operators <, <=, >, >= to make the comparison; however, in our solution, since the historical representation is based on a subtree
from the version tree, these comparison predicates have been re-defined. This module is used by vQuery module.

- vQuery Module: This module is formed by several version queries on vDocuments. In [23] we added the temporal axis to our vDocument allowing us to make temporal/versión queries on XML versioned documents such as temporal/versión history and snapshot queries.

- vUpdate Module: From our point of view, this is one of the most important modules of these scripts due to the fact that it allows us to update an XML versioned document. To do it, a set of update operations were defined in [23] based on element, attribute and content changes as well as its representation in an XML document. The creation of a new version is defined by a set of these operations in this XML scheme, calling it an XML transactional document. A vDocument is updated if and only if all changes represented in an XML transactional document are executed.

- vDiff Module: This module uses a diff-XML tool, that is, it provides us with a set of functions to obtain the differences between two XML documents, two versions of a vDocument or between an XML document and a vDocument's version. Currently, we use JXyDiff [4] to obtain the differences, however any other difference tool could be used by means of developing its associated XQuery script.

- vConversion Module: Those operations that take an XML document and turn it into another one are included in this module. The main function of this module is the conversion between an XML document to a vDocument and vice-versa.

- Version Module: This defines a set of functions to manage versions of an XML document. This API is shown in table 1 where a brief description of each function is included. We would like to clarify some features of these functions:

  - The GetDiff function can be run not only on two XML documents but also on two versions of a vDocument or on an XML document as opposed to a vDocument’s version. For the sake of clarity the signature of the first operation has only been shown in this table, however the rest of them have been included in our API.

  - The update operation works as follows: Firstly, the differences between the vDocument’s version and the given XML document are obtained using the vDiff Module. Later these differences are transformed into our normalized representation (vConversion Module) and finally they are executed by the vUpdate Module.

  - The Check module verifies 1) if the historical representation in the $vDoc_Uri is well formed, fulfilling the rules shown in [22] 2) if all elements in the document have a unique identifier.

This system uses a difference tool to obtain the changes between two versions. From [21], where several XML diff approaches were analyzed, we decided to choose JXyDiff [4] which is a Java tool for detecting changes in XML documents. We chose this for the following reasons: 1) It has the main features to get XML differences: can manage all kinds of XML nodes, can detect move and update operations and is based on tree oriented algorithms. 2) It is written in Java, so its integration in eXist is easier than others developed in other languages 3) Its output with the XML differences is very easy to export to our XML representation by means of XQuery scripts.

By means of the Version module, we can carry out the typical operations of the traditional versioning system such as: create an XML versioned document (doc2vdoc), to retrieve a valid version from a vDocument (vDoc2doc), to get the versions included in a vXML (getVersions), to update any version stored in a vDocument from an updated XML document (Transaction_Local) or to get the differences between two versions (vGetDiff). The source XQuery code of some of these functions are included in the appendix A.

The architecture of the proposed versioning system, shown in the figure 2, is divided into three parts: Repository, that stores all versioned documents and logs data, the XQuery interface which allows to access to the set of functions defined in our system, and, the core of our system, which is our set of XQuery modules where the relationship among these modules are also illustrated.

The specification of the W3C has been fulfilled as far as possible to build our XQuery modules where currently only two features, included in the majority of XQuery engine, are required: support for attribute xml:id, used in our version-stamp technique to represent the historical information and the eval () function which evaluates a string dynamically from an XPath/XQuery expression.

The main advantages of this solution is its usability and portability since these modules can be integrated in any XQuery engine or XQuery-support XML native database; we have checked them in Saxon[2] and eXist[12]. In [23] we also proposed a versioning system based on a set of web-services. The main differences between both solutions are: 1) The web-services solution, which uses XSLT modules, were created to be consumed by third parties meanwhile with XQuery modules anybody can construct their own ver-

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:doc2vdoc($col, $doc, $alias, $time, $name_vDoc)</td>
<td>The original $doc XML document located in $col collection is transformed into an XML versioned document. $time represents when the version was made and $name_vDoc is the new name of the resulted vDocument ($name_vDoc).</td>
</tr>
<tr>
<td>Version:edoc2vdoc($vDoc_urs, $version)</td>
<td>A document snapshot from $vDoc_urs is made in order to retrieve the valid information from the given $version.</td>
</tr>
<tr>
<td>Version:getVersions($vDoc_Uri)</td>
<td>Retrieves a list of identifier’s versions from the $vDoc_Uri vXML document.</td>
</tr>
<tr>
<td>Version:vGetDiff($vDoc_Uri, $vDoc2_urs)</td>
<td>The XML differences between two XML documents are retrieved. The resulting file is retrieved either in our XML transactional format or in jXYDiff format.</td>
</tr>
<tr>
<td>Version:vUpdate($vDoc_Uri, $Xml_Uri, $dim_now, $alias)</td>
<td>It updates the version $dim_now of a vDocument $vDoc_Uri from an XML updated document ($Xml_Uri).</td>
</tr>
<tr>
<td>Version:vCheck()</td>
<td>Checks the integrity of a vDocument.</td>
</tr>
</tbody>
</table>
Extending an XNDB with version support

The previous XQuery modules can be used in multiple scenarios such as XML document office, XML configuration documents or XML graphics documents (SVG). However, from our point of view, one of the most remarkable scenario to include them is in an XML Native Database (XNDB) with XQuery support, in order to manage versions in a native way. Integrating them in an XNDB some important versioning features such as concurrency, locking, indexes or users security are supported directly by the XNDB.

eXist [12] has been chosen to be extended since it is an open source database management system that stores XML data according to the XML data model. Some of its features are: it supports many popular XML query languages such as XQuery, XPath, and XSLT, it has automatic indexing features, supports data update, it supports SOAP, XML-RPC, WebDav and REST protocols and it is compliant with the XQuery standard (99.4%).

One of the most important features of eXist is that the query engine is highly extensible, that is, both Java and XQuery can be used to achieve this aim. In our case both languages have been used for version management, since modules are programmed in XQuery scripts and to integrate the jXyDiff tool we had to develop a Java jXyDiff adapter.

Firstly, in order to integrate XQuery scripts, eXist allows us to import them in two ways: as an URI and as binary documents into the collection. The first solution was rejected in order to prevent our versioning system from possible problems such as the possibility of URI being inaccessible for shutdown services, network problems, etc. Therefore, the solution was to integrate them in eXist. In this second case, eXist stores the sources XQuery code in the database in a binary way and it is only necessary to specify in which collection the modules are in order to be used from queries.

Furthermore, as has been said, our versioning system uses a difference tool to obtain the changes. The main problem, we had to integrate it in eXist, was to adapt both document object model (DOM) since jXyDiff uses a model based on nodes meanwhile eXist uses sequences of items, thereby being necessary to develop a Java adapter for joining both data models in order to get the differences between XML documents. As a future work our idea is to develop an XQuery diff-XML tool due to suffers the main-memory Java-diff algorithms as shown in [17].

Therefore, to add our extension to eXist, the following steps are necessary: 1) To copy our XQuery modules in an eXist XNDB 2) To add the Java adapter code to eXist source (extension/modules/src) as well as the jXyDiff jar file in the lib directory 3) To include the namespace of our extension in eXist configuration file (conf.xml) and finally 4) To recompile eXist in order to use this extension in a native way.

4.3 Web Version-Admin Interface

Once our versioning system has been shown, the next step is to provide an easy, transparent and intuitive interface so that users can manage their versioned documents. Here, we explain how the Web Exist interface has been extended.

Our web-version interface is shown in figure 3 where a new menu called as Browse vCollections has been added for our extension. After logging in, depending on the type of the document (XML or vDocument) different actions can be done as shown in the action column. We have decided to use different graphical icons to identify the actions to be done in order not to overload excessively this interface. Thus, if an XML document is chosen, the following actions can be done: edition (yellow icon) or its transformation to a vDocument (blue one). This latter action uses the doc2vdoc operation from vConversion module. Notice that this process can also be performed during the document’s upload by means of checking the vUpload box from this interface.

In the case of an XML versioned document, several actions can be done as shown in figure 3. The first action, green color, allows us to get an XML version from a vDocument using the vConversion module. Yellow color is for editing a version of a vDocument allowing us either to update the vDocument from the changes made or saving this
XML document in order to keep on later. Finally the last
icon, red color, allows us to update a vDocument from an
updated XML document, that is, obtaining the differences
and execute them.

There is an special XML document called logs_changes.xml
in figure 3 that stores all changes made for a specific user
with the aim of restoring documents after a system crash or
tracking the changes made by a specific user. To implement
it, we have had to use some eXist XQuery functions.

4.4 Experimentation

In [23] we exposed some experiments to measure our so-
lution against the temporal one, similar to [27], using the
Saxon processor[2]. In this paper, we continue with these
tests, using, in this case, the eXist XML database in order
to evaluate its performance in our versioning system.

The testing machine is a Pentium Quad Core 2,4GHz PC
with Linux (Ubuntu) where eXist is set up with 512MB
memory and the retrieval time obtained refers to the ex-
ecution time in a Java-client application. We have used the
same XML versioned documents as in [23]. The tests have
been carried out on studies specifically dealing with lineal
versioning and branch versioning, where in the latter case,
different branched versioning can be chosen. In this case we
have chosen which version to modify between 20%, 50% and
80% of choosing a different version from the current one.
The experiments have also been carried out on documents of
different sizes (Small, Medium and Large) where different
approaches related to the number of versions and the number
of changes per version could be chosen (100-5, 60-10, 30-20)
(version - changes per version) from the ACM XML Sigmod
Record[3] thereby evaluating the behavior of our system in
different scenarios (more characteristics of these documents
are shown in [23]).

In this work, we have decided to analyze the most useful
queries in our versioning system: (Q1) is a snapshot docu-
ment query in XQuery, not evaluated in [23], which retrieves
its valid document from a given version (Appendix A-1) as
well as (Q2): Snapshot Element Query which retrieve those
authors’ elements that are valid in a given version (Appendix
A-3). Both queries need to check for each versioned element
if its associated version region contains the given version, be-
ing necessary to dereference the v:start and v:end attributes,
using the id() XPath function.

Figure 4.a shows the Q1 retrieval time obtained using an
XQuery timestamped solution (Tstamp), an XQuery ver-
sionstamp solution (VStamp), an XQuery versionstamp opti-
mized solution (Optim.) and an XSLT versionstamp solu-
tion(XSLT). As result, our solution, as in [23], behaves less
efficiently than the lineal timestamped solution as shown in
figure 4, since the time solution uses the operators < and
>= to verify if a time belongs to a time interval, meanwhile
in our process, since branched versioning is used, we have to
analyze which versions are included in a version region and
check if the requested version belongs to them. To do it, as
shown in Appendix A-3, we have to retrieve all descendant
identifiers for the v:start and v:end attributes (id()-XPath
function) in order to check if the given version belongs to
a version region. From this figure, we can sum up our so-
lution greatly depends on 1) the number of lineal versions
that the document has (the more lineal versions the docu-
ment has the more descendants it needs to retrieve. This
can be clearly seen in the difference between the retrieval
time in the lineal versioning and in any of the branched ver-
sions) 2) the size of the versioned document since it needs a
deeper dereferencing operation.

In order to improve this query, an optimized solution for
XSLT was designed in [23] which has been restructured for
XQuery. This optimization consists of storing their descen-
dants within each version, thereby not having to constantly
recover this information in each query. As we can see in
figure 4.a the optimized solution reduces considerably the
retrieval time, being almost the same as the timestamped
solution. In those cases where our solution had its poorest
performance (large documents or several versions) this time
has been reduced enormously (for example as is shown in
figure 4a where the L100/5 retrieval time has been reduced
from 39 seconds to 4 seconds using the optimized solution).
Notice that this solution is independent from the number of
versions since it is not necessary to retrieve the descendants
of the \texttt{v:start} and \texttt{v:end} attributes. This situation can also be seen in figure 4b.

Notice that the retrieval time in XQuery is quite higher than the XSLT solution because the XQuery snapshot algorithm is recursive and needs to process all nodes of the XML document, thereby generating huge output XML fragments. For this reason we have decided to use the XSLT solution when a document snapshot operation is required in order to retrieve better performance (query shown in Appendix A-4).

Finally, regarding the update operations performance, not studied in this work, we believe it is not going to be very high. As described in [23] the creation of a new version is defined by a set of update operations where these operations work specifically on a node in the XML document. In this way, this time will be composed by: the execution time to look for the node to be updated and the execution time of the \texttt{XUpdate} operation as in whatever \texttt{XUpdate} operation.

Although it can be argued that our solution performs less efficiently than the linear timestamped solution in spite of similar results in the optimized query, it offers many advantages that timestamped solutions cannot: 1) it is able to manage branched versioning that is not supported in timestamped ones and 2) versioned documents can be queried both on a version and a temporal level as shown in [23].

5. CONCLUSIONS AND FUTURE WORK

Document version management has been used for years mainly in collaborative environments by means of diff line-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 with XML standard query languages. The XML temporal document solutions, based on the timestamped technique, cannot manage branched versioning. To solve these problems we proposed a versionstamp technique in [22].

In this paper, we have taken another step forward to solve one of the most important problems in versioning systems derived from the necessity to set up software which in many cases is characterized by not being very portable to other platforms and in the case where it is portable its migrations could not be considered easy. In this way we have developed a set of XQuery modules which do not have portability restrictions and allow us not only to manage the different versions of an XML document but also to integrate them in a XML native database. Since our proposal is open, it can be used for third agents either to manage their documents or to extend them incorporating new features.

As future works we propose the following steps:
- To develop a diff-XML solution in XQuery, making our implementation more portable.
- To extend the \texttt{XUpdate} language to include versioning constructors.
- To apply our versionstamp technique to other XML markup languages as XSLT stylesheets, SVG graphics or even to XML office documents.
- To develop an indexing technique for our vDocument.

6. ACKNOWLEDGMENTS

This work has been financed by the Spanish CICYT project “TIN-2008-03063”.

7. REFERENCES


APPENDIX

A. VERSIONING FUNCTIONS

A.1 Version:vDoc2doc

This function retrieves a valid document for a given version ($version) from a vDocument ($uri) where the vConversion: vDoc2docSnapshot is used to do it. Firstly it checks if the current element is valid (line 5) and then, it obtains their valid attributes (line 7 and 8), its valid v:data element (line 9) and their descendant elements in a recursive way (line11).

Query1: Version:vDoc2doc Query

1. declare function Version:vDoc2doc($uri,$version){
2.  vConversion:vDoc2docSnapshot(doc($uri)/[*]/*[2]/[*],$version) {
3. 5. if (f:vBelongTo($ev,$v)) then (:if element is valid:)
6. 6. element (name($ev)) {
7. 7. for $x in $ev/v:attrib[@v:BelongsTo($x,$v)] (attribute:)
8. 8. return attribute ($x/$name) ($x/$value),
9. 9. $ev/v:data//v:BelongsTo($ev,$v)/*][text()$. (data:)
10. (Descendant-element recursively:)
11. 10. for $x in $ev/child::*[name(.)='v:data' and name(.)='v:attrib]
12. 11. return vConversion vDocSnapshot ($x, $v)
13. 12. |}
14. 13. else ()
15. 14. };

This query can be used in any id()XPath-support XQuery processor. To do it, it is only necessary to add our namespace version module in an XQuery query and later to call the required function. The Saxon’s called is shown below.


A.2 Version:vGetDiff

This operation works with the jXyDiff tool. Our Java adapter has a public method called Compare (line 2) that retrieves the differences between two XML documents. In this case, it is used to retrieve the XML differences between the V2 acm.xml version (vConversion:vDocSnapshot is necessary) and an XML document (acm.xml), which is an updated document of the aforementioned version.

Query2: vGetDiff Function in eXist

1. declare function Version:vGetDiff($uri,$uri2,$v){
2.  jXyDiff:compare(vConversion:vDocSnapshot 
3.  (doc($uri)/[*]/*[2],*)$, doc($uri2)/*)
4. 3. }

A.3 vStamp:vBelongTo

This operation checks if a version region contains a given version (used in Query 1). To do it, it is 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-self in the v:start attribute and 2) the given version is not among the descendants-self in all versions for v:end attribute. Therefore, we have used the id() XPath function to derefer the v:start and the v:end attributes and thereby retrieve its descendants. We have defined this version function calling it vBelongTo (line 1) to check (line 4) if the given version belongs to the v:start attribute only (line 2) and not to the v:end attribute (line 3).

Query3: Snapshot Element Query. V3 Valid authors.

1. declare function vStamp:vBelongTo($ev,$v) as xs:boolean
2. 2. let $start:=$ev/id($ev/@v:start)//*/@xml:id
3. 3. let $end:=$ev/id($ev/@v:end)//*/@xml:id
4. 4. return (($start=$v) and (not($end=$v))}
5. 5. }

A.4 Version:vDoc2doc in XSLT

The typical snapshot transformation in XSLT is shown in this section. By providing a snapshot version as the parameter (line 2) and using the id() XPath function, it will return all valid information from this version.

Query4: XSLT Document Snapshot

1. <xsl:stylesheet>
2. 2. <xsl:param name="version">V1</xsl:param>
3. 3. <xsl:template match="*[1]"><xsl:value-of select="normalize-space()"/>
4. 4. <xsl:if test="not(id(./@v:end)//*/@xml:id=$version) and (id(./v:start)//*/@xml:id=$version)">
5. 5. <xsl:element name="[name(.)]">
6. 6. <xsl:for-each select="v:attrib[@v:BelongsTo(.,$v)]/*/@xml:id=$version" and (id(./v:start)//*/@xml:id=$version)">
7. 7. <xsl:attribute name="[@name]"/>
8. 8. <xsl:value-of select="@value"/>
9. 9. </xsl:attribute>
10. 10. </xsl:for-each>
11. 11. </xsl:element>
12. 12. </xsl:if>
13. 13. <xsl:apply-templates select="*" except v:attrib,v:data/>
15. 15. </xsl:stylesheet>
16. 16. </xsl:template>
17. 17. <xsl:template match="*[1]">
18. 18. <xsl:apply-templates select="*[not(version)][not(version)="V3"]"/>
19. 19. </xsl:template>
20. 20. <xsl:template match="versioned_doc"/>
21. 21. <xsl:apply-templates select="*"/>
22. 22. </xsl:template>
23. 23. </xsl:stylesheet>