

A Time Machine for XML

Ghislain Fourny
ETH Zürich
Zürich, Switzerland
ghislain.fourny@inf.ethz.ch

Daniela Florescu
Oracle
Redwood City, CA
dana.florescu@oracle.com

Donald Kossmann
ETH Zürich
Zürich, Switzerland
donaldk@inf.ethz.ch

Markos Zacharioudakis
Oracle
Redwood City, CA
markos.za@yahoo.com

ABSTRACT

As storage (main memory, disk) becomes cheaper, the amount of available information is increasing and it is a challenge to organize it. A special but very important case is when several versions of some data are collected. Versioned information is very valuable, as it allows to go back in time, but it is only valuable if it is efficiently queryable. Nowadays, there are two main, competing versioning paradigms: on the one hand, document versioning (CVS, SVN) can version any kind of document, even binary, but is not aware of what is in the documents other than lines of text. On the other hand, versioned databases provide fine-grained query capabilities, but only on highly structured data. Our broader aim is to provide a unified framework for efficiently versioning and querying data, documents, as well as any kind of semi-structured information between data and documents, which can be stored as XML. In order to query this information, we choose to start with the XQuery programming language, and to extend its data model, its syntax and its processing model to make it seamlessly time-aware. We also provide a new data structure as well as algorithms for the efficient implementation of such a versioning system.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Performance, Standardization

Keywords

XQuery, Versioning, XML, CVS, SVN

1. INTRODUCTION

Over the last few decades, following Moore's law, hardware trends have been going towards more, cheaper storage. In addition, software allows compression, virtually increasing the amount of data stored. Main memory databases are getting more widespread. According to EMC, more than 600 EB (600,000,000 TB) of data will have been created in

2009 - already 647 EB when this paper was written. With so much memory, it becomes useless to delete information, unless there are good reasons other than there being not enough space. With services such as Gmail, you are invited to keep all your e-mails and to search them for any information you might need.

The benefits of not deleting information do not stop here. Keeping track of the information allows to go back in time, compare, analyze, and produce even more information.

Finally, tools for working collaboratively are proliferating. Google, Adobe and soon Microsoft are offering Web applications for word processing, spreadsheets, presentations, which allow users to edit, share and view documents in real time.

However, it is not enough to just accumulate information and versions of the same document side by side. This has to be done in a smart way so as to fully benefit from all available information.

There are two basic approaches to keep track of the past versions of a document:

- Traditionnally, time can be modeled as a parameter in any application, i.e., as part of the application. This can be very tedious, as extra code is needed to explicitly refer to time, and this leads to poor performance.
- A more modern approach would be to add a temporal model to the database. This only needs to be done once, and then it is possible to seamlessly go back in time for any application using this temporal model.

The state of the art is that there exists document versioning systems such as SVN, CVS or git, which analyze documents line by line, compare and compress them. There are also versioning systems for structured data, such as versioned databases.

The general contributions of our work are (i) to bridge the gap between documents and data (semi-structured data) and provide a unified model for data versioning, document versioning and anything between data and documents. (ii) to have a programming language suited for processing (powerful querying, updating) this versioned data. We chose to extend XQuery [?], which is an existing, Turing-complete W3C standard.

The contributions of this paper are as follows:

1. We extend the XQuery Data Model [?], so that we have an XML-aware versioning system.

2. We extend the XQuery programming language with new functions and new time axes, so that it is possible to query against versioned data.
3. We extend the XQuery processing model to allow check-out, check-in and collaborative work.
4. We provide data structures and algorithms for the implementation of such a versioning system, together with possible optimizations.

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. Section 4 introduces our extension of the XQuery programming language. In Section 5, we give an overview of the processing model for checking out from and checking in to the repository. Section 6 presents the data structure which implements the versioning system together with querying and updating algorithms.

2. MOTIVATING EXAMPLE

The running example for this paper is the following: a research group has a weekly lunch seminar and collaboratively maintains a spreadsheet with relevant information, stored as XML. This is semi-structured information, so that it could not directly be stored in a relational database (unless canonical XML mappings are used).

Initially, the document looks as shown in Figure 1(a). Figures 1(b) and 1(c) show two subsequent versions after modifications, described below.

(a)

Weekly Lunch Seminar			
Wednesdays	12pm		
Date	Talks		By

(b)

Weekly Lunch Seminar			
Fridays	12pm		
Date	Talks		By
Sept. 18	Cloud Computing and Ducks		Donald Duck
Attendees	Meal		
Donald	Quattro Formaggi	The boss	

(c)

Weekly Lunch Seminar			
Fridays	12pm		
Date	Talks		By
Sept. 18	Cloud Computing and Ducks		Donald Duck
Attendees	Meal		
Mickey	Hawaii	Have to go at 1pm	
Donald	Quattro Formaggi	The boss	

Figure 1: (a) (b) and (c) show the different versions of a spreadsheet.

The spreadsheet is stored as XML (e.g., with Microsoft Excel or a similar format). The first version could be stored as:

```
<?xml version="1.0" encoding="UTF-8"?>
<Worksheet Name="Seminar">
  <Table>
    <Column Index="2" Width="96.0"/>
    <Row Index="1">
      <Cell Index="1" MergeAcross="2">
        <b>Weekly</b> Lunch Seminar
      </Cell>
    </Row>
    <Row Index="3">
      <Cell Index="1">Wednesdays</Cell>
      <Cell Index="2">12pm</Cell>
    </Row>
    <Row Index="5">
      <Cell Index="1">Date</Cell>
      <Cell Index="2">Talks</Cell>
      <Cell Index="4">By</Cell>
    </Row>
  </Table>
</Worksheet>
```

Then somebody edits the document, by modifying the day to Friday, inserting a talk, and inserting the boss in a new attendee table, which leads to a second version. Finally, somebody adds a new attendee by first inserting a new row with his name, and then inserts his pizza preference and a comment. This creates a third version.

Reasons why versioning documents is important are widespread in literature [?] [?], so that we take this for granted. In our example, a speaker could cancel a talk, but then hold her talk later, in which case we need to get the corresponding data.

The spreadsheet, stored as XML, is too structured to be simply considered as a document (lines of text), and not structured enough to be considered a table (relational database). This means that we need to version not only documents, but also data, and anywhere between documents and data.

Our main emphasis is that not only versioning documents is important, but also the ability to flexibly query against past versions. For example, we could be interested in the average number of participants during a semester, in the average number of talks each week, or in getting a list of all participants in the past together with the food they ordered.

What we need is:

- A versioning system supporting any kind of information: data, documents, or anything inbetween (semi-structured). We are bridging the gap between CVS and versioned relational databases.
- The ability to use powerful queries against versioned information. It should be possible to express these queries in a declarative way, leaving leeway for optimization. Also, we need the same querying power as in versioned relational databases.

The solution we propose to fulfill these needs is (1) XML-aware versioning to handle any kind of data. The versioning system should be aware of XML trees and nodes. This means that we have to extend an XML data model to allow versions of nodes and trees. (2) XQuery-aware versioning for powerful queries and updates. It should be possible to easily navigate to past versions with XQuery code, and to create new versions with XQuery Update/Scripting code. This means that (i) the XQuery syntax should be extended

to allow navigation to past versions (ii) deltas between two versions should correspond to Pending Update Lists as defined in the XQuery Update Facility specification.

3. EXTENSIONS TO THE DATA MODEL

3.1 Tree timelines

At this point, we would like to recall our broader aim: 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) [?].

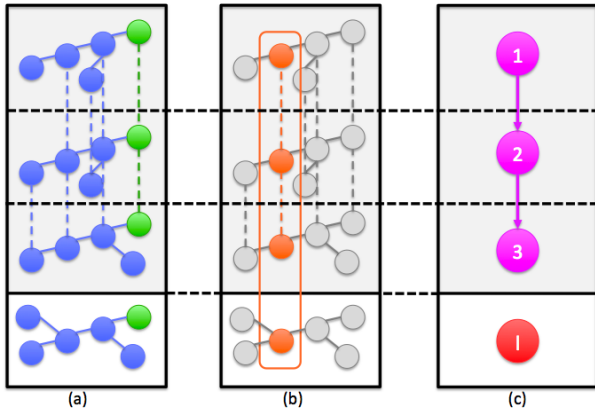


Figure 2: (a) a tree timeline as a sequence of trees. The root node timeline is shown in green. (b) a node timeline as a sequence of nodes (in orange) (c) a sequence of versions. The tree, nodes and versions at the bottom correspond to the local modifications by the user, not committed yet.

We introduce the following definitions:

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.

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 tree 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.

Figure 2 shows how tree timelines, node timelines and versions are related.

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

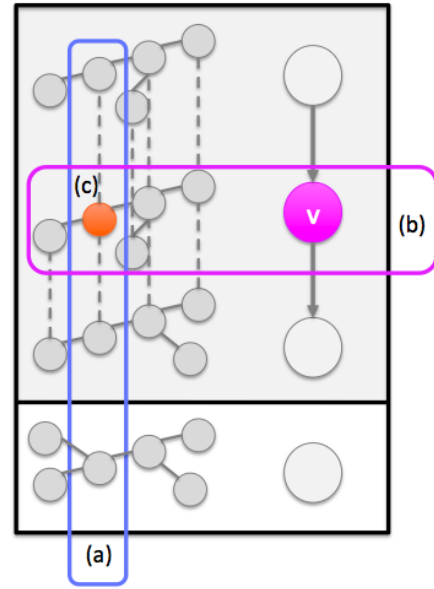


Figure 3: A node timeline URI (a) together with a version URI (b) uniquely identify a node item (c).

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

The node timeline URI and the version URI uniquely identify a node item (Figure 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 space, and then in time.

In the remaining 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 (tree) uniquely identified by this version and the node (tree) timeline. This should be clear from the context.

3.2 Collection timelines

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 (Figure 4).

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.

In the remaining of the paper, we will rather focus on tree timelines than on collection timelines, but we think that the framework for collection timelines can be deduced from the framework for tree timelines.

3.3 Serialized Pending Update Lists

As we are performing updates using XQuery Update/Scripting, deltas between two versions of a tree timeline can

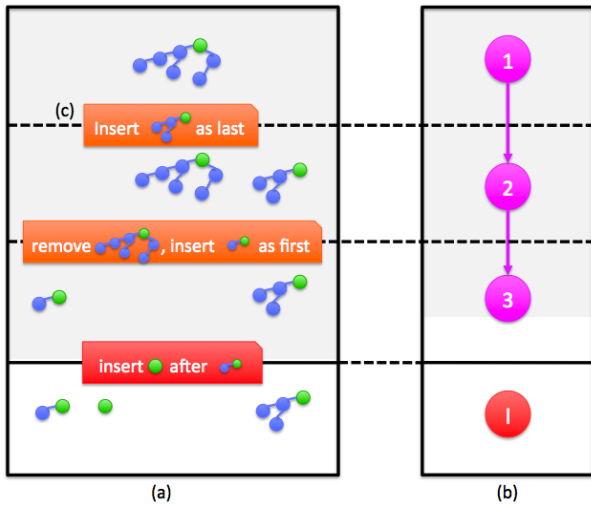


Figure 4: (a) A collection timeline is a succession of collections. (b) Versions can also uniquely identify a collection in a collection timeline. The bottom collection is local: it is being modified by the user and has not been committed yet. (c) Deltas between two versions are modeled as serialized PULs

theoretically be expressed as a sequence of PULs (Pending Update Lists), as a sequential, side-effecting expression might apply several PULs before committing. These PULs might be dependent on one another. Actually, such a sequence of PULs can be aggregated to a single PUL as will be explained in Section 6 (PUL Composition). In order for deltas to be queryable, we choose to serialize the corresponding Pending Update List to XML.

A *serialized Pending Update List* is a Pending Update List (as defined in XQuery Update Facility) 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. Such a serialized Pending Update List can be expressed as XML and is queryable with XQuery. For example:

```
<pending-update-list
  xmlns="http://www.example.com/spul">
  <insertBefore>
    <target>
      http://www.example.com/theTarget
    </target>
    <content>
      <xdm><a/></xdm>
    </content>
  </insertBefore>
  <delete>
    <target>
      http://www.example.com/theTarget2
    </target>
  </delete>
</pending-update-list>
```

A PUL can be serialized to a serialized PUL. A serialized PUL can be deserialized to a PUL.

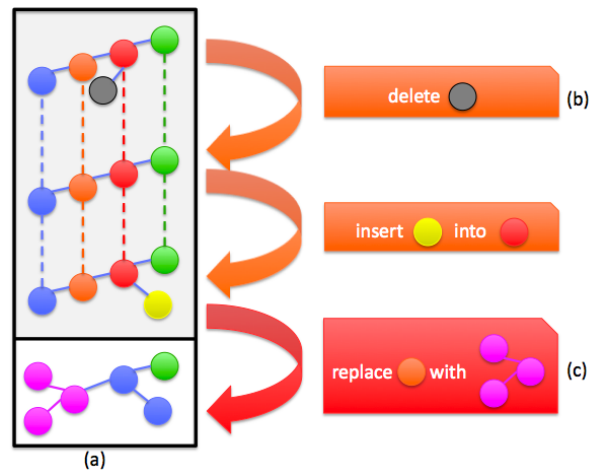


Figure 5: The delta between two versions of a tree timeline (a) can be described as a serialized PUL (here we show only one update primitive in each PUL, but a single PUL can have several update primitives) (b). Locally, a serialized PUL is also maintained with a delta to the local version (c).

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 through time, the following functions are added to the XQuery functions and operators [?] and are available to the user:

- `vng:reference` which takes a node and returns its node timeline URI (obtained through `dm:node-timeline`).
- `vng:version` which takes a node and returns its version URI (obtained through `dm:version`).
- `vng:dereference` which takes a node timeline URI and returns its local version.
- `vng:ttdereference` which takes a node timeline URI and a version URI and returns the corresponding 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)
vng:ttreference($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::, earlier::, past::, past-or-current::, current::, later::, future::, future-or-current::, last::, all-times:: and local::. For example, later::* 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::* / later::* will always return the empty sequence, as well as first::* / earlier::*.

Time axes are compatible with node tests:

```

<time axis>:::<node test>

is defined as

<time axis>::* / self::<node test>.

```

For example

```
future:::mickey
```

is defined as future::* / 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.

We also suggest the following abbreviations: ++ for later::* , - for earlier::* , +++ for last::* — for first::* and ++++ for local::* .

Back to our motivating example in Section 2, the following query outputs the name of all participants to the Weekly Lunch Seminar until now (the version of \$doc) as well as all meals ordered by them:

```

let $participants :=
  $doc//Cell[. = "Attendees"]/parent::Row
  /past-or-current::: *
  /following-siblings:::Row/Cell[@Index=1]
for $participant-unique
in fn:distinct-values($participants)
return
<participant name="{ $participant-unique }">
  for $meal
  in $participants[. = $participant-unique]

```

```

  /following-siblings:::Cell[@Index=2]
  <meal>{ $meal } </meal >
</participant >

```

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 (corresponding to 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 Newly created nodes

Newly created nodes (with element constructors or nodes copied in updating expressions) belong to no timeline and are in no version: vng:reference(.) and vng:node-version(.) raise an error.

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 Current processing models

The XQuery language is made of three parts:

- A side-effect-free core (W3C recommendation) [?] which just reads, processes XML documents, and outputs an XDM instance.
- The XQuery Update Facility (W3C candidate recommendation) [?] which reads XML documents and outputs a list of changes (Pending Update List) that can subsequently be applied to these XML documents.
- The XQuery Scripting Extension (W3C working draft) [?] which reads from and writes to XML documents.

5.2 Checkout and checkin

We saw earlier that there is a local version corresponding to local, non-committed yet, modifications. The nodes in this version are only visible to the user.

The local version of a tree timeline is obtained by checking out the tree timeline from the repository. It can then be edited, and committed by checking in. To make checkouts and checkins as flexible and configurable as possible, we use checkout and checkin policies, as is illustrated on Figure 6.

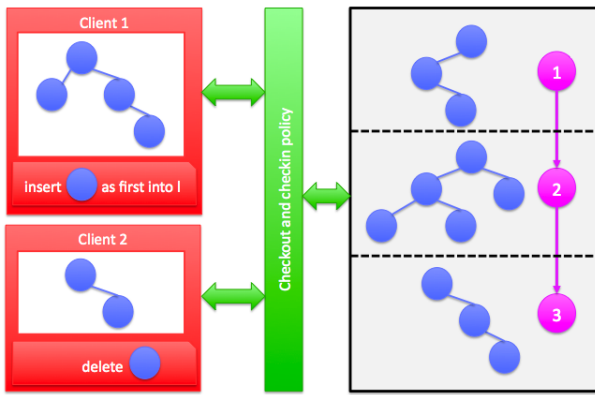


Figure 6: The checkout-checkin processing model

Checkouts and checkins are performed in a symmetric way. Both of them can be done implicitly, or explicitly with a sequential 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.

Checkins and checkouts are atomic and isolated operations: there can only be one checkout or checkin at a time.

5.3 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:

1. The URI of the tree timeline to check out
2. The URI of the version to check out
3. The PUL containing local changes (empty if it is the first checkout or if there are no local changes)

It outputs:

1. The local version of the tree timeline which has just been checked out
2. The updated PUL containing local changes
3. A boolean indicating whether it has been successful. If this boolean is false, then the local version and the local PUL are reverted.
4. 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.

5.4 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:

1. The URI of the tree timeline to check in
2. 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:

1. The PUL which is to be applied to the last version in the repository.
2. 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.
3. Possibly an XDM instance containing unsolved conflicts if the checkin failed.

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

5.5 Using Checkout and Checkin policies

Again, 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 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 by applying a serialized PUL on the last version.
- provide proofs of correctness, formulated as theorems stating (i) that deltas between versions can be modeled by a single serialized PUL and (ii) that the two algorithms defined above lead to the expected behavior of the data structure.

6.1 Pi-Trees and Pi-Forests

From a logical point of view, it is very easy 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 as defined in the XQuery Update Facility specification. The resulting tree is the new version.

Keeping each version as whole tree in memory would be very expensive. This is why we define a new data structure called π -tree to implement a tree timeline.

For commodity, we extend it to π -forests, of which π -trees are a special case.

DEFINITION 1. A π -tree is a tree whose nodes have an identity l , a time-to-name mapping n , a creation time c and a deletion time d . The creation time and deletion times are positive integers, possibly infinite. An abstract π -forest is a sequence of π -trees.

In practice, identities are unique, however we do not need this assumption for our proofs of correctness. The set of identities could be, for example, the set of pointer addresses in memory, or the set of URI references, or even a mixture of both.

π -forests are illustrated on figure 7.

DEFINITION 2. π -forests can be recursively defined as follows:

- \emptyset is a π -forest.

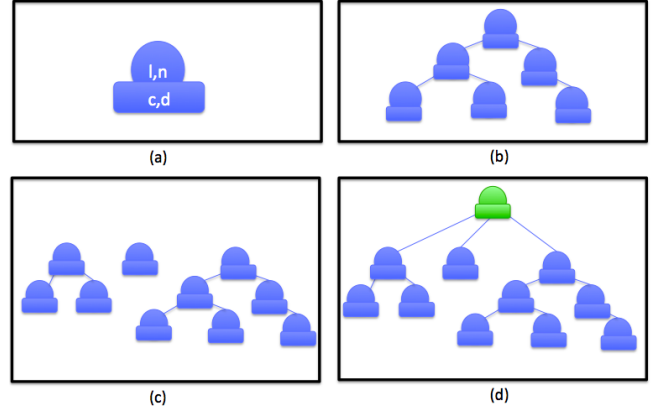


Figure 7: (a) An π -node with an identity, a time-to-name mapping, a creation and a deletion time. (b) A π -tree. (c) A π -forest as a sequence of π -trees (d) A π -tree obtained by induction using a node and the former π -forest.

- If $[l, n, c, d]$ is a node and F is a π -forest, then $\begin{matrix} [l, n, c, d] \\ \Delta \\ F \end{matrix}$ is a π -forest (and a π -tree).
- If F and G are two π -forests, then (F, G) is a π -forest.

We provide an algorithm to retrieve a version from the π -forest data structure. This is done with the instantiation function. In this algorithm, we model the versions of the tree timeline implemented by the π -tree as trees and forests in which each node has an identity and a name. These trees and forests can be recursively defined in a similar manner to π -trees and π -forests.

DEFINITION 3. The instance $inst(F, t)$ of a π -forest F at t is recursively defined as:

- $inst(\emptyset, t) = \emptyset$
- $inst((F, G), t) = (inst(F, t), inst(G, t))$
- $inst\left(\begin{matrix} [l, n, c, d] \\ \Delta \\ G \end{matrix}\right) = \begin{cases} [l, n(t)] & c \leq t < d \\ \Delta & \\ inst(G, t) & \end{cases}$ otherwise

Computing instances is illustrated on figure 8. For example, instance 1 could be the first version of the Weekly Lunch Seminar document. Then an attendee (d) could be added in version 2, and in version 3, the first subtree of attendees (b, c, d, who graduated) is replaced with a new subtree of attendees (e, f, who are beginning their PhD).

This 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 an instance (up to a constant factor) by using indexes. For each node of the data structure, the index maps time intervals to the children which exist within this time interval. The children not pointed to by an interval need not be considered in the instantiation for times in this interval.

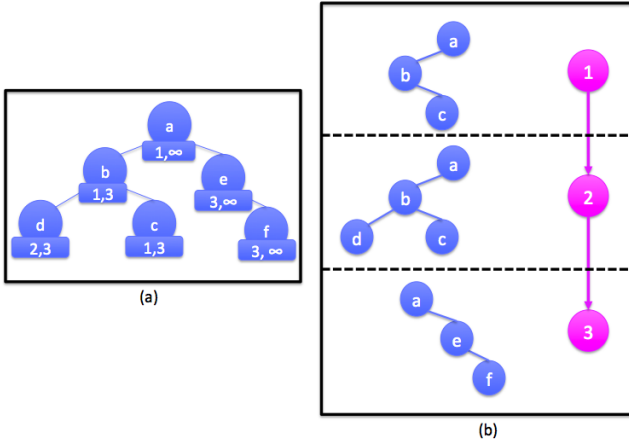


Figure 8: (a) A π -forest. Identities and time-to-name mappings are simplified and represented with a single letter, but creation and deletion times are shown. (b) The instances 1, 2 and 3 of the π -forest.

6.2 PUL Composition

In a tree timeline, each tree is derived from the former by applying updates. These updates can be contained in a single PUL if they correspond to the output of an XQuery Update program. However, they can also be contained in several, interdependent PULs if they correspond to the execution of a fragment of an XQuery Scripting program containing several Apply Expressions. In this part, we briefly present how these interdependent PULs can be composed and aggregated to a single PUL which can then be serialized. This means that the delta between different versions of a tree timeline can always be modeled as a *single* serialized PUL.

Our first main theorem states that PULs are composable, as shown on figure 9. In the theorem, we use the dot notation $p.f$ to apply a PUL p to a forest f . $p.f$ is the resulting forest after the PUL has been applied.

THEOREM 1. *There exists a composition operator $*$ taking two PULs as operands and returning a PUL, satisfying:*

$$\forall \text{PULs } p, p', \forall \text{forest } f, p'.(p.f) = (p' * p).f$$

The proof, by structural induction on forests, is omitted (but it is available from us). Appendix A explicitly defines the composition operator $*$ which fulfills Theorem 1, defined on an abstract version of PULs.

This legitimates the fact that deltas between versions can be expressed as single (serialized) PULs.

6.3 Updating a Pi-Forest

In Appendix B, we give the algorithm to apply updates (contained in a PUL p) 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:

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

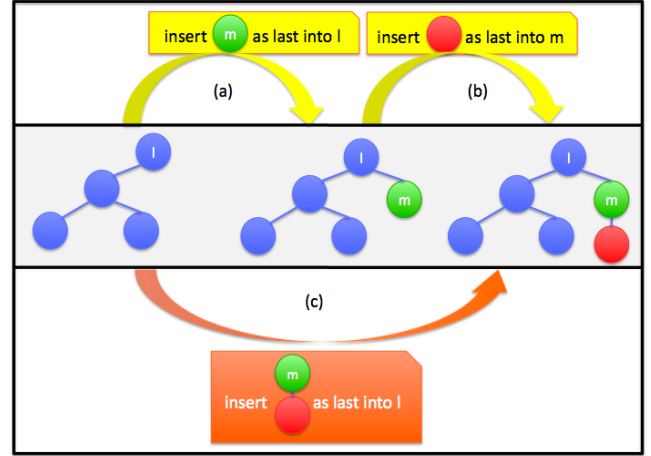


Figure 9: PUL Composition: applying the PUL (a) and then (b) is equivalent to directly applying (c) obtained with the $*$ operator.

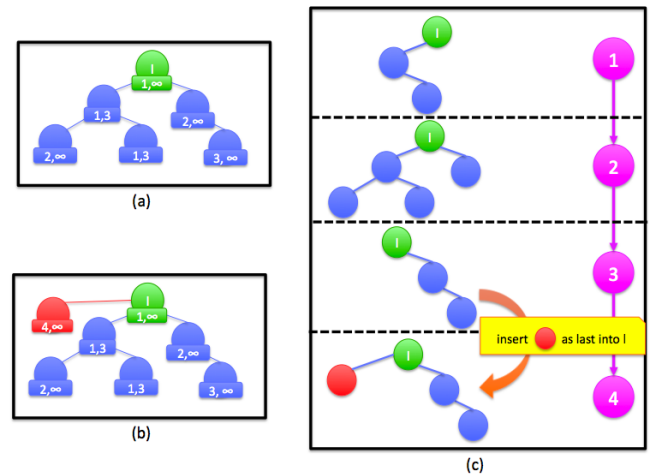


Figure 10: The older π -tree (a) can be instantiated to the first three versions in (c). The updated π -tree (b) instantiates to the exact same first three versions, but also to a fourth version, which is exactly that obtained from the third version by applying the PUL.

These two constraints are illustrated on figure 10.

The complexity of the algorithm is $O(s)$ where s is the size of the PUL, assuming that nodes can be randomly accessed with a reference in $O(1)$ and if no indexes are used in the data structure. If, however, indexes are used to reduce instantiation time, then they need to be updated as well, which increases the complexity to the size of the new instance (up to a constant factor).

Hence, there is a compromise to find between query time and update time. With no indexes, query time grows with the size of the data structure but update time remains constant (provided the PULs remain constant), whereas with indexes, query time only grows with the size of the instances (i.e., there is almost no overhead w.r.t. to an unversioned tree) but update time then also grows with the size of the instances.

Our second main theorem states that the data structure behaves as expected when it is updated and queried. The second theorem is formally given in Appendix B, as it depends on the formal definition of the updating algorithm.

The interpretation of the two parts of the theorem are the two constraints stated above, i.e.:

1. updating the data structure to create a new version t with the PUL p does not alter previous versions.
2. in the updated data structure, the newly created version t is obtained from version $t - 1$ by applying the PUL p

The proof is again a structural induction on π -forests and is omitted (but is available).

7. RELATED WORK

There are other proposals around for versioning XML data.

Zholudev and Kohlbase developed TNTBase [?] which is a versioning system for versioning XML documents on top of an SVN repository, linked to an XML database. TNTBase seems to focus on document-oriented versioning, e.g., deltas are regular SVN deltas.

Fusheng Wang and Zaniolo [?] introduce an XML versioning system addressing data and document versioning. It allows temporal queries by introducing an explicit time parameter in XML documents (with a creation and a deletion time, like in π -trees). Queries have to be aware of this and explicitly include time filtering. Furthermore, versioning a document requires that it be 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.

8. CONCLUSION

We presented an XML-aware, XQuery-aware versioning system which seamlessly integrates time in 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 [?] and the Zorba XQuery engine [?]. Sausalito already implements a checkout/checkin model [?] which we extended to allow for versioning.

The extensions we made are backward-compatible with queries not using the versioning system. Future work includes a performance overhead benchmark to measure the overhead induced by using a versioned store compared to a non-versioned store.

Also, we would like to investigate how to build XML indexes on top of such a versioning system.

APPENDIX

Appendixes A and B aim at giving more technical details on the results of Section 6 to the interested reader.

A. BACKGROUND ON THE PUL COMPOSITION THEOREM

For the definition of PUL composition, we abstract from sequences of XML trees and model them as forests of trees (each node $[l, n]$ having an identity l and a name n), defined recursively in a similar manner to π -forests. We use the same triangle notation to build a new tree from a node and a forest (the children).

Then, we use an abstract model for PULs, used in the definition of PUL composition (Note that inserting attributes could be done on top of this model using sentinel nodes).

DEFINITION 4. *An abstract PUL is an unordered sequence containing abstract update primitives, i.e.:*

- *insert t as first into l*
- *insert t as last into l*
- *insert t before l*
- *insert t after l*
- *delete l*
- *replace l with t*
- *replace value of l with t*
- *rename l to n*

where t is a tree, l is an identity reference and n is a name.

We collapse abstract “insert” update primitives of the same type and with the same identity target. The XQuery Update Facility gives us leeway for that, as this behavior is implementation-dependent [?] (2.4.1). Hence, we can assume each kind of insert occurs at most once. Furthermore, abstract PULs are constrained in that there is at most one “replace”, at most one “replace value of” and at most one “rename” update primitive for the same target, as having more than one update primitive for each of these kinds raises dynamic errors in the XQuery Update Facility specification (3.2.2.1).

An abstract PUL p can be partitioned w.r.t. its target identities $(p_i)_i$. Each partition p_i can be further expressed as (insert u_i as first into l , insert v_i as last into l , insert w_i before l , insert x_i after l , delete l ?, replace l with r_i ?, replace value of l with v_i ?, rename l to n_i ?), which we abbreviate as $(u_i, v_i, w_i, x_i, d_i, r_i, y_i, n_i)$. u_i, v_i, w_i and x_i are possibly empty forests of trees, d_i can be 0 (no delete) or 1 (delete l is in the PUL), and r_i, y_i are forests and n_i is an XML name.

r_l, y_l and n_l are possibly \perp , meaning that the corresponding update primitives are absent in the abstract PUL.

An abstract PUL $p = (u_l, v_l, w_l, x_l, d_l, r_l, y_l, n_l)_l$ can be applied to a forest f to produce a new forest $p.f$, defined as follows, following the XQuery Update Facility specification (3.2.2.2).

DEFINITION 5. For an abstract PUL $p = (u_l, v_l, w_l, x_l, d_l, r_l, y_l, n_l)_l$, and forests f, g (0 is the empty forest) the apply operator \cdot is recursively defined as follows:

- $p.0 = 0$
- $p.(f, g) = (p.f, p.g)$
- $p.\left(\begin{array}{c} [l, n] \\ \Delta \\ g \end{array}\right) = \begin{cases} (w_l, r_l, x_l) & \text{if } r_l \neq \perp \\ \text{else } (w_l, x_l) & \text{if } d_l = 1 \\ \text{else } (w_l, \Delta, x_l) & \text{if } y_l \neq \perp \\ \text{else } (w_l, \Delta, x_l) & \text{otherwise} \end{cases}$

$$\text{where } m = \begin{cases} n_l & \text{if } n_l \neq \perp \\ n & \text{otherwise} \end{cases}$$

Actually, it is even more general than the specification, as it allows for a target identity of the abstract PUL to be in the content of an other abstract update primitive in the same abstract PUL (example: insert $[a, n]$ into b , insert $[b, m]$ into c). For this more general case, which never happens in practice, we apply strict snapshot semantics, i.e. the first update primitive does not "see" the second one when it is applied.

Now that we have defined forests, abstract PULs and the apply operator, we can introduce the $*$ operator which fulfills Theorem 1:

DEFINITION 6. Given two abstract PULs

$p = (u_l, v_l, w_l, x_l, d_l, r_l, y_l, n_l)_l$
and
 $p' = (u'_l, v'_l, w'_l, x'_l, d'_l, r'_l, y'_l, n'_l)_l$,
the composition operator $p' * p$ is defined for each l as follows:

- if $r_l \neq \perp$ then
 $(p' * p)_l = (0, 0, p'.w_l, p'.x_l, 0, p'.r, \perp, \perp)$
- else if $d_l = 1$ then
 $(p' * p)_l = (0, 0, p'.w_l, p'.x_l, 1, \perp, \perp, \perp)$
- else if $y_l \neq \perp$ then
 $(p' * p)_l = (0, 0, (p'.w_l, w'_l), (x'_l, p'.x_l), d'_l, r'_l, \text{first of } (y'_l, p'.y_l), \text{first of } (n', n))$
- otherwise
 $(p' * p)_l = ((u'_l, p'.u_l), (p'.v_l, v'_l), (p'.w_l, w'_l), (x'_l, p'.x_l), d'_l, r'_l, y'_l, \text{first of } (n', n))$

where first of (a, b) means a if $a \neq \perp$, b otherwise.

B. BACKGROUND ON THE PI-TREE PROOF OF CORRECTNESS

When a checkin occurs, a PUL p is output, which is to be logically applied to the last version of the tree timeline in the repository. The corresponding updating algorithm for the underlying PI-tree T outputs an updated data structure denoted $p^t.T$ defined as follows:

DEFINITION 7. For an abstract PUL $p = (u_l, v_l, w_l, x_l, d_l, r_l, y_l, n_l)_l$, the updating operator at t is defined as:

- $p^t.0 = 0$
- $p^t.(F, G) = (p^t.F, p^t.G)$
- $p^t.\left(\begin{array}{c} [l, n, c, d] \\ \Delta \\ G \end{array}\right) = \begin{cases} \left(\begin{array}{c} [l, n, c, d] \\ \Delta \\ G \end{array}\right) & d \neq +\infty \\ \text{else } (C(w_l, t), \left(\begin{array}{c} [l, n, c, t] \\ \Delta \\ G \end{array}\right), C(r_l, t), C(x_l, t)) & r_l \neq \perp \\ \text{else } (C(w_l, t), \left(\begin{array}{c} [l, n, c, t] \\ \Delta \\ G \end{array}\right), C(x_l, t)) & d_l = 1 \\ \text{else } (C(w_l, t), \left(\begin{array}{c} [l, m, c, t] \\ \Delta \\ D(G, t), C(y_l, t) \end{array}\right), C(x_l, t)) & y_l \neq \perp \\ (C(w_l, t), \left(\begin{array}{c} [l, m, c, d] \\ \Delta \\ C(u_l, t), p^t.G, C(v_l, t) \end{array}\right), C(x_l, t)) & \text{otherwise} \end{cases}$

where

- C is the creation function:

$$C\left(\begin{array}{c} [l, n] \\ \Delta \\ g \end{array}, t\right) = \left(\begin{array}{c} [l, n, t, +\infty] \\ \Delta \\ C(g, t) \end{array}\right),$$

$$C(0, t) = 0 \text{ and } C((f, g), t) = C(f, t), C(g, t)$$

- D is the deletion function:

$$\left(\begin{array}{c} [l, n, c, d] \\ \Delta \\ G \end{array}, t\right) = \left(\begin{array}{c} [l, n, c, (d = +\infty?t : d)] \\ \Delta \\ D(G, t) \end{array}\right),$$

$$D(0, t) = 0 \text{ and } D((F, G), t) = D(F, t), D(G, t)$$

- m is the new time-name mapping $m(u) = n(u)$ for $u < t$ and $m(t) = n_l$

The theorem stating the correctness of the data structure is as follows:

THEOREM 2. For any p , any t and any π -forest F so that all creation times and deletion times are smaller than t :

1. $\text{inst}(p^t.F, u) = \text{inst}(F, u)$ for $u < t$
2. $\text{inst}(p^t.F, t) = p.\text{inst}(p^t.F, t - 1)$