Report

A Time Machine for XML

Author(s):
Fourny, Ghislain; Florescu, Daniela; Kossmann, Donald; Zaharioudakis, Markos

Publication Date:
2011

Permanent Link:

https://doi.org/10.3929/ethz-a-007313560 →

A Time Machine for XML

Ghislain Fourny[1], Daniela Florescu[2], Donald Kossmann[1], Markos Zaharioudakis[2]

[1]ETH Zürich
Zürich, Switzerland
ghislain.fourny
donald.kossmann
@inf.ethz.ch

[2]Oracle
Redwood City, CA
dana.florescu
markos.zaharioudakis
@oracle.com

ABSTRACT

With sinking storage costs, it becomes more and more feasible, and popular, to retain past versions of documents and data. While undoing changes is worthy, this becomes even more valuable if the data is queryable. Nowadays, there are two widespread version control paradigms: document versioning (SVN, git, etc.) and versioned databases. The former handles any kind of document, even binary, but only sees lines of text, so that the query capability is limited. The latter provide fine-grained temporal query capabilities on highly structured data - but storing everything in a relational database is not desirable. The goal of this paper is to provide a unified framework for efficiently versioning, querying and updating not only data and documents, but also, inbetween, any kind of semi-structured information, like XML. We start with the XQuery programming language and meticulously extend its data model, its syntax and its processing model to make it seamlessly time-aware. We provide data structures and algorithms for the efficient implementation of such a versioning system. Finally, we show that there is no significant performance loss for traditional queries when enriching an existing engine with versioning capabilities.

Categories and Subject Descriptors

D.3 [**Software**]: Programming languages; H.4 [**Information Systems**]: Information Systems Applications

General Terms

Design, Languages, Performance, Standardization

Keywords

PUL, SVN, Transaction time, XML, XQuery, Versioning

1. INTRODUCTION

Over the last few decades, hardware trends have been going towards cheaper, denser storage. In addition, software allows compression, virtually increasing the amount of data stored. Several times, Jim Gray [23] [30] pointed out that with so much storage capacity available, it becomes possible to keep track of crucial information: instead of updating in place, a new version can be created.

Versioning information allows to go back in time, compare, analyze, and produce even more information. In addition, versioning can serve as a transaction control mechanism, allowing applications to undo or merge concurrent changes. Such features could be very useful, for example, for collaboration tools, which are proliferating. Google, Adobe and Microsoft are offering Web applications for word processing, spreadsheets, presentations, which allow users to view, share and edit documents in real time [1] [4] [6]. Applications like Microsoft Word already have basic versioning (reviewing) capabilities.

Some versioning systems are already widespread. On the one hand, there exist document versioning systems such as SVN [2], git [3] which analyze documents line by line, compare and compress them. On the other hand, there are also versioning systems for structured data, like versioned databases (Oracle Flashback [7], Microsoft ImmortalDB [24]).

There are two basic approaches to keeping track of versioned data. The first one is to use valid time [10][22][26][29], i.e., the time at which the data is valid (which can even be far away in the past, before computers existed at all). The second one, which we are taking in this paper, is transaction time, i.e., the time at which the data is written or committed.

The general contributions of our work are:

(i) to provide a unified model for data versioning, document versioning and anything (semi-structured data) between data and documents.

(ii) to have a programming language suited for processing (powerful querying as well as updating) this versioned data.

(iii) to perform measurements on a prototype that demonstrate that time travel can be implemented with reasonable storage and query time overhead.

We choose to base our work on XML technologies and extend XQuery [12], which is a Turing-complete programming language standardized by the W3C.

What distinguishes our approach from existing literature is the combination of two factors. First, time is often modeled as an explicit parameter in an application or document. Extra code is needed to explicitly refer to time in the queries, which can lead to poor performance. Rather, we choose to add time to the logical data model [20] and hide physical

Technical Report, ETH Zurich, January 2011.

implementation and storage details from the programmer, so that it is simple to go back in time and query any, or even several versions. Also, we ensure backward compatibility, allowing the user to completely ignore the versioning features. Second, our data and query model seamlessly integrates with the updating model: instead of collecting snapshots over time, the database is constructed and updated using the XQuery Update Facility [14], which has the nice side effect of providing us with the deltas as Pending Update Lists and of unambiguously keeping track of node identities.

The remainder of this paper is organized as follows: in Section 2, we present a motivating example. Section 3 introduces our extension of the XQuery data model to obtain an XML-aware versioning system. Section 4 introduces our extension of the XQuery programming language with new functions and time axes, so that it is possible to query against versioned data. In Section 5, we give an overview of the processing model for checking out from and checking in to the repository, to allow collaborative work. Section 6 presents data structures to implement the versioning system together with an high-level overview of querying and updating algorithms, as well as a storage scheme based on existing literature. Section 7 presents the performance measurements showing that extending a non-versioning engine with versioning capabilities does not alter performance on traditional queries. Section 8 discusses related work.

2. MOTIVATING EXAMPLE

The running example for this paper is the following: an investment bank maintains data about stocks, bonds, funds, and portfolios thereof in a database. Users can collaboratively modify the data. This is semi-structured information (for example, it has optional comments and nested elements). The bank would like to set up a versioning system for its data.

Initially, the data could look as shown in Fig. 1(a). There are three sections: users, securities and portfolios. The users are Alice, Bob, and Charles. The securities part contains Apple stock (Nasdaq Symbol AAPL), a bond of the Swiss Confederation (Valor 3141) and a fund managed by Charles (ISIN CH123456789) investing in the former two assets. Bob has a portfolio containing bonds and shares in Charles' fund. Fig. 1(b) shows a subsequent database state after some modifications: Alice changed her hometown, the fund's content was updated, Bob updated his data to use references (Symbol, Valor, ISIN) to the underlying securities, and Alice created her portfolio with AT&T stocks.

While keeping all the functionality already available to query and update the current version, the bank would like to be able to keep track of and analyze past data as well. For example, it could be interested in the evolution of the diversification of a portfolio, or in checking that a balanced fund has always allocated the right stock/bond ratio over time, or in comparing the performance of a portfolio between two versions (using an external source to retrieve the latest pricings). Versioning functionality is needed. We emphasize that not only versioning data is important - reasons are widespread in literature [29], [31] - but also the ability to flexibly query against past versions.

Versioning it as a document, e.g., in SVN (diffing lines of text) would miss the continuity of the tree structure: the identity of the nodes is lost between the versions, drastically limiting the querying functionality. A versioned database

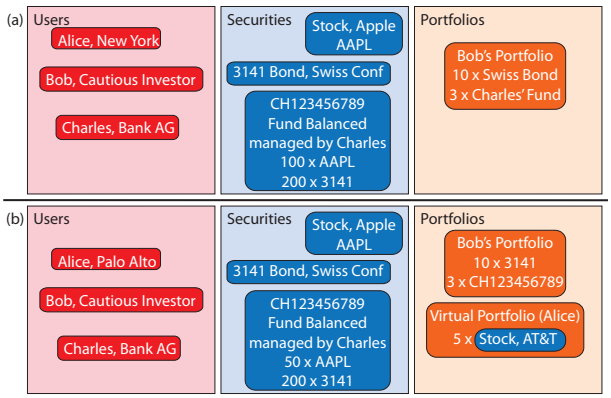

Figure 1: Two versions of the data (a) and (b).

would not be of great help either, as this is semi-structured information. For example, the users do not provide the same information (hometown, company, or even comments). The keys used to identify securities are not using the same standard (Nasdaq Symbol, Swiss Valor, ISIN). Then, there are levels of nesting (like the AT&T stocks nested in Alice's portfolio). Finally, since there is no schema, the evolution of the data structure is unpredictable. For all these reasons, versioning semi-structured data cannot be reduced to a versioned database with uniform rows and colums in a straightforward way.

There are already many technologies around for storing and manipulating semi-structured information: this data can be stored as XML (e.g., a large XML document, or as three collections of XML trees) or JSON. XML can be queried with XPath, XQuery, XSLT. These three languages are all based on the same data model (XDM) [20]. The XQuery language itself is made of three main parts:

- A side effect-free core (W3C recommendation) [12], which reads XML documents and outputs an XDM instance.
- The XQuery Update Facility (W3C candidate recommendation) [14] which reads XML documents and outputs a list of changes, standardized as a Pending Update List (PUL), that can subsequently be applied to these XML documents.
- The XQuery Scripting Extension (W3C working draft) [13] which reads from and writes to XML documents, applying any number of PULs.

There is also literature about storing versioned, semi-structured information [16] [17] or query it [26] [19]. We suggest to leverage these technologies and techniques and provide a unified framework which is (1) XML-aware to handle any kind of data. The versioning system should be aware of XML trees and nodes, which can be done by extending the XDM. (2) XQuery-aware for powerful queries and updates. It should be possible to easily navigate to past versions with XQuery code, which can be done by extending the XQuery syntax. It should be possible as well to create new versions with XQuery Update (which outputs a PUL) or XQuery Scripting (which produces a sequence of PULs). We showed in [21] that a sequence of PULs can be composed to a single PUL, so that a delta can always be represented by a PUL. Serializing these PULs as XML can even allow querying the deltas.

A typical setting to start with for the implementation would be to use an XQuery Web Application Server like

MarkLogic [5] or Sausalito [8]. The latter runs on and stores its XML data in the Amazon cloud. We chose to extend Sausalito with versioning capabilities.

3. EXTENSIONS TO THE DATA MODEL

Our broader aim is bridging the gap between data and document versioning. Because we want to query versioned semi-structured data, we need to work at the level of the data model. For XML- and XQuery-aware versioning, we need to extend the XQuery Data Model (XDM) [20] with a time dimension.

3.1 Tree timelines

We introduce the concepts of *node timeline*, *tree timeline* and *version*. In the context of a Web application, we are interested in linear versioning. Branching can be emulated by setting up a tree of repositories, and merging can be done with the processing model defined in Section 5.

A *node timeline* is a succession of all node items sharing a given identity. It models the lifetime of a node item on which updates are applied. A node timeline is identified by a URI.

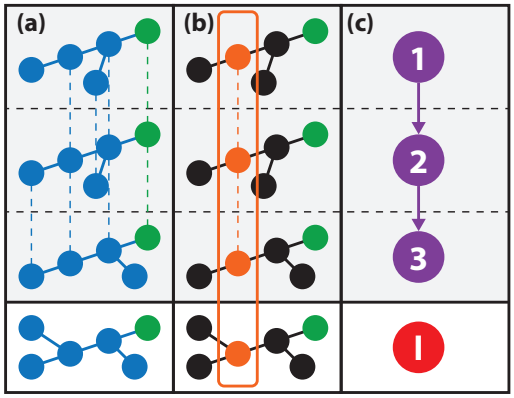

Figure 2: A tree timeline (a), a node timeline (b), a sequence of versions (c).

A *tree timeline* is a succession of trees whose roots share a given identity. It models the lifetime of an XML tree on which updates are applied. All roots of these trees belong to the same node timeline, which is called the root node timeline of the tree timeline. A tree timeline is identified by the URI of its root node timeline.

A *version* uniquely identifies a tree among a tree timeline. Each version has a number. There is also a special version with no number, called the local version. This is the version which is currently being modified by the user and which has not been committed yet. Each user can have her own local version. A version is identified by a URI.

Fig. 2 shows how tree timelines, node timelines and versions are related. The version shown at the bottom in red is the local version.

Node items are defined as in the original XQuery Data Model. Two new accessors are defined on them:

- `dm:node-timeline` returns the URI of the node timeline this node item belongs to
- `dm:version` returns the URI of the version that uniquely identifies this node item within its node timeline.

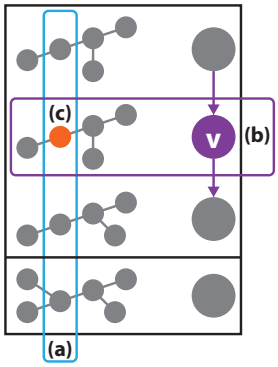

Figure 3: A node timeline URI (a) together with a version URI (b) uniquely identify a node item (c).

The node timeline URI and the version URI uniquely identify a node item (Fig. 3). The other accessors (dm:children, dm:attributes, dm:node-name, ...) are defined as stated in the XDM specification. Document order is extended to allow time-awareness: node items are first sorted in time, and then in space. Newly created nodes (with element constructors or nodes copied in updating expressions) belong to no timeline and are in no version, so that these accessors are reserved.

In the remainder of this paper, when we talk about "a version of a node (or tree) timeline", we actually take a shortcut and refer to the node item (tree) uniquely identified by this version and the node (tree) timeline. This should be clear from the context.

3.2 Collection timelines

In some XQuery implementations like Sausalito, collections are first-class citizens (for example, they can be updated with extra update primitives). Thus we also extend the corresponding data model.

A *collection timeline* is a succession of collections of node items. It models the lifetime of a collection to which nodes are inserted or deleted. Collections are defined in the XQuery specification. Versions can also uniquely identify a collection in a collection timeline (Fig. 4).

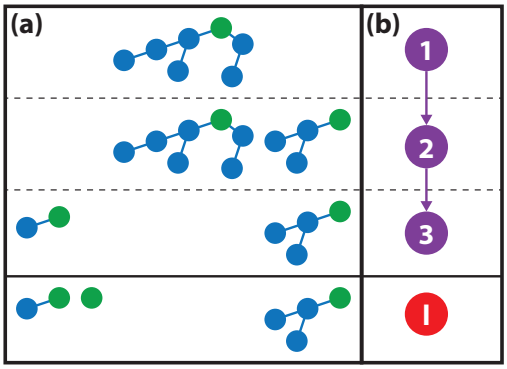

Figure 4: A collection timeline (a), and the corresponding versions (b).

We restrict collection timelines to two different kinds: (i) collection timelines which only contain non-local (frozen)

versions of the node timelines, and (ii) collections timelines which only contain local versions of the node timelines. Collections of the second kind "follow" the evolution of the node timeline.

The implementation of collections differs between XQuery engines. In Sausalito, collections and trees are tied together: collections own their trees on the storage layer, i.e., a collection points to the roots of its trees, and a tree can only belong to a single collection [11]. To allow for this entanglement, we can synchronize tree and collection (type (i)) version numbers, which means that the versioning granularity is at the collection level ("synchronized collection timeline").

The framework for synchronized collection timelines can very often be deduced from the framework for tree timelines as such a collection can be represented as a forest of trees (equivalently, a tree without its root). The framework for other collection timelines (type (ii), or unsynchronized (i)) is very similar to document-oriented versioning as each slice is a sequence of identifiers.

3.3 Serialized Pending Update Lists

The XQuery Update standard [14] defines how to update an XML document. The updates themselves are organized as a set of update primitives (of the kinds insert, delete, replace, rename) standardized as a Pending Update List (PUL). For example, it is possible to delete Bob's portfolio like so:

```
delete node //Portfolio[@owner="Bob"]
```

This XQuery Update program outputs a PUL containing, in this case, one update primitive `upd:delete($ref)` with a reference `$ref` to the node to delete (the one computed by the XPath expression above). Such a PUL can then be applied to the XML document, which deletes the node.

An updating XQuery program always returns a PUL. XQuery Scripting is an extension of XQuery Update which allows to sequentially apply several PULs - however, as explained in Section 2, this can always be reduced to a single PUL. PULs are hence ideal candidates for deltas in an XQuery-based versioning system, as shown on Fig. 5.

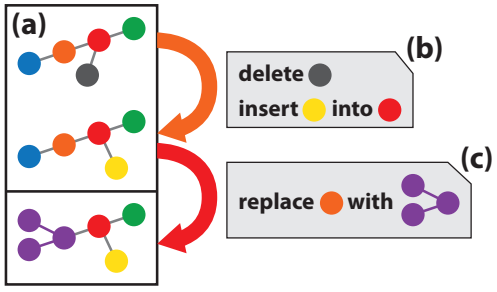

Figure 5: Three versions (one being local) of a tree timeline (a), and serialized PULs modeling the deltas (b).

In order for these deltas to be queryable, we choose to serialize the PULs to XML. For example, the PUL output by the program above can be serialized as:

```
<pending-update-list xmlns="http://www.example.com/spul">
    <delete>
        <target>http://www.example.com/Doc1#1.3.3</target>
    </delete>
</pending-update-list>
```

Technically, a *serialized Pending Update List* is a Pending Update List (as defined in [14]) in which the target is replaced with the URI of its node timeline and the content is an XDM serialization. It is beyond the scope of this paper to specify an XDM serialization model. Serialized Pending Update Lists, since they are XML, are queryable with XQuery.

4. EXTENSIONS TO THE PROGRAMMING MODEL

4.1 Tree timelines

To navigate through a tree, i.e., within a version of a tree timeline, the user can use the various axes defined in XQuery (`child::`, `descendant-or-self::`, ...). To navigate in time, the following functions are added to the XQuery functions and operators [25] and are available to the user (they are defined in a new versioning namespace, represented with the *vng* prefix):

- `vng:reference` which takes a node and returns its node timeline URI (using dm:node-timeline).
- `vng:version` which takes a node and returns its version URI (using dm:version).
- `vng:dereference` which takes a node timeline URI and returns its local version.
- `vng:ttdereference` which takes a node timeline URI, a version URI and returns the associated node item.
- `vng:node-versions` which takes a node and returns a list of all version URIs of its node timeline.
- `vng:version-number` which takes a node and returns its version number.
- `vng:time` which takes a node and returns the creation date and time of its tree (when it was committed).
- `vng:is-local` which takes a node and returns true() if it is local, false() otherwise.

For example:

`vng:dereference(vng:reference($node))`
gets the local version of node $node.

```
let $ref := vng:reference($node)
for $version in vng:node-versions($node)
return vng:ttdereference($ref, $version)
```

builds a sequence of all versions of node $node.

```
for $n in $nodes
where vng:is-local($n)
return $n
```

filters nodes which are local within a sequence $nodes.

Because using these functions remains tedious for fast time travel, we also introduce the following axes:

`first::` (the first version)
`earlier::` (the former version)
`past::` (all past versions)
`past-or-current::` (the same, plus the current one)
`last::` (the last version)
`later::` (the next version)
`future::` (all future versions)
`future-or-current::` (the same, plus the current one)
`current::` (the current version)
`all-times::` (all versions)
`local::` (the local version)

For example, `later::node()` navigates to the next version of the node timeline to which the current node belongs

or the empty sequence if it does not exist. For example,
`last::node()/later::node()`
will always return the empty sequence, as well as
`first::node()/earlier::node()`.

Time axes are compatible with node tests:
`<time axis>::<node test>`
is defined as
`<time axis>::node()/self::<node test>`.

For example
`future::mickey`
is defined as
`future::node()/self::mickey`
and looks for all newer versions of the current node whose names are "mickey".

Also,

```
if(vng:reference($temperature-left)
       = vng:reference($temperature-right))
then
    fn:avg(
       $temperature-left/future-or-current::*
       intersect
       $temperature-right/past-or-current::*
    )
else ()
```

checks whether the temperature nodes `$temperature-left` and `$temperature-right` belong to the same node timeline and if such is the case, it computes the average temperature between them.

Back to our motivating example in Section 2, the following query checks that balanced funds correctly invested their funds (according to `is-compliant`):

```
every $fund in $doc//all-times::Fund
satisfies ($fund/Type != "balanced"
           or is-compliant($fund))
```

The following query looks for all versions of Bob's portfolio which, today, would be worth more than the last version (`value` gives access to the latest quotes):

```
let $current-portfolio
       ::= $doc//Portfolio[owner = "Bob"]
for $portfolio in $current-portfolio/past::*
where value($portfolio)
       > value($current-portfolio)
return vng:version($portfolio)
```

4.2 Collection timelines

To navigate through time in collection timelines, the following functions are available to the user:

- `fn:collection` which takes a collection timeline URI and returns a sequence of nodes (its local version)
- `vng:collection-versions` which takes a collection timeline URI and returns a list of all its version URIs
- `vng:ttcollection` which takes a collection timeline URI and a version URI and returns the corresponding collection (sequence of nodes).

4.3 Retrieving deltas

We define the function `vng:pul` which takes a tree or collection timeline URI and two version URIs, and returns the corresponding delta (serialized PUL). This query counts how many security-related nodes have been deleted so far:

```
let $ref := $doc/vng:reference(Securities)
let $first := $doc/vng:version(first::Securities)
let $last := $doc/vng:version(last::Securities)
let $pul := vng:pul($ref, $first, $last)
return count($pul//*:delete/*:target)
```

5. EXTENSIONS TO THE PROCESSING MODEL

One of the requirements for our framework is flexibility, which should also be true for collaborative work. We need flexibility to embed different collaboration models, for example: (a) database transactions, where queries are executed in an isolated way. An error is thrown if there are any concurrent changes. (b) document editing, where it is always attempted to merge changes into the repository even if other users modified the document.

In order to allow for several users to edit a document or data with different collaboration models, we need to extend the processing model of XQuery.

5.1 Checkout and checkin

We rely on a classical server-clients architecture. The server maintains a central repository with all past versions. Each client can set up a local version of a tree timeline by checking out the tree timeline from the server repository. It is only allowed to manipulate this local version (storing non-committed yet modifications as a local PUL). The nodes in the local version are only visible to the client.

The local modifications can be committed by checking in. New versions are appended to the repository only upon a successful checkin. Checkins and checkouts are atomic and isolated operations: there can only be one checkout or checkin at a time. This architecture is illustrated on Fig. 6.

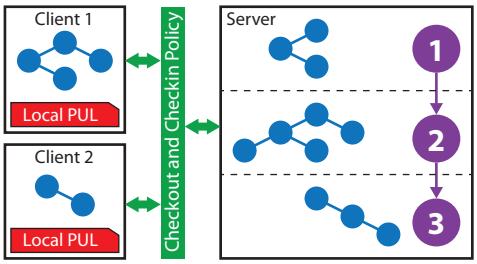

Figure 6: The checkout-checkin processing model

To make checkouts and checkins as flexible and configurable as possible, we use checkout and checkin policies. A client chooses a policy to decide how it wants to communicate with the server (read/write access to the repository, concurrent changes allowed or not, ...).

Both checkouts and checkins can be done implicitly, or explicitly with a function call. An implicit checkout occurs whenever it is attempted to access the local version of a tree timeline that has not been checked out yet. An implicit checkin occurs at the end of the program for all tree timelines which were checked out and modified during the program.

5.2 Checkout policy

A checkout policy can be seen as a blackbox which checks out a version of a tree timeline so that the user can modify

it. As one could have already checked out and made modifications before, the framework must allow for merging this version of the tree timeline with an existing local version.

Its input parameters are (i) the URI of the tree timeline to check out, (ii) the URI of the version to check out and (iii) the PUL containing local changes (empty if it is the first checkout or if there are no local changes).

It outputs (i) the local version of the tree timeline which has just been checked out, (ii) the updated PUL containing local changes, (iii) a boolean indicating whether it has been successful. If this boolean is false, then the local version and the local PUL are reverted and (iv) possibly an XDM instance containing unsolved conflicts if the checkout failed.

In addition, it saves the URI of the version which has been checked out in the dynamic context so that the checkin policy knows on what version the local PUL is based.

5.3 Checkin policy

A checkin policy can be seen as a blackbox which attempts to check in the modifications done on the local version.

Its input parameters are (i) the URI of the tree timeline to check in and (ii) the PUL containing local changes. In addition, it reads the URI of the version which has been checked out from the dynamic context.

It outputs (i) the PUL which is to be applied to the last version in the repository, (ii) a boolean indicating whether it has been successful. If this boolean is true, then the local version and the local PUL are erased and the version URI is removed from the dynamic context. (iii) possibly an XDM instance containing unsolved conflicts if the checkin failed.

If the checkin succeeds, the output PUL is applied to the last version of the tree timeline to create a new version.

5.4 Using Checkout and Checkin policies

The reason why our framework allows checkout and checkin policies is to allow each user to choose her collaboration model. The user is provided with several checkout and checkin policies.

There are several possibilities for a checkout policy. Here are four examples of checkout policies:

- no-checkout: always throw an error (no checkouts possible)
- single-checkout: copy the version of the tree timeline to the local version when it is implicit, and subsequently return errors (only one checkout possible)
- merge: always copy the version of the tree timeline to the local version and then merge any former local changes.
- discard: always copy the version of the tree timeline to the local version and then discard local changes.

Also, there are several possibilities for a checkin policy, like for example:

- no-checkin: always throw an error (no write access to the repository)
- conservative: check whether the version which has been checked out is (still) the last version in the repository and output the local PUL if such is the case. Otherwise throw an error.
- merge: in any case attempt to merge local changes to changes between the version which has been checked out and the last version in the repository. Throw an error if conflicts arise so that the user solves them.
- discard: discard any changes done in the repository since the last check out and output a merge of a reverse PUL and of the local PUL.

Choosing which policy to use is done at the application level, rather than at the repository level, i.e., different users of the same repository may use different policies. Each policy has a URI and choosing a policy is done in the prolog, exactly like collations for string comparison.

For example, for a database transaction mode, one could use:

```
declare checkout policy
    "http://www.example.com/single-checkout";
declare checkin policy
    "http://www.example.com/conservative";
```

where only one checkout is made, and an error is thrown whenever other users concurrently wrote to the repository.

And for document editing:

```
declare checkout policy
    "http://www.example.com/merge";
declare checkin policy
    "http://www.example.com/merge";
```

where it is always attempted to merge any changes.

6. ALGORITHMS AND DATA STRUCTURES

The last three parts gave a logical view on the expected behavior of tree timelines for an XQuery program. The contribution of this part is threefold:

- introduce a new data structure, called π-tree, which efficiently implements tree timelines
- give algorithms (i) defining how versions of this tree timeline can be retrieved and (ii) defining how this data structure is modified when a new version is produced.
- reusing and completing an existing storage schema to allow for collection versioning.

6.1 Pi-Nodes, Pi-Trees and Pi-Forests

Keeping each version as whole tree in memory would be very expensive. Many document versioning systems (SCCS, RCS, CVS, SVN...) store reverse or forward deltas to reconstruct past versions. The thought process explaining why we are not doing this is as follows. Each XML data slice can be represented as a tree. With the XQuery Update model [14], over time, the parent of a node never changes. A node can be connected, or disconnected, only once: at no two points in time it has two distinct parents (XQuery Update does not support moving nodes around). For example, on Fig. 7(a), in any version, node β is always a child of node α, γ and δ of β, ϵ of α and ϕ of ϵ.

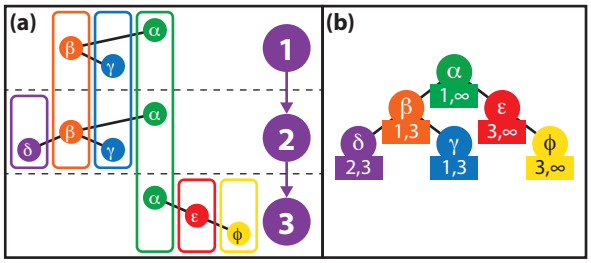

Figure 7: Three versions of a tree timeline (a) and its implementation as a π-tree (b)

Hence, for each tree timeline, its node timelines are organized in a tree in a natural way as shown on Fig. 7(b). It is this materialization of the tree timeline as a tree of node

timelines, called π-tree, which we use for our implementation. From now on, we focus on element nodes, but it is straightforward to extend to other kinds of nodes.

A node timeline is implemented as a π-node (Fig. 8(a)), which has an identity λ (here ORDPATH 1.3), a creation time c (here 2) and a deletion time d (here 5), a mapping n of the versions to the successive names of the node ("a" for 2 and 3 and "b" for 4). Creation and deletion times are positive integers, possibly infinite. A π-tree is then a tree of π-nodes. We call a (possibly empty) sequence of π-trees (Fig. 8(b)) a π-forest. Note that π-forests can be used to implement synchronized collection timelines (Section 3.2).

For convenience, the algorithms are described recursively on π-trees and π-forests, with the following definition: a π-tree (Fig. 8(c)) is obtained by attaching a π-forest below a π-node.

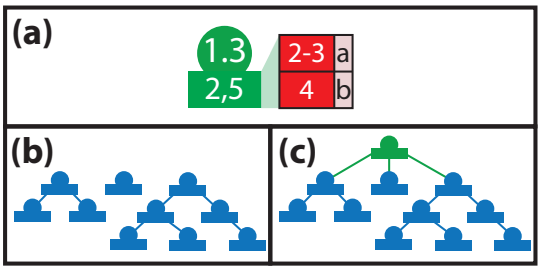

Figure 8: A π-node (a), a π-forest (b), a π-tree (c).

Retrieving a version (slice) v from the π-forest data structure is done with the instantiation function. This function is defined recursively as follows: the instantiation of a π-node (λ, c, d, n) is the node $(\lambda, n(v))$ if $c \leq v < d$, is an empty forest otherwise. The instantiation of a π-forest is the sequence of the instantiations of its π-trees, and the instantiation of a π-tree is done by instantiating its root π-node and, if the result is not empty, recursively the underlying π-forest.

Using this algorithm, the three versions on Fig. 7(a) can be reconstructed from Fig. 7(b). The algorithm works in $O(n)$ where n is the size of the π-forest. This grows linearly with the number of versions, but it is possible to reduce this time to the average size of a slice (up to a constant factor) by using clustering, as shown in Section 6.4.

6.2 Updating a Pi-Forest

From a logical point of view, it is straight-forward to compute a new version for a tree timeline: the last version is copied, the serialized PUL is deserialized to a PUL with targets living in this copy, and the PUL is applied. The resulting tree is the new version. From a physical point of view, the π-forest needs to be updated. We defined an algorithm to apply updates (contained in a PUL) to our π-forest data structure at time t (assuming all creation and deletion times in the π-forest are lower than t). This algorithm is such that, at the logical level:

- it does not modify previous versions ($u < t$)
- it creates a new version t which differs from version $t-1$ exactly according to the PUL.

An illustration is given in Fig. 9, where the new π-tree instantiates to an additional version obtained by applying the PUL to the former version.

The algorithm is defined as follows. Applying a PUL is done:

- for each deleting update primitive: by updating the deletion time of the target,
- for each renaming update primitive: by updating the time-to-name mapping,
- for each inserting update primitive: by inserting all contents at the corresponding locations (initializing the creation time to t and the deletion time to $+\infty$)
- for each replacing (content-replacing) update primitive: by inserting the content after (or as last child of) the target and updating the deletion time of the target (or of its children).

The complexity of the algorithm is $O(sd)$ where s is the size of the PUL and d the maximum depth of a tree, assuming a node can be reached in $O(d)$ with a reference.

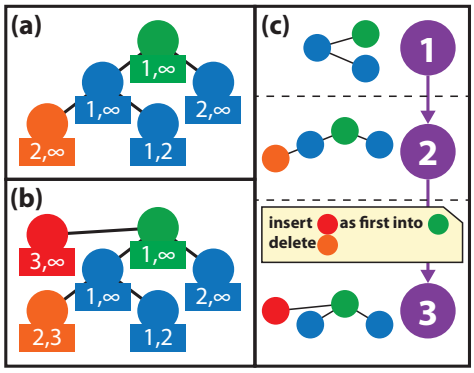

Figure 9: The older π-tree (a) can be instantiated to the first two versions in (c). The updated π-tree (b) can be instantiated to the three versions.

6.3 PUL retrieval

PUL retrieval may be implemented in two ways: (1) The PULs are stored besides each π-tree, and PUL composition is used to compute deltas. (2) No PUL is stored and deltas are computed automatically from the π-tree, as shown in existing literature, e.g. [15]. (1) and (2) might give different results, so the choice must be documented to the user.

6.4 Tree and Collection Timeline Clustering

Over time, as the data structure gets updated, its size grows. Retrieving a given version will hence take increasingly long times, as the entire data structure needs to be loaded.

This problem was solved by Chien, Tsotras and Zaniolo in [16] with a technique called Usefulness-Based Copy Control (UBCC). With this technique, timestamped XML elements (in our case, these would be π-nodes) are organized in constant-size pages. Each of them is identified with Sparse Preorder and Range (SPaR) [17] (in our case, we can use ORDPATH embedded in our URIs). Hence, using a snapshot index [28] to get all relevant pages, it is possible to reconstruct any version.

UBCC's main idea is that if the usefulness (i.e., the percentage of space which is relevant for the current version) of a page sinks below a treshold, the items of this page which are relevant for the current version are copied to a new page and the page is invalidated for the current version. Using this scheme allows version reconstruction in a time which is linear in the size of the target version.

If versioning is done at the collection level (see Section 3.2), we need to support synchronized collection timeline storage (as opposed to tree timeline storage originally). A synchronized collection timeline can be regarded as a "bigger tree" (with a virtual root), so that the same storage and reconstruction model as in UBCC can apply.

In the illustration on Fig. 10, we use ORDPATH identifiers [27], used only once within a collection. To generalize ORDPATH to a collection, we also use the ordering mechanism (1, 2.1, 3, ...) for the root of each tree. These identifiers allow to unambiguously reconstruct each π-tree, as well as the entire collection π-forest.

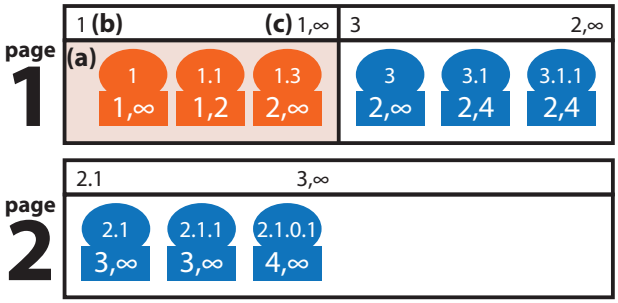

Figure 10: Two sample UBCC pages containing three π-trees, like (a). A π-tree has a root ORDPATH (b) and an overall timestamp (c).

However we suggest to introduce a "single-page" mode for large collections with small tree timelines, which reduces the number of pages used when retrieving small parts of such collections:

- The node timelines of a tree timeline which is smaller than a page are stored on the same page, in one piece, while collection timelines can be spread over several pages. New node timelines are inserted in place instead of being appended to the current page, which maintains the single-page consistency.
- The copy operation in UBCC (i.e., copying nodes that are still alive to a new page) is performed not only upon lack of usefulness, but also when a page becomes full because of in-place updates. Technically, this splits the corresponding tree timelines into an older, no longer updated, and a newer, fresh representation (see Fig. 11).
- For efficiency, each tree timeline stored on a single page can be assigned a timestamp interval (the smallest intervall containing all intervals of its node timelines). That way, the snapshot index technique can be applied in a coarser way, with fewer intervals.
- If the current version of a tree timeline can no longer fit on a single page, we revert to the original UBCC model by relaxing the single-page constraint.

7. PERFORMANCE MEASUREMENTS

We performed measurements to check that adding versioning features to our engine did not lead to a significant loss of performance for traditional queries. We built a prototype based on Sausalito [8], implementing the π-tree data structure and the instantiation and update algorithms, not using clustering. Our measurements compare the existing, non-versioning Sausalito with the versioning-enabled prototype.

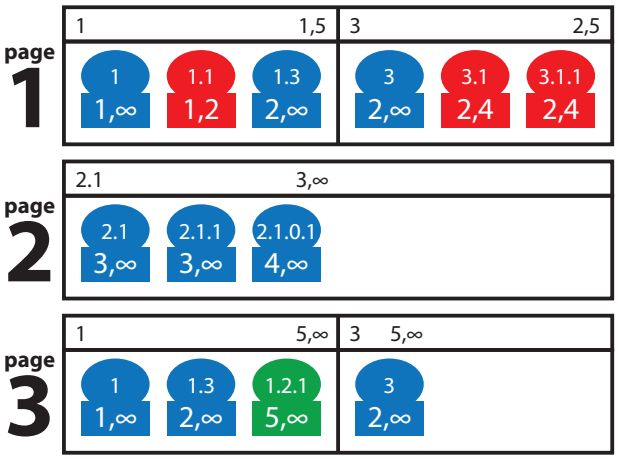

Figure 11: Page 1 is full. To insert the green π-node 1.2.1, the current (not red) π-nodes 1, 1.3 and 3 need to be copied to a new page.

7.1 Query classification

For convenience, we group queries into three kinds:

- Traditional queries do not use versioning facilities (time axes, versioning functions), are automatically reading the local, checked out version and potentially creating a new version. They can be classical (XQuery core), updating or scripting queries.
- Time travel queries correspond to classical queries that are executed on a given past version.
- Spacetime queries are the most general queries and can travel back to any version. They can be classical, updating or scripting (in the latter case they may perform explicit checkins or checkouts).

7.2 Measurement settings

We chose to base our repository on XMark. XMark is a benchmark suite providing automated generation of auction documents of any size, as well as a DTD against which these documents are valid and 20 sample queries.

We adapted this benchmark as follows: we designed updating queries which read an XMark document and simulate a construction scenario thereof in 25 steps. This means that we obtain a repository of up to 25 versions against which we can perform our measurements. We can reuse the original 20 XMark reading queries either as they are, as traditional queries or adapt them as time travel queries.

The first part of the measurements is done in the generation phase. We generated our 25 versions on the traditional engine and obtained the times $g(\text{vers}, v)_{v=1..25}$. We also measured the size of each intermediate version $s(v)_{v=1..25}$. Then we generated our 25 versions in the versioning engine and obtained the times $g(\text{vers}, v)_{v=1..25}$. We measured the intermediate sizes of the entire repository $r(v)_{v=1..25}$.

The second part is done in the query execution phase. We built a three-dimensional pivot table $t(n, v, q)$ where n is the total number of versions in the repository ($n = 0$ for the traditional engine), v is the version queried ($v = 0$ if we query the local version on the versioning engine), and q the XMark query being executed (between 1 and 20, or 0 for a trivial display of the version) is the query being executed.

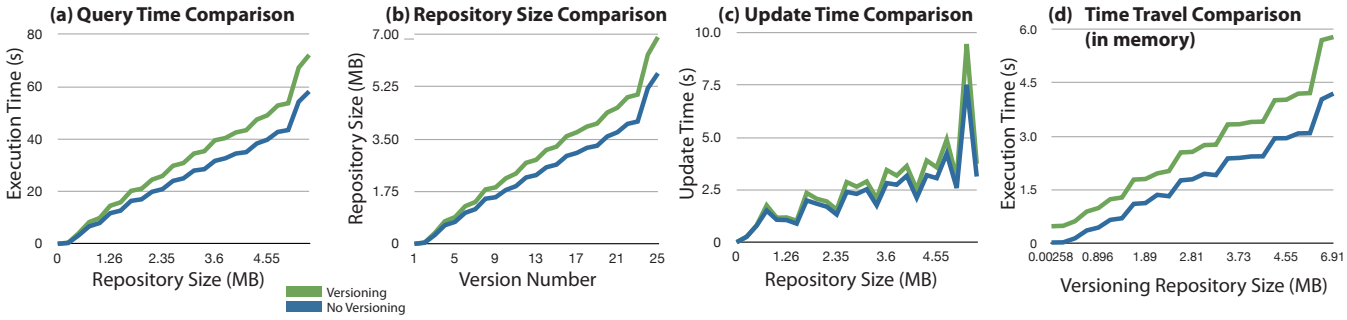

Figure 12: The three goals: query time (a), repository size (b), update time (c) in the same complexity class with versioning as without versioning. (d) shows the time travel performance.

$t(n, v, q)$ is the corresponding execution time.

7.3 Design goals

Before implementing a prototype, we formulated three design goals aiming at almost no loss of performance for traditional queries (with "about the same", we mean in the same complexity class, with a constant factor as small as possible):

1. Traditional queries should be executed on a versioning engine in about the same time as on a traditional engine: $\sum_q t(n, n, q) \sim \sum_q t(0, n, q)$ for n = 1..25
2. The physical repository size of a versioning engine should be about the same than that of a traditional engine: $r(n) \sim s(n)$ for n = 1..25
3. Traditional updating queries should be executed in about the same time as with a traditional engine: $g(\text{vers}, n) \sim g(\text{trad}, n)$ for n = 2..25

7.4 Results

We are showing measurements obtained with an XMark document size of 1MB (scale factor 0.01) on Fig. 12. The three goals are achieved, as the complexity classes are equivalent (the constant factor rises by 20 to 30 %). Fig. 12(d) also shows how the performance in memory of time travel to each version (with a 25-version repository) compares with accessing the corresponding documents directly on non-versioning Sausalito. Since we did the measurements without clustering (i.e., one single cluster), the loading time is constant and irrelevant. Clustering would only become meaningful with more versions, where existing literature shows that performance scales [16])

8. RELATED WORK

Example of versioned database technologies are Oracle Flashback [7] and Microsoft Immortal DB [24], which allow non-destructive querying of the past versions of the database.

Zholudev and Kohlbase developed TNTBase [31] which is a versioning system for versioning XML documents on top of an SVN respository, linked to an XML database. TNTBase seems to focus on document-oriented versioning, e.g., deltas are regular SVN deltas, whereas we attempt to bridge the gap between documents and data.

There are non-XML proposals for versioning semi-structured data. Combi et al. [18] introduce a data model and an SQL-like query language.

There are also other proposals for extending XML data models. Many approaches work with valid time intervals and introduce it explicitly in XML documents, so that queries have to be aware of the syntax chosen (i.e., do filtering with valid-time attributes). For example, Amagasa et al. [10] as well as Mendelzon et al. [26] extend the XPath data model with valid-time intervals. Dengfeng and Snodgrass [22] extend the XQuery language, also with valid-time intervals, and XQuery queries either need to be aware of the implementation (representational queries) or need to be transformed to a corresponding representational query. Fusheng and Zaniolo [29] suggest an XML versioning system which does not tamper with the data model, but here also, an explicit valid-time parameter is introduced in XML documents. Furthermore, the latter approach requires that the document is valid against a DTD: the DTD for the corresponding time-aware XML document is algorithmically deduced from the DTD of the original document. In our versioning system, XML data needs not be valid, as we are willing to leverage the fact that semi-structured data can exist without a schema.

Dyreson [19], like us, focuses on transaction time and extends XPath with time axes. However, his approach is based on the concept of an *observant system*, which has access to documents as well as reading and modifying times, but cannot update them. Our approach, on the other hand, follows very closely the evolution of XML data from the inside because it is seamlessly integrated with the XQuery update facility, which is the language to update XML data. Furthermore, our deltas are serialized XQuery Update's Pending Update Lists. Dyreson also suggests time axes, but the approaches are different: he defines the concept of known or assumed status (since the observer is outside), uses transaction time literals and introduces new node tests. We use the concept of version number and chose a more concise approach (all of the time travel information is in the axes).

Shu-Yao, Tsotras, Zaniolo and Donghui [16], [17], [28] provide an XML Document versioning storage scheme. They store the document as a set of nodes, clustered on several pages. Usefulness-based copying is used for efficiency, and each version of the document can be reconstructed thanks to the SPaR labeling scheme and the snapshot index. This is complementary to the contribution of this paper, and we base our clustering technique on their results, as explained in Section 6.4.

9. CONCLUSION

We presented an XML-aware, XQuery-aware versioning system which seamlessly integrates time to the data model. This versioning system bridges the gap between data and document versioning and allows for powerful queries against versioned data. We implemented a first basic prototype of this versioning system, based on the Sausalito Web Application Server [8] and the Zorba XQuery engine [9]. The extensions we made are backward-compatible with non-versioning queries and there is no significant loss of performance for traditional queries.

10. REFERENCES

[1] Adobe acrobat.com. http://acrobat.com.

[2] Apache subversion. http://subversion.apache.org.

[3] git. http://git-scm.com/.

[4] Google docs. http://docs.google.com.

[5] MarkLogic. http://www.marklogic.com.

[6] Microsoft office web apps. http://office.microsoft.com/en-us/web-apps/.

[7] Oracle Flashback. http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html.

[8] Sausalito: an XQuery Web Application Server. http://sausalito.28msec.com.

[9] The Zorba XQuery Engine. http://www.zorba-xquery.com.

[10] T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In M. Ibrahim, J. Küng, and N. Revell, editors, *Database and Expert Systems Applications*, volume 1873 of *Lecture Notes in Computer Science*, pages 334–344. Springer Berlin / Heidelberg, 2000.

[11] C. Andrei, M. Brantner, D. Florescu, D. Graf, D. Kossmann, and M. Zaharioudakis. Extending XQuery with Collections, Indexes, and Integrity Constraints. In *XML Prague, Czech Republic*, 2010.

[12] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robbie, and J. Siméon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, jan 2007.

[13] D. Chamberlin, D. Engovatov, D. Florescu, G. Ghelli, J. Melton, and J. Siméon. XQuery Scripting Extension 1.0 Working Draft. http://www.w3.org/TR/xquery-sx-10/.

[14] D. Chamberlin, D. Florescu, and J. Robbie. XQuery Update Facility. http://www.w3.org/TR/xquery-update-10/.

[15] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

[16] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Version Management of XML Documents. In *The World Wide Web and Databases: WebDB2000*, 2000.

[17] S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In *2nd International Conference on Web Information Systems Engineering (WISE)*, 2001.

[18] C. Combi, N. Lavarini, and B. Oliboni. Querying semistructured temporal data. In T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, and J. Wijsen, editors, *Current Trends in Database Technology – EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 625–636. Springer Berlin / Heidelberg, 2006.

[19] C. E. Dyreson. Observing Transaction-Time Semantics with TTXPath. In *WISE '01: Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'01) Volume 1*, page 193, Washington, DC, USA, 2001. IEEE Computer Society.

[20] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.

[21] G. Fourny, D. Florescu, D. Kossmann, and M. Zaharioudakis. A Time Machine for XML: PUL Composition. In *XML Prague*, 2010.

[22] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *In VLDB*, 2003.

[23] J. Gray. Database Operating Systems: Storage and Transactions. In *SIGMOD Invited Talk*, 2006.

[24] D. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction Time Support Inside a Database Engine. In *ICDE*, 2006.

[25] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xpath-functions/.

[26] A. O. Mendelzon, F. Rizzolo, and A. Vaisman. Indexing Temporal XML Documents. In *proceedings of the 30th international conference on Very Large DataBases*, pages 216–227, 2004.

[27] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: insert-friendly XML node labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, New York, NY, USA, 2004. ACM.

[28] V. J. Tsotras and N. Kangelaris. The Snapshot Index: An I/O-Optimal access method for timeslice queries. *Information Systems*, 20(3):237–260, May 1995.

[29] F. Wang and C. Zaniolo. Temporal queries and version management in XML-based document archives. *Data and Knowledge Engineering*, 65(2):304–324, May 2008.

[30] M. Winslett. Jim Gray Speaks Out. *Sigmod Record*, June 2008.

[31] V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In I. Mulberry Technologies, editor, *Proceedings of Balisage: The Markup Conference 2009*, 2009.