Utah State University

DigitalCommons@USU

All Graduate Plan B and other Reports | Graduate Studies

8-2011

Creating a Representation of Items and Version that Support Efficient Evaluation of the Transaction-Time Axis in SML-Based Databases

Kaylan Goutham Mekala
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Computer Sciences Commons

Recommended Citation

Mekala, Kaylan Goutham, "Creating a Representation of Items and Version that Support Efficient Evaluation of the Transaction-Time Axis in SML-Based Databases" (2011). *All Graduate Plan B and other Reports*. 59.
https://digitalcommons.usu.edu/gradreports/59

This Report is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Plan B and other Reports by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

CREATING A REPRESENTATION OF ITEMS AND VERSIONS THAT SUPPORT EFFICIENT EVALUATION OF THE TRANSACTION-TIME AXIS IN XML-BASED DATABASES

by

Kalyan Goutham Mekala

A report submitted in partial fulfillment
Of the requirements for the degree
of
MASTER OF SCIENCE
in
COMPUTER SCIENCE

Approved:

Dr.Curtis Dyreson
Major Professor

Dr.Stephen W. Clyde
Committee Member

Dr. Daniel Bryce
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah
2011

ABSTRACT

Creating a Representation of Items and Versions That Support Efficient Evaluation of the Transaction Time Axis in XML-based Databases

by

Kalyan Goutham Mekala, Master of Science
Utah State University, 2011

Major Professor: Dr. Curtis Dyreson
Department: Computer Science

This project was developed to create a platform for implementing the features and query support provided by the transaction time axis (tt-axis). The basis for this platform is a new numbering plan called item version timestamp level numbering (IVTLN), and it extends an existing numbering plan, namely, dewey level numbering (DLN), by including version and timestamp information. Thus, the transaction time axis provides a temporal perspective for XML nodes in addition to non-temporal axes like the ancestor and descendant axes. This project provides an efficient, extensible, and comprehensible platform for the implementation of the new numbering plan and the services provided by the transaction time axis.

AKNOWLEDGMENTS

I would like to thank my major professor, Dr. Curtis Dyreson, for all his support and guidance through the course of this project. There were many trying times during the design and development of this project, the consequence of which I have become a better student and problem solver.

I also wish to thank my committee members, Dr. Stephen Clyde and Dr. Daniel Bryce, for extending their support.

Kalyan Goutham Mekala

CONTENTS

ABSTRACT..

ACKNOWLEDGMENTS ....

LIST OF FIGURES ....

LIST OF TABLES ....

CHAPTER

ORGANIZATION 8

PAPER - Prefix-based Node Numbering for Temporal XML.................9

INTRODUCTION 33

OBJECTIVES AND METHODOLOGY..38

NEW NUMBERING PLAN 41

SCHEMA CHANGE 47

QUERYING...58

IMPLEMENTATION...67

MEMORY ANALYSIS 83

CONCLUSION..88

REFERENCES 91

LIST OF TABLES

Table 1. DLN for Course Info. *43*

Table 2. Schema Change Tracking Table.. *45*

Table .3 Course Info (Instructor Name) Details During Fall 2000.................................. *50*

Table 4. Course Info (Instructor Name) Details During Spring 2001................................ *50*

Table 5. DLN Changes to Course Info.... *52*

Table 6. Numbering Change Due to Schema Change. *53*

Table 7. Change in Node ID Due to Schema Change (Schema Change Tracking Table) . . *56*

Table 8. Future Access Table.... *57*

Table 9. Schema Change Tracking Table. *63*

Table 10. Schema Change Tracking Table. *65*

Table 11. Future Access Table for the current example.... *65*

LIST OF FIGURES

Figure 1. A temporal XML fragment. 10

Figure 2. An XML fragment. 10

Figure 3. A non-temporal XML data model instance (DOM instance). 14

Figure 4. Nodes with node numbers. 14

Figure 5. The second slice (at time 2). 17

Figure 6. The third slice (at time 3.) 17

Figure 7. The first slice as a temporal data model instance with items and versions represented. 20

Figure 8. The second slice added. 21

Figure 9. The third slice added. 22

Figure 10. Element insert. 25

Figure 11. Element delete. 26

Figure 12. Value update. 26

Figure 13. Element move. 27

Figure 14. An XML file. 34

Figure 15. XQuery code. 35

Figure 16. Course Info (tree structure). 42

Figure 17. Data change in Course Info. 48

Figure 18. No schema change example. 51

Figure 19. No schema change (appending a child node after its deletion) 53

Figure 20. Before schema change. 54

Figure 21. Course Info after schema change. 55

Figure 22. The IVTLN class........ 71

Figure 23. The Item (item) package. 72

Figure 24. The Collection class. 72

Figure 25. The DLN class in eXist db......... 73

Figure 26. Adding a document to a collection (Sequence Diagram)......... 75

Figure 27. Creation of IVTLN objects (Sequence Diagram). 76

Figure 28. eXist db user login screen......... 77

Figure 29. eXist Admin Client Screen......... 78

Figure 30. eXist Client (left) and File selection (right) screens. 79

Figure 31. Screen shown while a document is being added. 80

Figure 32. eXist admin client screen (updated with bookstore4.xml)......... 81

Figure 33. bookstore4.xml. 82

Figure 34. Number of objects across five snapshots......... 85

Figure 35. Memory usage (in bytes) across five snapshots. 85

Figure 36. Impact of add, delete, and update node operations on object count......... 86

Figure 37. Impact of add, delete, update node operations on memory usage......... 86

Chapter 1

ORGANIZATION

This report is organized it as follows. Chapter 2 comprises a paper co-authored with Dr. Curtis Dyreson and submitted to the International Conference on Web Information Systems Engineering (WISE 2011). The paper presents an easily digestible summary of the research presented in the remainder of the report. Chapter 3 describes research in XML, XQuery, XPath and the new numbering plan (IVTLN) that was devised to support temporal queries and documents. Next, Chapter 4 discusses the objectives and methodology of this research. Details of the new numbering plan (IVTLN), schema changes, and their corresponding application in querying are provided in Chapters 5, 6, and 7, respectively. Chapter 8 provides an in depth discussion of the implementation, which uses an XML DBMS called eXist. Next, the details of the memory analysis are provided in Chapter 9, followed by inferences and conclusions in Chapter 10.

Chapter 2

PREFIX-BASED NODE NUMBERING FOR TEMPORAL XML

Abstract. Prefix-based numbering (also called Dewey level order or Dewey numbering) is a common, popular method for numbering nodes in an XML data model instance or document. The nodes are numbered so that spatial relationships (e.g., is a node a descendant of another) can be determined just from the numbers. In a temporal XML data collection, the spatial relationships change over time as nodes are edited, deleted, and inserted. In this paper, we adapt prefix-based numbering to support the concise representation of *items* (elements that share the same identity over time) and *versions* (changes to an item over time). We call our new numbering system time-tunneling dynamic level numbering (TTDLN). We show how to create, store, and update TTDLNs, and how they can be used to efficiently evaluate sequenced and non-sequenced temporal queries.

Introduction

XML is an important language for data representation and exchange, especially in web applications. XML is used to mark-up data by adding complex, descriptive structure. For instance, data about a book could be meaningfully marked up in XML with <title>, <author>, and <publisher> elements, each of which describes its enclosed data.

XML data collections change over time as new elements are inserted and existing elements are edited and deleted. Timestamps can be added to XML data to capture this editing history [1] [12]. As an example, Figure 1 shows a fragment of an instance of a temporal XML data model for bibliographic data. The data in Figure 1

contains information about publishers, books, and authors; it also records when each datum was put into the data collection, i.e., the timestamps represent the *transaction-time* lifetime of each element [13]. The bibliography began on 12/21/01, and remains current (until *now*). Information about the Butterfly Books publisher was entered on

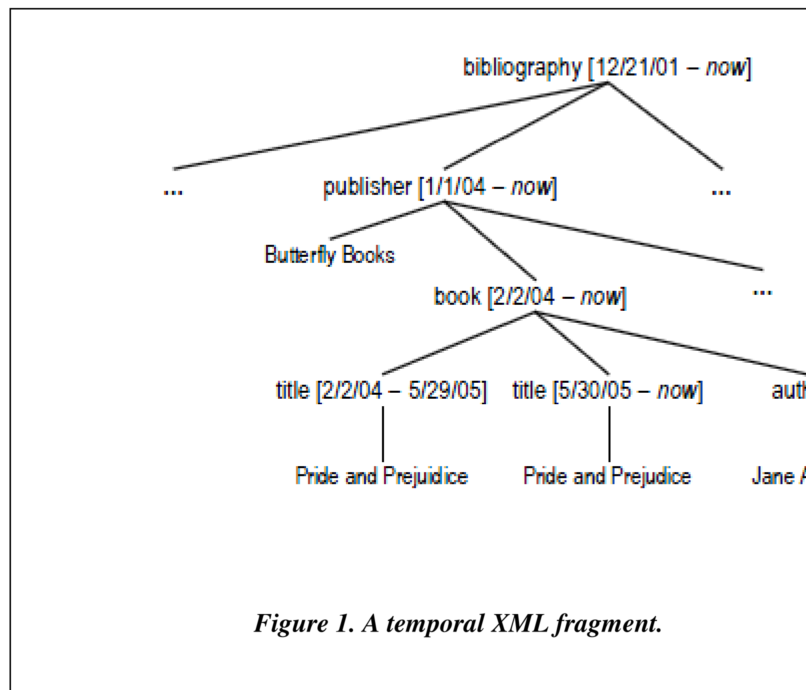


Figure 1. A temporal XML fragment.

```
...
  <part>
    <supplier name="ACME">
      <color>red</color>
    </supplier>
    engine
  </part>
...
```

Figure 2. An XML fragment.

1/1/04, and it started publishing a book by Jane Austen on 2/2/04. The title of that book was originally misspelled, and was corrected on 5/29/05.

Efficient evaluation and management of XML data have benefited from the development of numbering schemes for an XML data instance. Among the various numbering systems, *prefix-based* numbering is popular [14]. In prefix-based numbering, the prefix of each node number is its parent's node number. The node numbers are used to improve the speed of query evaluation since any query language *axis* (e.g., preceding-sibling, ancestor, following, etc.) can be evaluated by just comparing numbers.

Prefix-based numbering schemes are currently designed only for non-temporal data model instances which record the state of the data at just a single time point. For temporal data management, prefix-based numbering falls short in several ways.

- Prefix-based numbering cannot capture node identity over time. A temporal data collection has to track edits to a node over time and capture whether a node is "new" or previously existed. It is important to observe that prefix-based numbering is incapable of representing this identity over time since a node will not always have the same node number; a node can switch to a new parent or swap position with a sibling, both operations change its prefix and hence its number.
- Prefix-based numbering does not represent node versions. Each time a node is "edited," a new version of that node is created. Query operations to find previous or next versions of a node [8] need to be supported by the numbering scheme, but versions are not currently part of a number.
- Prefix-based numbering does not support sequenced evaluation, in which a

temporal query is (logically) evaluated at each point in time [10]. The numbers do not have any temporal extent so are incapable of being used for sequenced evaluation of queries.

The challenge is to construct a numbering system for the items, versions, elements, and other components of a persistent (or in memory) temporal document. The numbering system should have the following properties.

- *Conciseness* – The space cost should be proportional to the size of the document. For a temporal document, it should be linear in the number of items and versions. Said differently, it should be linear in the size of an initial slice and the size of changes over time to that slice.
- *Efficient update* – The cost of updating the numbers should be proportional to the size of an edit. Small changes should require few updates.
- *Efficient querying* – The numbers will be used to compute "relationships" between pairs of nodes, ideally in constant time. The non-temporal relationships of interest are parent, ancestor, descendent, etc. Additional temporal relationships of interest are sequenced and non-sequenced versions of all of the non-temporal relationships, plus relationships about identity over time (is same item) and changes over time (is same/later/earlier version). Ideally, all of these relationships can be determined by simply examining or comparing a number for each node.

In this paper, we extend prefix-based numbering to support the full panoply of temporal operations. We focus on dynamic level numbers (DLNs) which are a typical kind of prefix-based number. DLNs are used in eXist [2]. We show how timestamps can be associated with DLNs, how *items* (elements that have identity over time) and *versions* of

those items can be represented, how sequenced and non-sequenced queries (e.g., find next version) can be supported, how XUpdate edits are supported, and how all of this can be implemented with minimal changes to an XML DBMS by capturing the temporal history in an XML document [10].

The field of temporal databases has been an area of intense study for the past 25 years [22], with Oracle now perhaps having the most mature temporal support: transaction-time, valid-time, and bitemporal tables, current modifications, and automatic support for temporal referential integrity [18]. Concerning the representation of temporal data and documents on the web, Grandi has created a bibliography of previous work in this area [11]. Marian et al. [15] discuss versioning to track the history of downloaded documents. Chien et al. [5] have researched techniques for compactly storing multiple versions of an evolving XML document. Buneman et al. [3] provide another means to store a single copy of an element that occurs in many snapshots. This paper differs from all of the above papers since our focus is on node numbering schemes.

Dynamic Level Numbering (DLN)

For a non-temporal document, DLN is a popular node numbering scheme. In DLN each node in a DOM is numbered as follows: a node is numbered *p.k*, where *p* (the prefix) is the number of its parent and *k* represents that it is the *k*th sibling in document order. We explain prefix numbering with an example. Figure 3 shows an XML data instance for the fragment of a larger XML document that is shown in Figure 2. We focus just on this fragment in the examples in the remainder of the paper. The instance is represented as part of a larger tree with different kinds of nodes: *element*, *text*, *attribute*,

etc.; whitespace has been stripped. DLNs are assigned as shown in Figure 4.

The

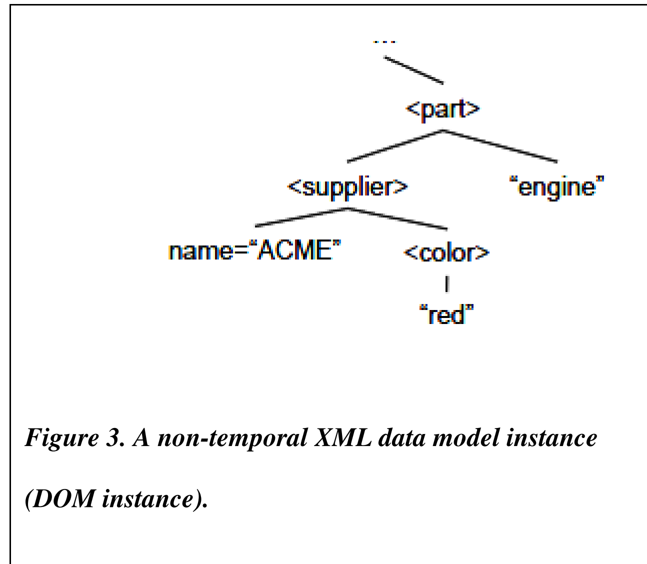


Figure 3. A non-temporal XML data model instance (DOM instance).

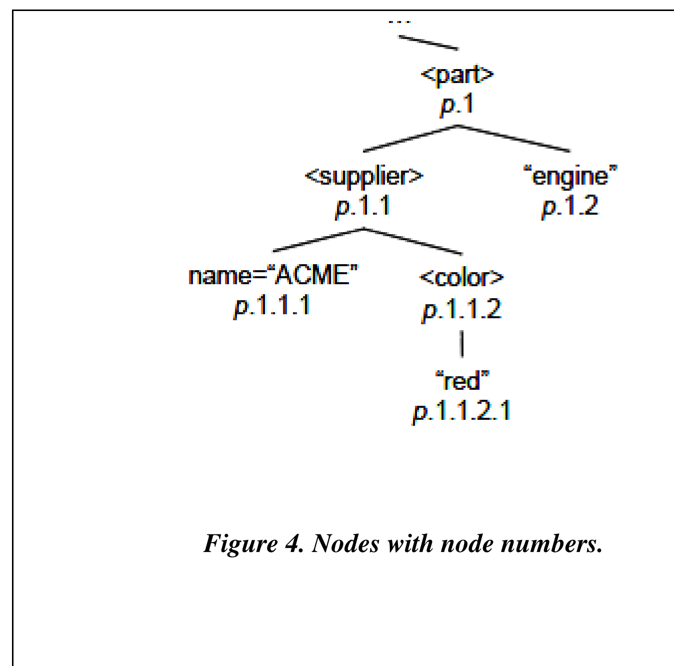


Figure 4. Nodes with node numbers.

DLNs are shown below the elements. The node numbered *p*.1 (<part>) represents the first child in document order relative to its parent (which has a DLN of *p*).

There are strategies for packing DLNs into as few bits as possible, making DLNs relatively concise [14]. DLNs are also very efficient for queries [14]. Given two DLNs, we can quickly compute (by comparing the numbers) a specific relationship (child, parent, ancestor, descendent, following sibling, preceding, etc.) of one DLN relative to another. For instance, *p*.1.1.2 can be compared to *p*.1.2. Since *p*.1.1.2 is neither a prefix nor a suffix of *p*.1.2, it is not a child, parent, ancestor, or descendent. *p*.1.1.2 precedes *p*.1.2 in document order, but is not a preceding sibling since the parent of *p*.1.1.2 (*p*.1.1) is different from that of *p*.1.2 (*p*.1). The efficiency of DLNs in quickly determining these relationships makes them ideal for query processing in XML DBMSs. There also exist schemes for efficient updating [23]. Observe that inserting a node high in the tree may force renumbering of an entire subtree, e.g., inserting a node before *p*.1.2 may force the renumbering of *p*.1.2.1.3 to *p*.1.3.1.3. But by using *fractional DLNs*, renumbering subtrees can often be avoided, e.g., the node inserted before *p*.1.2 becomes *p*.1.1/1, indicating that it is between *p*.1.1 and *p*.1.2 (in fact, renumbering can be avoided entirely [23]). This paper is orthogonal to fractional DLNs or other renumbering strategies.

Numbering a Temporal Document

In this section we extend DLN to number a temporal document or data collection. Extending DLN is complicated by the fact that the DLNs “change” over time, even for nodes that have the same temporal “identity.” We first develop a running example of changes to a document, and then present our new numbering scheme using the example.

Running Example

There are four basic edit operations: deletion, insertion, update, and subtree move (which can be modeled as a sequence of deletions and insertions). The edits could be specified in an update language like XUpdate, or applied directly in an XML editor. In the running example, we focus on the effects of each kind of edit, treating each edit (or combination of edits) as an individual *snapshot* or *slice* of an evolving temporal document. Assume a second slice shown in Figure 5, which has one deletion from the previous slice. The new slice is parsed and the (non-temporal) DLNs are assigned as shown in Figure 5. Note that the DLN for element <color> changes from *p*.1.1.2 in Figure 4 to *p*.1.1.1 in Figure 5.

Next a third slice is created by combining three edits: moving the <color> element subtree from <supplier> to <part>, updating “engine” to “wing”, and inserting the text “ACME” in the body of <supplier>. The third slice and its DLNs are shown in Figure 6.

Time Tunneling DLN

Now let’s elaborate our new numbering scheme, which we call time-tunneling DLN (TTDLN). We first describe the components of the TTDLN and then continue the running example. In TTDLN, a temporal XML document consists of two separate but related XML documents: a *history document* and an *item document*. (Additionally, the *current* document can be stored in order to improve the speed of queries as of “now,” but we do not discuss this optimization further in this paper.) The history document can be thought of as the “coalescing” of all of the slices into a history. Each node in the history has the following kinds of numbers.

- DLNH – The history document is an XML document ,so each node has a DLN.
- *Time stamped DLN* (*t*DLN) list – A *t*DLN is a DLN together with a timestamp.

The timestamp represents the lifetime of the DLN. Since the DLN may change over time, each node stores a list of *t*DLNs.

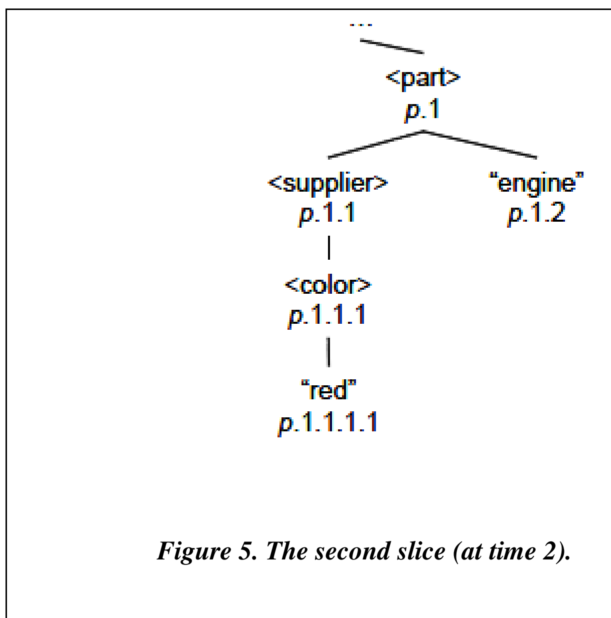


Figure 5. The second slice (at time 2).

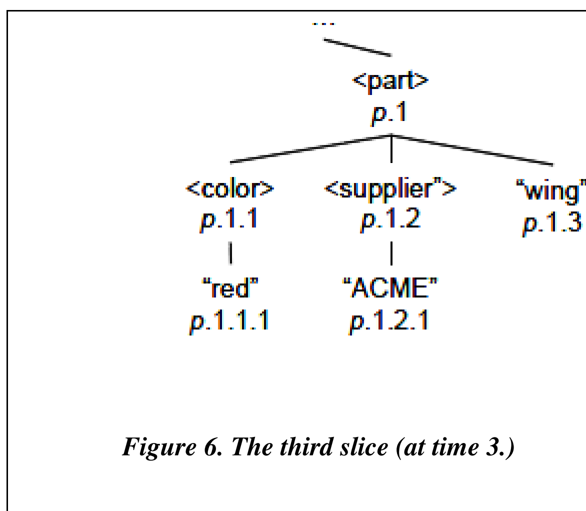


Figure 6. The third slice (at time 3.)

In addition to the history document, a temporal document has to identify *items*, which capture node identity over time, and *versions*, which are changes to the items [7]. In previous research, we described how to associate nodes that persist across various slices by *gluing* the nodes [9] [21]. When a pair of nodes is glued, an item is created. A version is a change to the item over time.

For simplicity, let's assume that every element is an item. The items and versions are represented in the *item document*. The item document has the following kinds of numbers.

- *item number* or DLNI – The item document is an XML document, so each node has a DLN.
- *item forward/backward chain number* – The item forward (backward) chain number is the next (previous) DLNI in the lifetime of the item (only present when an item moves within the history document).
- *version number list* – Each version number in the list is *v* [*b*, *e*] where *v* is the version number and [*b*, *e*] is the lifetime of the version.
- *parent's version number* – A parent's version number is *w*:*z* which is a range of versions of the parent to which this node belongs, from version *w* to version *z* (* represents an increasing latest version).

Figure 7 shows a temporal version of the initial slice in Figure 4. On the left-side of the figure is the item/version document. On the right side is the history document. (To simplify the figure, we have omitted the DLNHs for each node.) We have added pointers to highlight the different kinds of information in each node. The item corresponding to <supplier> has item number of "*i*.1.1", a version number "1 [1, now]"

and a parent's version number of "1:*". There are no forward/backward chain numbers. Each dashed line represents an annotation or connection between the item document and the history document. The connection means that element <supplier> records that it belongs to item "*i*.1.1", and item "*i*.1.1" records that it tracks the element with *t*DLN "*p*.1.1 [1,now]". Note that only elements are items. In general, any type of node could be an item, but in the running example, we assume that the schema designer wants to capture the history of elements rather than text or attribute nodes.

Next, we glue the second slice, yielding Figure 8. Changes are indicated in bold in the figure. The timestamp in the *t*DLN of element name="ACME" is terminated since it was deleted. A new *t*DLN is added for <color> (and descendants) since it changes from *p*.1.1.2 to *p*.1.1.1 starting at time 2. In the item document, a new version number is added for the item annotating <supplier>, and the version history of its left child is updated (the right child is in versions 1 and 2, but * represents the latest version so no changes are needed).

Next, a third slice extends the documents, as shown in Figure 9. Changes are shown in bold. New items were created for the moved subtree and item backward/forward chaining is utilized. The change to <part> creates two versions of the element. The first version has children present from within the time interval "[1, 2]". The second version is children present from "[3, now]".

There are four important points to observe about our two document approach. First, and most importantly, the history document retains the shape of each slice so extant queries can be directly applied. For example, the XPath query "//color [text () ="red"]" will locate color nodes that contain the text "red" across the entire history of the document. Second,

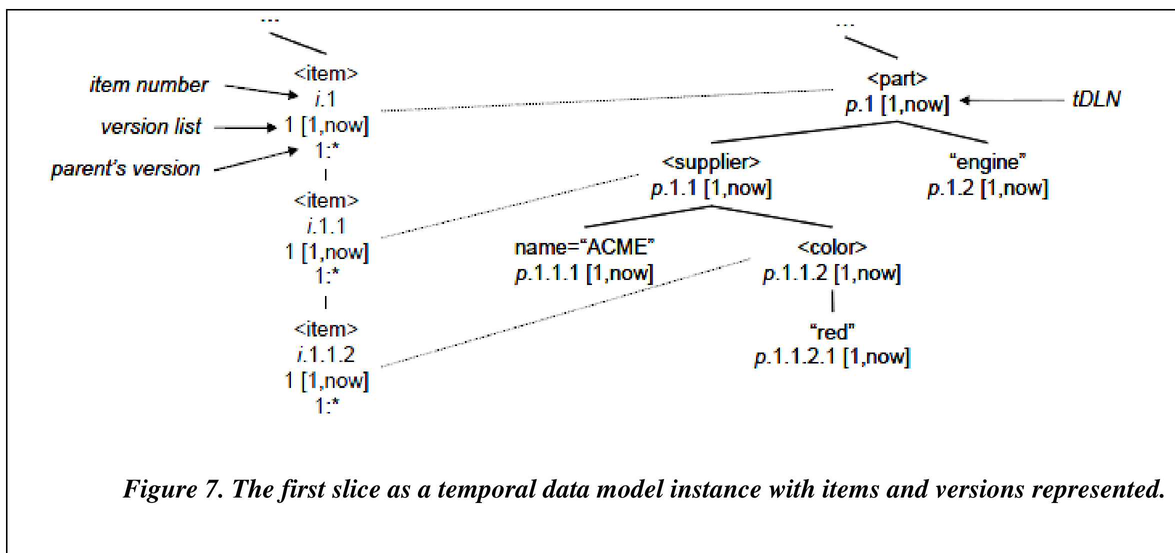


Figure 7. The first slice as a temporal data model instance with items and versions represented.

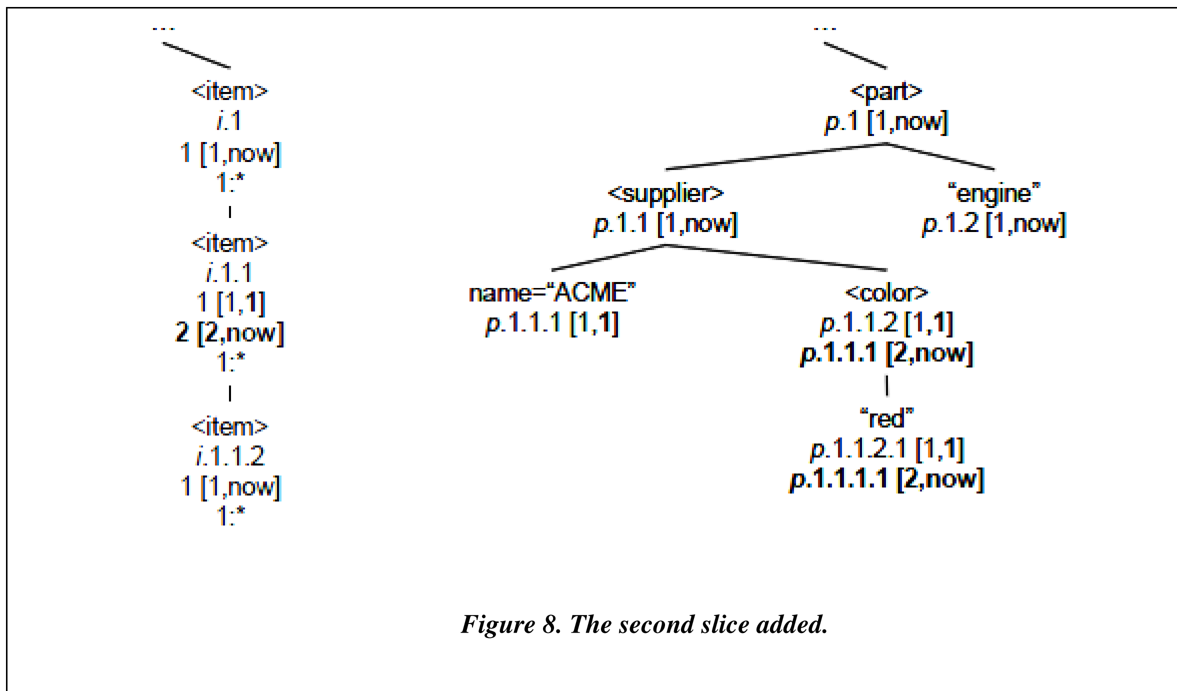


Figure 8. The second slice added.

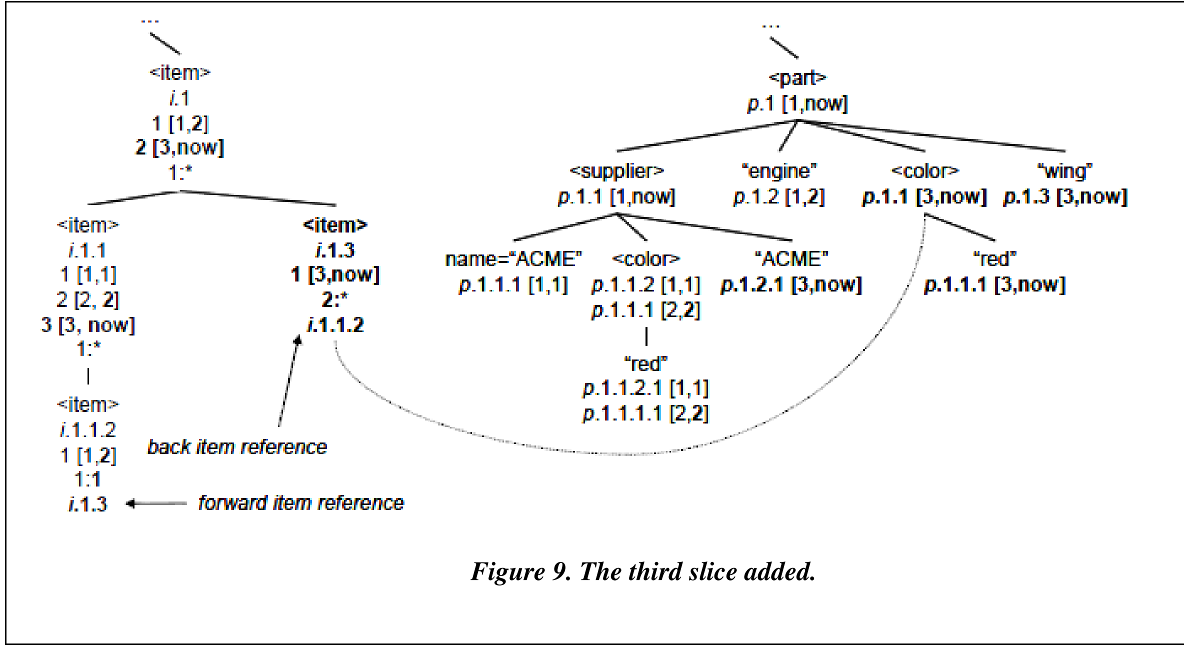


Figure 9. The third slice added.

the *t*DLNs capture document order and spatial positioning within each slice. Said differently, sequenced evaluation of temporal queries can be supported by simply using the *t*DLNs (and checking timestamp intersection). Third, the item document contains all information necessary to navigate along temporal axes to access past/future versions of elements [8]. Fourth, the history and item documents are append-only. Timestamps and other information are modified but not deleted. Fifth, the history document and the item document are XML documents, and so can be stored using existing native XML DBMSs, with one caveat: changes to an attribute's value will create multiple attribute children with the same name. So, a small modification to a native XML DBMS is needed to store the history document, e.g., in eXist, the HashMap of attributes must be changed to a MultiValueMap, which supports duplicate keys.

The item document has one node for every item. There is generally one item for every element in a history document. Nodes in the item document also grow in size over time as new version numbers are added to the version number list. Every edit of an element potentially creates a new version of the item connected to the element, so with C changes; the size of each node is bounded by $O(C)$. Thus, the item document size is also bounded by $O(C(N + M))$.

To get a better feel for the cost, we designed four experiments using the XMark benchmark [20]. We generated an XML document using a benchmark factor of 1, yielding a document approximately 1.1GB in size with approximately two million elements. We then wrote Java programs to randomly edit the document and update the history and item documents. The edit operations insert an element, delete an element, update the string value of an element, and move an element to a new parent. The

operations choose elements at random from the existing document to use in the edit, but only a single element is inserted, deleted, updated, or modified in each edit.

The first experiment tests the cost of the insert. We inserted two million elements, essentially doubling the size of the document. Each insertion also added to the history document, and created a new item in the item document. Figure 10 shows the result. The Document line in the figure shows the size of the XMark document (a single slice), which continues to grow as elements are inserted. The History Document line plots the size of the history document, while the Item Document line plots the size of the item document. The history document is always slightly larger than the current document since each node in the history document includes *t*DLNs. The item document starts small and grows very slowly.

Experiment two measures the cost of delete. Two million elements were deleted. Figure 11 shows the result. The current document shrinks to nothing. The history and item document stay the same size (only timestamps are updated as elements are logically deleted). Experiment three measures the cost of update. Two million updates were performed. The results are shown in Figure 12. The current document stays the same size. (The size actually varies, but the variation is within a few MBs, not visible in the graph.) Each update creates a new text node in the history document, so it essentially doubles in size since text strings are the largest component in the document. No items are created, but new versions are. The new versions add only a small amount to the item document.

The final experiment measures element move. Two million moves were performed. The results are shown in Figure 13. The moves change the size of the XMark

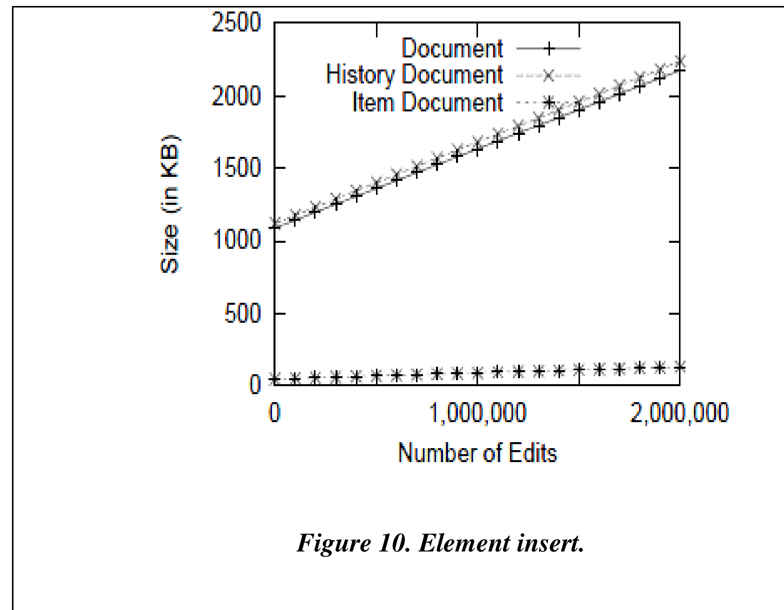


Figure 10. Element insert.

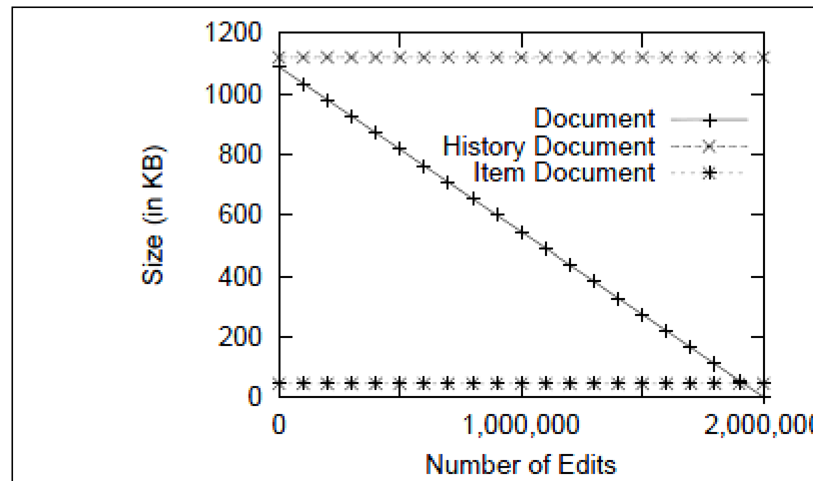


Figure 11. Element delete.

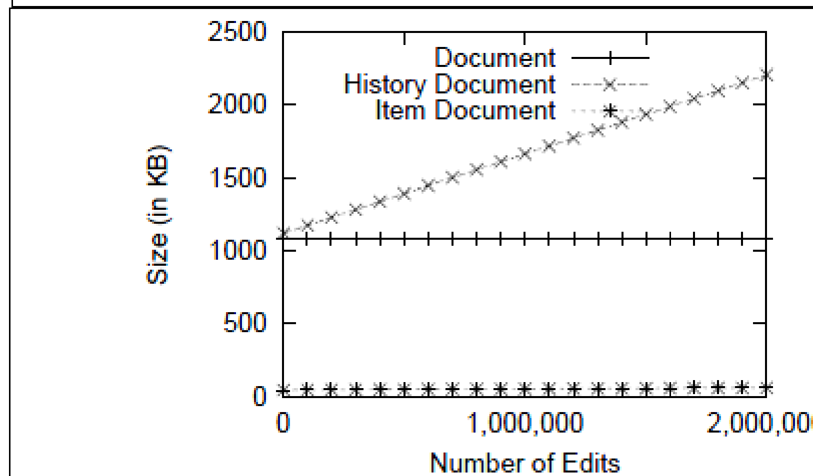


Figure 12. Value update.

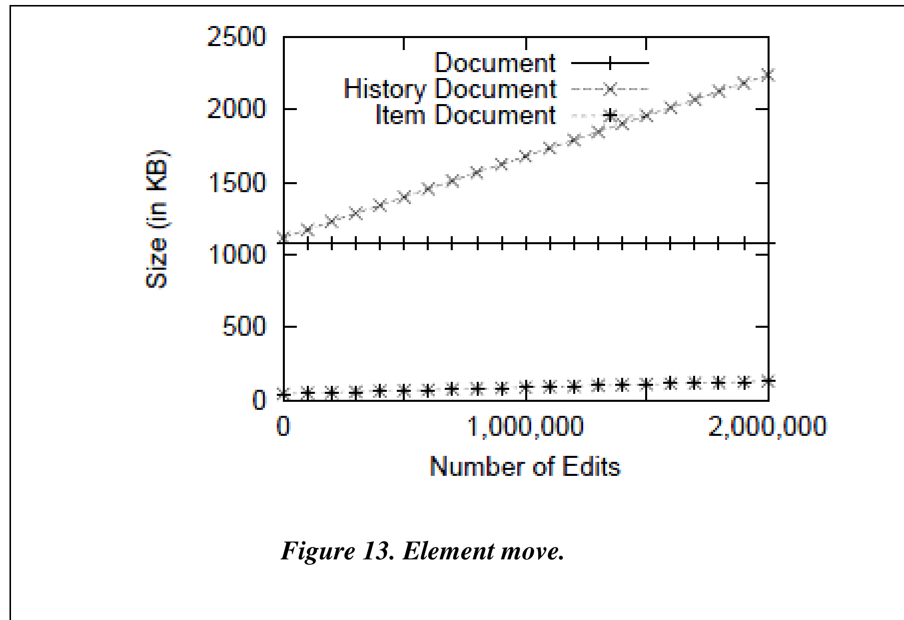


Figure 13. Element move.

document only by a small amount (not visible in the graph). Each move creates a new item and updates the forward/backward chaining, so the item document doubles in size. The history document also doubles in size as each move is a logical deletion and a physical insertion.

The experiments confirm the analysis that the space cost is linear in the size of the original slice and the number of changes over time.

XUpdate

XUpdate is a simple data modification language for updating an XML document. In XUpdate we can insert, delete, and update elements, attributes, and text content. We focus on element operations in this section.

When an element is inserted as illustrated in Figure 9, at most one node is added to the history document and one item to the item document. (A new item or history node

is not added if the item previously existed in the history, but was deleted.) For the item/version document, the new item may create a new version of its parent, but only the parent's version number list needs to be updated (all the other children include the latest parent version automatically). So at most, one node is added and one node is modified. For the history document, if the inserted history node is the last child, the change is simple, only one node needs to be added; no other nodes change. However, if the inserted node is not the last child, for every following sibling, the insertion will force the renumbering of the subtree (assuming that fractional DLNs are not used) rooted at the sibling, adding a new *t*DLN to each node in the subtree. So for the history document, the cost of an insertion varies between O (1) and O (N) where N is the combined size of the affected subtrees. Using fractional DLNs can lower this cost substantially, but is orthogonal to this paper.

The deletion of an element terminates the lifetime in a *t*DLN for that element and trigger the renumbering of each subtree rooted at following siblings (just as in the non-temporal case), adding a new *t*DLN to each node. For the item document, the deletion terminates the lifetime of the current version of the item, and creates a new version of its parent, affecting at most two nodes. So, the cost is similar to insertion.

The modification of an element creates a new version of the item to which that element corresponds, changing at most one node in the item/version document. The modification also changes the name, text, or attribute value in the history document, but does not impact the DLNs, so the modification adds one new node to the history document (text and attributes are generally modeled as nodes), but does not change any DLNs.

In summary, XUpdate operations can be applied to a temporal XML document. The cost of an operation is similar to that of the non-temporal case.

Query Evaluation

The additional space needed for node numbers in the TTDLN scheme pays off in evaluating queries. In this section, we consider sequenced versions of DLN operations, as well as non-sequenced extensions that encompass both version- and item-related queries. Just as with their DLN counterparts, the temporal operations are implemented by manipulating the numbers without referring to the actual document.

Sequenced Spatial Relationships

DLNs have predicates to determine whether a specific relationship holds between two nodes. There is one predicates for each axis: parent, child, ancestor, preceding, etc. As canonical examples, we consider only two such predicates.

- IsParent(*X*, *Y*) – Is node *X* a parent of node *Y*?
- IsFollowing(*X*, *Y*) – Does node *X* follow node *Y* (in document order)?

  The sequenced version of these predicates determines whether the relationship holds at every point in time for a given time interval from [*b*, *e*].

- isParentSeq(*X*, *Y*, *b*, *e*) – Is node *X* a parent of node *Y* in the history document at each time between time *b* and *e*? We can evaluate this by examining the *t*DLNs in each node. Let *Z* be a *t*DLN in node *Y* that intersects [*b*, *e*]. Then, for every *t*DLN, *W*, in node *X* such that the timestamp of *W* intersects that of *Z*, test whether isParent (*W*, *Z*) holds. Essentially, this is the same logic as the nonsequenced case, except for the additional processing of the timestamps.
- isFollowingSeq(*X*, *Y*, *b*, *e*) – Does node *X* follow node *Y* (in document order) between times *b* and *e*? We can evaluate this by utilizing the *t*DLNs of *X* and

Y. Let *Z* be a *t*DLN in *Y* that intersects [*b*, *e*]. Then, for every *t*DLN, *W*, in *X* such that the timestamp of *W* intersects that of *Z*, test whether isFollowing (*W*, *Z*) holds.

Unsurprisingly, the other sequenced predicates have a similar form.

Sequenced Constructors

DLN operations also include constructors that build children or parents relative to a node. The sequenced versions of these constructors are similar to their non-sequenced brethren.

- getParentSeq(*X*, *b*, *e*) – Get the parent of *X* in the history document at each time between time *b* and *e*. We can quickly find the parent by examining the *t*DLNs of *X*. Let *Z* be a *t*DLN in node *X* that intersects [*b*, *e*]. Then, getParent (*Z*) is in the list of sequenced parents.
- getChildren(*X*, *b*, *e*) – Get the children of *X* in the history document at each time between time *b* and *e*. Let *Z* be a *t*DLN in *X* that intersects [*b*, *e*]. Then, Add getChildren (*Z*) to the list of children returned by the constructor.
- getSliceDLN(*X*, *t*) – Get the DLN in the slice of node *X* at time *t*. Extract the DLN from the *t*DLN in node *X* at time *t*.

Other constructors (getAncestors, getDescendants, etc.) are variations of the above constructors.

Non-sequenced Operations

The item document is used primarily for non-sequenced operations. A non-sequenced operation tunnels through time between different states of a document.

- nextVersion(*X*, *v*) – Extract the time of version *v*+1 of history node *X*. We can evaluate this by first following the connection to the item for *X* (node X stores

the item number or DLNI for the item). From the item, we extract the version number list. Next, we traverse the list until we locate version *v*+1. If no version *v*+1 exists, check whether the item has a forward item reference. If not, there is no version *v*+1 for this item. Otherwise, follow the forward reference to the first version of the next item. Assume that the version starts at time *t*. Return *t*. (Alternatively, we could build the text of version *v*+1 by visiting each child of the item to determine whether it is a member of version *v*+1. For the item and its children that are members of the version, we traverse the connection back to the corresponding history nodes to obtain the text of the version.)

- previousVersion(*X*, *v*) – Similar to nextVersion(*X*, *v*).

Since the item numbers are essentially DLNs, all of the (non-temporal) DLN operations can be applied to item numbers.

- parentItem(*X*) – Return item *P*, the parent of item X. The item number for *P*, is a prefix for that of *X*.
- childItems(*X*) - Return the list of items that are the children of item *X*. Find the smallest *k* such that there is no item with item number *X.k*, then build a list of items *X.1* through *X.k*-1.

Conclusion

Prefix-based numbering is a node numbering scheme that promotes the efficient querying of XML data. In prefix-based numbering, a node is numbered using its parent's number as a prefix. The scheme cannot be applied to a temporal data collection since parent node numbers (the prefixes) change over time. In this paper, we present a new node numbering scheme that we call time-tunneling dynamic level numbering (TTDLN).

Prefix-based numbering remains at the core of TTDLN, but the new scheme splits a document into two pieces: a history document, which is the history of the XML document over time, and an item document that keeps track of node identity over time and node versions. In the history document, each DLN is transformed into a list of time stamped DLNs. In the item document, each DLN becomes an item number, a list of version numbers and timestamps, a range of parent versions, and a forward/backward reference to a new/previous position of the item in the history. We show how TTDLNs can be maintained in XUpdate operations and how they can be used to support sequenced and non-sequenced temporal query operations.

Chapter 3

INTRODUCTION

Introduction to XML

Extensible Markup Language (XML) is a language used to describe information in documents for communication. The main feature of XML is to provide a mechanism to represent information in a hierarchical manner, using tags and thereby making it self-descriptive. XML can represent data pertaining to diverse data sources, such as structured, semi-structured, relational databases and object repositories.

The basic unit of an XML file is called a node. A node can be of type element, text or attribute. An element node is represented using a ta and it has a name. The attribute node has a name and a value. A text node has only text. The attribute and text nodes are each associated with an element node. For example, let us consider a sample XML file (***Figure 14***). As we can see in ***Figure 14***, the information is structured in a hierarchical manner. The node named bookstore is called the root element of the document. It has one child element node named book. The book element node also has seven child nodes out of which there are five element nodes, namely, title, author, year, price, and pub, as well as two attribute nodes called category and genre. The element named title has two child nodes, the first being ‘lang’ which is an attribute node, and the second being ‘Harry Potter’ which is a text node. So, the element book resembles a record in any relational database. The important thing to note is that XML information is aimed at representing information in a structured manner and the structure itself provides the relation and meaning to the information.

```
<bookstore>

<book category="CHILDREN" genre="ADVENTURE">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2011</year>
  <price>40.99</price>
  <pub>Bloomsbusy Pub.</pub>
</book>

</bookstore>
```

Figure 14. An XML file.

Introduction to XQuery and XPath

Because XML follows a standard structure for representing information, we can also query for required information by filtering over the data present in it. This may be performed using XQuery, a query language to query information from an XML document (source). XQuery is the counterpart of SQL in the context of XML-based databases or resources. There is also the need for representing various sections of information present in XML using an expression language. This is done through the use of XPath expressions. XPath provides the syntax to represent elements or parts present in an XML document. It uses path expressions to navigate through the nodes present in an XML document. For example, to represent the node title in the XML file present in ***Figure 14***, we use the XPath expression /bookstore/book/title. XQuery is built on XPath expressions. It uses a type of expression called FLOWR, which is a sequence of statements representing For, Where, Order by, and Return. This expression is similar to an SQL select statement with where and order by clauses. For example, to query all the titles of the books whose price is less than $30, we use the XQuery code present in Figure 15.

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

Figure 15. XQuery code.

Node ID and Numbering Plan

It's not always convenient to identify a node based on its path using XPath expressions. An XML store may contain many documents that might have nodes with the same name. For this reason, implementations of XML stores provide nodes with a unique identity using a property called node id. The node ID is essentially a number. There are several numbering plans proposed to define the node IDs. The Dewey level numbering (DLN) system is one such plan that provides decimal numbers as node IDs. Stated differently, the DLN is based on the Dewey decimal classification system.

This type of numbering helps in representing the level (or depth) of a node in the document as well as the relation between two nodes of the document. For example, the node with node id 1.1 would be the parent of the node with id 1.1.x (where x represents the order of this node among the children of the parent node (with node id as 1.1)).

Problem and Proposed Solution

Currently, XML data storage and querying is limited to non-temporal needs. But consider an application that might require data relative to a specific period in an environment wherein data changes are quite frequent. For example, if an educational institution wishes to check for the instructor's details for a course, say CS 4444 during the year 2001, it is not possible to query the current XML store from a temporal

perspective. Such situations form the basis for the need of an index that can associate nodes or data in general to time.

We propose a solution that would add temporal features to the existing XML nodes. These temporal features primarily include the following properties:

- Version: This property keeps track of the version numbers. It starts from 1 (one) for the oldest version and counts by increments of one for every new version of the node.
- Timestamp: This property provides details about a node's creation and deletion or modification. In general, it provides the details about the life time of a node.
- Item: Some node information needs to be persistent over time. Such nodes are termed as items. So, this property provides a reference to details about the node, which are persistent across various revisions of the document.

This plan forms the basis for the need of a new axis that is composed of the above mentioned properties namely version, timestamp, and item.

The transaction- time axis (tt-axis) is proposed to satisfy this requirement. The proposed tool is useful for storing and referencing information that is maintained through a period of time. The tt-axis provides a temporal axis for XML data in addition to other axes like the ancestor, descendant, etc., which are non-temporal axes. Thus, with the addition of the new tt-axis, nodes can be queried or filtered based on a specific time period. For example, to retrieve all of the nodes holding the information about the instructor for the course CS 6700 during January 2006, the query may be formulated to

include the time period in addition to the general xpath information (Example doc ("Course information.xml")/course[@name="CS 6700"]/instructor (time=January 2006)). The query is then executed by using the tt- axis in addition to the generic xml axis. First, the results of the generic axis search query are determined. Next, these results are filtered using the tt-axis for matching nodes that are active during the specified period (January 2006).

Since the DLN numbering plan does not have a reference for the newly proposed temporal properties, we propose a new numbering plan to accommodate the temporal properties and provide support in the implementation of the tt-axis described above. This new numbering plan extends the existing numbering plan, namely called DLN. The new numbering plan is named as item version timestamp level numbering (IVTLN). As the name suggests, IVTLN consists of version, timestamp, and item information in addition to the information provided by the existing numbering plan DLN.

Chapter 4

OBJECTIVES AND METHODOLOGY

Overall Goal

The goal of this project is to create a representation of items and versions to support efficient evaluation of the tt-axis. We evaluated system efficiency with respect to the transaction-time and space usage. So, the trade-off considered here is the time taken for executing temporal queries vis-a-vis the additional memory utilized for the purpose of constructing the transaction-time index.

This project provides the option to store data in a time sensitive manner, including storing data based on record or creation times, and updating the same using timestamps and versions. This primarily involves the creation and implementation of a platform for time-based indexing and its associated data structures. The final product helps in querying the database from a time period perspective, so, internally a node's information is retained in spite of it updates through time.

XML Database Considered

The XML-based database we chose chosen for our implementation is the Exist Db developed by Wolfgang Meier. eXist-db is an open source database management system built using XML technology based on the Java platform. It stores XML data according to the XML data model and features efficient, index-based XQuery processing.

Tasks

The following tasks were identified in the planning and implementation of the tt-axis for eXist db.

- Create and extend items: As each slice of the document is parsed, we need to recognize the nodes that persist over time. We name such nodes as 'items' and the process as 'itemization of nodes.' Currently, all element type nodes are treated as items, and we may refine the same or extend the same to other node types such as text and attribute nodes.
- Create versions: Every time a node is parsed, we check if the node is present in the database. If found, we compare the existing node with the new node. If there is some difference in their properties, we classify the new node as a new version of the existing node which itself would be the previous version of the new node. Else if none of the existing nodes match, the new node is assigned a version property that makes it the very first version.
- Create timestamps or lifetime: Every node is assigned a life time or timestamp property that contains the node's creation time, deletion or modification time. For a new node, the deletion or modification time is 0 or until changed, denoting that the node is currently active. Modification of a node gives rise to a new version. So, the old version's deletion time is updated to the current time which is also the creation time of the new version.

Work Plan

In order to complete this project in a timely manner, we developed a plan to implement tt-XPath (Transaction-Time XPath) in eXist. Our plan included studying the workings and backend of the eXist database. It also included the study of various custom data structures and techniques followed during the development of eXist database source code.

Our plan also encompassed adding our modules to eXist source. This includes the analysis of scope and support for additional modules by eXist database source code. Additionally, the plan included implementation of new features and identifying the impact of such implementation on the existing features.

Criteria

The criterion followed during the process of tt-axis implementation was first to construct a plan to describe a new model for indexing, new query support, and xupdate changes, and second to implement code modules based on the proposed plan.

Chapter 5

NEW NUMBERING PLAN

Introduction

The current plan was designed with a view to provide and support temporal index to XML data. This includes creation of the tt-axis which helps in querying data pertaining to a specific period of time (temporal data). This axis also helps in setting a base (pivot) time slice pointer to a time period and then traversing the past and future versions of that node.

The result is specific to a particular time slice (period). Thus, data that has been updated can be stored and viewed for future reference. Consequently, this axis allows viewing of old data that was modified during subsequent versions. Hence, the axis provides a mapping between data and its time stamp.

Basis for the New Index

The new index is based on transaction- time of a particular transaction. This can be the write operation, which may involve the creation of a new node, the modification of an existing node, or the read operation, which involves reading information from an existing node,

Basis for and Description of New Numbering Plan

A new numbering plan was devised based on the index mentioned above.

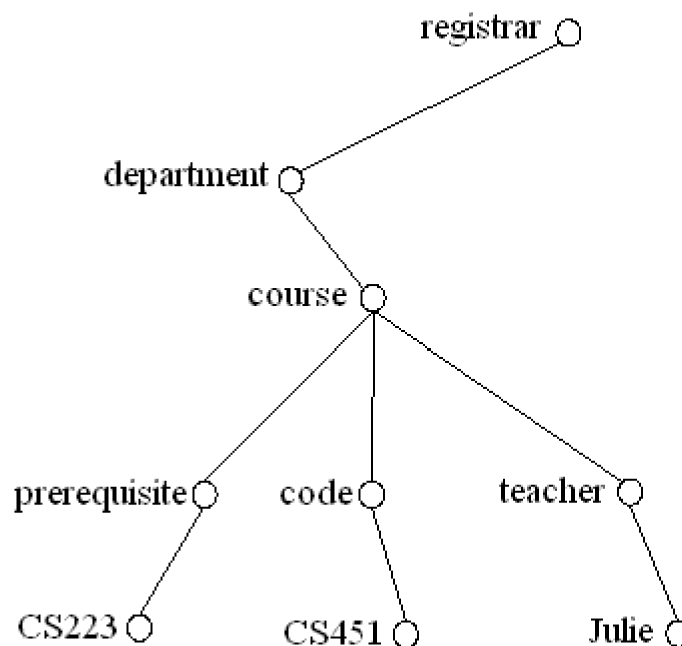


Figure 16. Course Info (tree structure).

Consider the tree in Figure 16; the new numbering plan is created including the index in 4.2 as follows. The new numbering plan is an extension to the Dewey level numbering (DLN) plan followed in eXist db.

Existing Numbering Plan

Currently, the numbering plan followed in eXist db is based on node IDs using the DLN, as shown in ***Table 1***.

Node	Node Ids
Registrar	1
Department	1.1
Course	1.1.1
Prerequisite	1.1.1.1
Code	1.1.1.2
Teacher	1.1.1.3
cs223	1.1.1.1.1
cs541	1.1.1.2.1
Julie	1.1.1.3.1

.

Table 1. DLN for Course Info.

New Numbering Plan

As stated above, in this project, we propose a new numbering plan to implement the tt-axis. This new numbering plan extends the existing numbering plan, DLN. The new numbering plan is named item version timestamp level numbering (IVTLN). As the name suggests, the IVTLN consists of version and timestamp information in addition to the information provided by the existing numbering plan. So, IVTLN includes a time stamp and a few bits representing version and schema change identifiers.

Timestamp

This section of the node ID may have three parts, namely:

- Start timestamp: the time when the node was created.
- Last read timestamp (optional) : time when the node was last read.
- End timestamp: time when the node info was modified.

Representing Timestamps

We use the format yyyymmddhh as the means to represent the timestamp of a transaction. Thus, currently nodes can be compared up to the level of hours of their creation or modification.

For example, assuming that the tree provided in Figure 16 above was created on 8th January 1999 at 21:00 hrs. The start timestamp of all the nodes present in the tree would be < yyyyMMddHH>, i.e., <1999010821>, and the corresponding end timestamp would point to 0, meaning currently active or alive.

Bits Representing Schema Change and Version Identifiers

In addition to the time stamp, the new numbering plan includes bits representing the occurrence of schema change (0/1/2 – no/yes after/yes before respectively) and

version range (m-n, where m, n are the current and most recent version numbers of the current node, respectively).

For example, if the root node (registrar) in the previous example (Figure 16) has no schema change and if this is the first version of this node, the new node ID is Dewey no, <time stamp>.Schema change bit. (m-n version range), where m – represents the current version of the node and n - represents the version number of the latest version. So, it is 1<199901, 199901, until changed>.**0**.(1-1) and if there were a schema change and this were the second version of this node, the new node id would be 1<199901, 199901, until changed>.**2**. (2-2). (Note: Here, the schema change bit of 2 indicates the current version was created after a schema change).This would also have an entry in the schema change tracking table.

Old Node id	New Node id
7.2.1<199601, 199901, 199901>.**1**.(1-1)	1<199901, 199901, until changed>.**2**.(2-2)

Table 2. Schema Change Tracking Table.

Note in Table *2*, the schema change bit is 2 only for the node version created immediately after a schema change. Also, the schema change bit is updated to 1 from 0 in the old node ID so as to indicate that there was a schema change after this version.

Updating a Node

A node that was changed gives rise to a new version and is assigned a new label (a new time stamp, version bits, and schema change bit). The previous version is

considered to be logically deleted, so its deletion time stamp is updated with the creation time stamp of the new version.

Thus, for example a node (1<199901, 199901, until changed>.0.(2-2)) that was updated in September 2000 would result in the following steps:

1) Create a new node with the node id as 1<200009, 200009, **until changed**>.0.(3-3), assuming there was no schema change.

2) Update the node ID of the old version to 1<199901, 199901, **200009**>.0.(2-**3**). Here, we update the delete time stamp as well as the maximum version range value. (Changes are represented in bold).

Chapter 6

SCHEMA CHANGE

Description

When a node is moved into a different document or as a child to a different parent, we treat such a change as a schema change or transition. Since our current numbering and indexing plan has no method to track such changes, we propose a technique to satisfy this requirement. This technique acts as an addition to IVTLN. Schema change may be explained with the help of the following cases.

No Schema Change

Some transactions or changes do not affect the schema of the document. Following are some of the cases with description which fall into this category.

Case where the child version changes. Let's consider Figure 17, here a teacher's name for a course was updated from Miguel to Julie during January 2001.

From the figure above we can observe that the node value was changed from Miguel to Julie for the node Teacher. The change has occurred during January of 2001. So we consider the node with value as Miguel active from its creation time which is December 1999 to its deletion (logically) in January 2001. Figure 17 shows the change in more detail. The following tables (Table 3 and Table 4) provide the node id details for both the versions corresponding to the root node (registrar) and the changed node (Miguel to Julie).

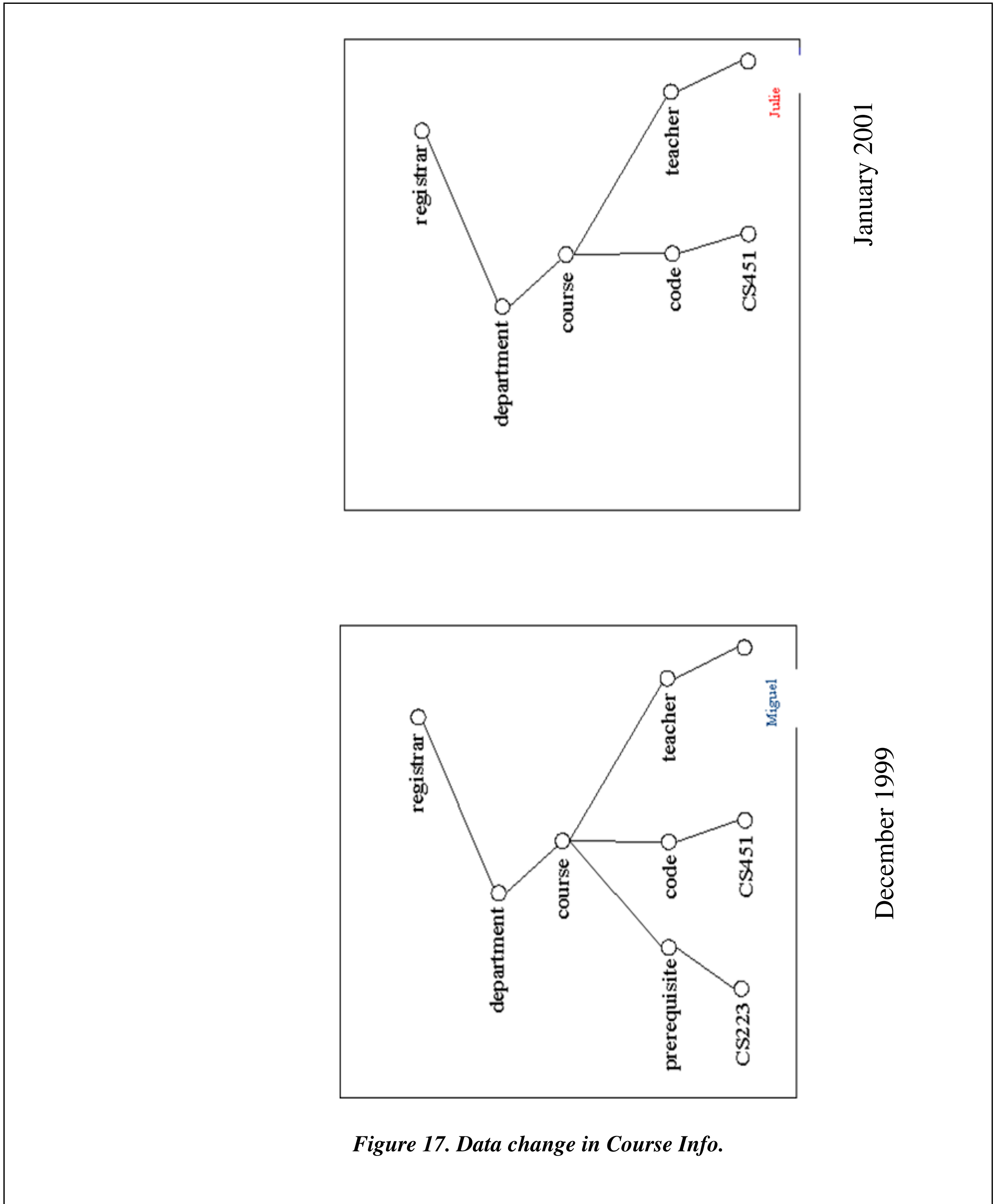


Figure 17. Data change in Course Info.

Please not the change from the instructor name from ‘Miguel’ to ‘Julie’ during the years 1999 to 2001 respectively.

Node	Node id
registrar	1< 199912, 199912,until changed>.0.(1-1)
Miguel	1.1.1.3.1<199912, 199912 , until changed>.0.<1-1>

Table 3. Course Info (Instructor Name) Details During Fall 2000.

Node	Node id
registrar	1<199912, 2000101,until changed>.0.(1-1)
Julie	1.1.1.3.1< 2000101, 2000101, until changed>.0.<2-2>

Table 4. Course Info (Instructor Name) Details During Spring 2001.

As we can see, the root node and other nodes are not affected by the update; hence, their IDs remain unchanged, whereas the child of the node teacher node was changed from Miguel to Julie.

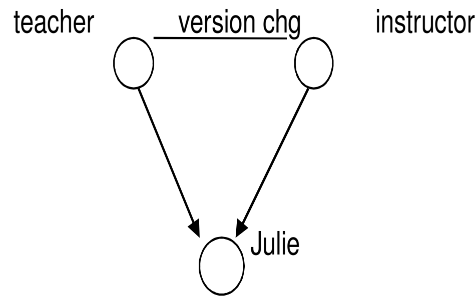


Figure 18. No schema change example.

A new node (Julie) is created for the new version with a new time stamp and version range. The old (Miguel) node's ID is updated to include the delete time stamp. The version range is updated from 1-1 to 1-2 as following,

Miguel - 1.1.1.3.1<199912, 199912, 200101>.0. <**1-2**>.

It should be noted that the version range of the child node Julie is not altered. Figure 18 shows the case wherein the parent version changes and the new version retains the child of the old version.

Node	Node id
Teacher	1.1.1.3<200012, 200012, 200101>.0.<1-2>
Instructor	1.1.1.3< 200101, 200101,until changed>.0.<2-2>
Julie	1.1.1.3.1<200012, 200101,until changed>.0.<**1-1**>

Table 5. DLN Changes to Course Info.

In Figure 19, we see that the teacher node undergoes two version changes. We note that after the first change from teacher to instructor, its child node (Julie) was deleted, and the same child node (Julie) was added as a child after the second change. In this case, we treat it as update (to the child node Julie) since an intermediate version of the parent (i.e., instructor) does not have a child.

While updating, if the old node is read first and then updated, we need to update both the read and delete time stamps, whereas if it is a blind update, only the delete time stamp is updated.

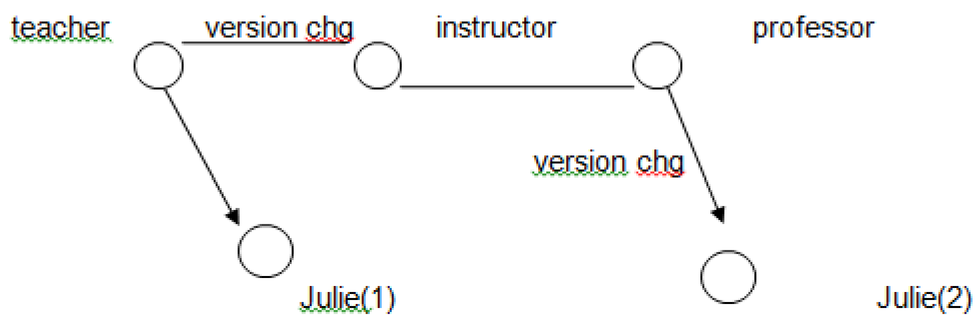


Figure 19. No schema change (appending a child node after its deletion).

Node	Node id
Teacher	1.1.1.3<200012, 200101, 200101>.0.<1-**3**>
Instructor	1.1.1.3< 200101, 200106, 200106>.0.<2-**3**>
Professor	1.1.1.3< 200106, 200106,until changed>.0.<**3-3**>
Julie(1)	1.1.1.3.1< 200012, 200012, 200101>.0.<**1-2**>
Julie(2)	1.1.1.3.1<200106, 200106, until changed>.0.<**2-2**>

Table 6. Numbering Change Due to Schema Change.

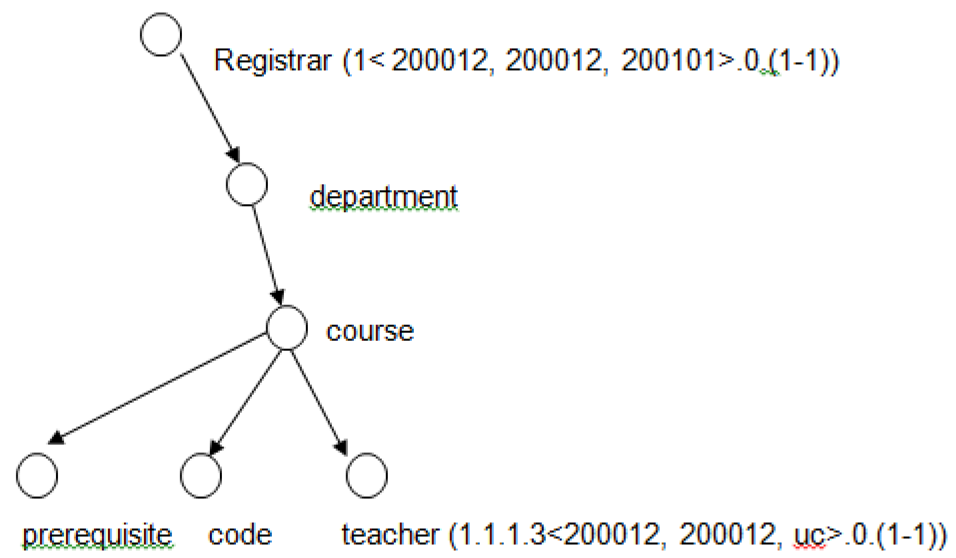


Figure 20. Before schema change.

Schema Change

When a node is moved as child to a different node or when a node is missing in some of the intermediate versions of the document while still retaining its information, we note such changes as schema or plan change since the alignment of the node with respect to its document has changes. In other words, it becomes a child of a different node(s).

The following cases provide an example and description about the effect of schema change and the necessary steps to update the same in our index and numbering plan.

Let's consider the case wherein schema change occurs. Consider the Couse Info tree before schema change, as in *Figure 20*, and the same tree after schema change, as in Figure 21.

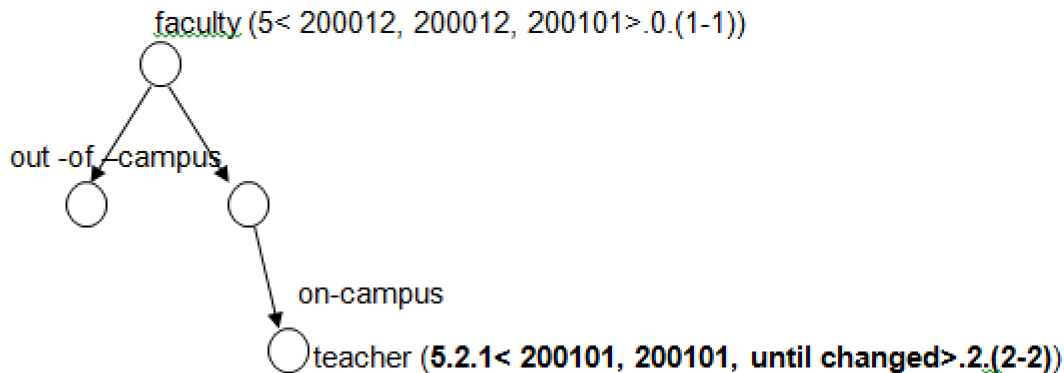


Figure 21. Course Info after schema change.

In this case an entry is added in the schema change tracking table to represent the change as shown in Table 7.

Old Node Id	New Node Id
1.1.1.3<Fall 2000,Fall 2000,Spring 2001>.1.(1-1) (Note: Change in schema bit from 0 to 1 after the change)	5.2.1< Spring 2001, Spring 2001,until changed>.2.(2-2)

Table 7. Change in Node ID due to Schema Change.

As shown in Table 7, after the schema change, the schema change bit of the old node is set to 1 and the schema bit of the new version would be set to 2. So, a schema change bit of 1 indicates that a schema change has occurred after the current node. Similarly, a schema change bit of 2 indicates that the current node version was created after a schema change. An entry is added in another table called the future access table (Table 8) which stores the latest version of numbers of all the nodes for that schema. Said table helps in avoiding the overhead of updating all the node IDs with the recent version identifier across different schemas (if there were some schema changes).

Node id	Maximum version number (with respect to schema)
1.1.1.3	2
5.2.1	1

Table 8. Future Access Table.

Chapter 7

QUERYING

Querying for Versions

Referring to the examples in Figure 17 and Figure 18, if we need to query the teacher node version(s) during the time slice (December 2000), we first check for the node IDs of all the teacher nodes whose time stamp range (duration between creation and read timestamp / deletion time stamp (if defined)) match the time slice requested in the query. So for a query like

/tt-past slice (‘Dec 2000’) /registrar/department/course/teacher

the node with id 1.1.1.3< 200012, 200012, 200101>.0.(1-2) is returned. If the next (future) version of the previous resultant node is requested, we check for the node with id 1.1.1.3<200101, 200101, until changed >.0.(2-2)

Note the check here is a simple match with respect to node id (1.1.1.3) numbers and version numbers (2-2), so the timestamp is ignored. If found, the same is returned; else, a schema change might have occurred, so a check is performed over the schema change tracking table (as in the table above) taking the current sliced node’s ID as the key.

If an entry is found with the node ID as the key in the schema change, tracking table then its corresponding value is returned as the resultant node id (**5.2.1< 200101, 200101, until changed>.1. (2-2)**). Here, we just check for the node number which is 5.2.1 and the version range which is 2-2.

Hence, navigation between nodes of the same type is performed first by assuming there was no schema change, and only on failure of this case is a schema change considered and checked with the help of the schema change tracking table to get the required result (node ID). The same strategy can be applied for querying (navigating) between future as well as past versions of a node.

Querying for Axis Data

Querying for information based on the axis is performed using the steps below.

Axis Description

The ancestor axis points to the set of nodes that are ancestors to the current node. This axis starts from the parent of the current node at its root. The descendant axis points to the set of all the descendant nodes of the current node.

Implementation of Axes Using the Current Plan

The current numbering plan with transaction- time index helps in retrieving or loading information for the two axes as follows:

Ancestor axis: Let the current node id be 1.1.3.2.1<200009, 200009, until changed>.0.<2-2>. Then, we compute the list of ancestor node numbers as

1.1.3.2 (Parent of the current node) …...... Level 4

1.1.3 …..Level 3

1.1 ....Level 2

1 (document root). ….................................Level 1

Problem: At each level we get a list of node versions. For example, in the fifth (5th) level we may have many nodes with node number as 1.1.3.2 and the IDs as

1.1.3.2<200009, 200009, uc>.0.<10-10>

1.1.3.2<199901, 200009, 200009>.0.<09-10>

1.1.3.2<199801, 199801, 199901>.0.<08-10>

1.1.3.2<199701, 199701, 199801>.0.<07--10>

1.1.3.2<199601, 199601, 199701>.0.<06-10>

Solution: We consider only those node versions whose timestamp range (the time between creation and delete time stamp, if the delete time stamp is defined) contains the creation time of the current node. Since the current node creation time stamp from its node id 1.13.2.1<200009,200009,uc>.0.<2-2> is 200009, we select only those ancestor nodes at the above level that contain 200009 in its slice range (creation time – read /(delete time stamp -1)). Thus, from the above set, we filter out the node versions and get the required node version(s) which is/are ancestor(s) of the current node which in this case is 1.1.3.2<200009,200009,uc>.0.<10-10>.

Descendant axis: Let us consider the current node id as 1.1.3.2.1<200009,200009,uc>.0.<2-2>. We may then derive the node numbers of the descendants as

1.1.3.2.1.X (X can be 1,2,3, ...)

1.1.3.2.X.X

1.1.3.2.X.X.X

and so on, depending on the depth of the document.

Problem: Even here, we get a set of nodes at each level that have to be compared with the current node version to retrieve the required version matching descendants of the current node.

Solution: Here, we compare the slice range of the current node which is 200009-200009 since the delete time stamp is not defined (uc). We check whether the creation time stamp value of the descendant node(s) is within the time slice range of the current node. We then get only those descendant nodes that belong to the current node version.

Note: While checking for axes, the schema change is not considered since we treat the schema change as a version change.

XPath

XPath is the syntax that represents elements or parts present in the xml document. It uses path expressions to navigate through the nodes present in an xml document. XPath was defined by the World Wide Web Consortium. So Xpath provides a way to navigate through the XML tree based on a given criteria.

Tt-XPath

The transaction- time Xpath (or tt-Xpath) is an extension to Xpath. The tt-Xpath adds new axes and methods in addition to the ones provided by Xpath based on the transaction- time axis. So, tt-Xpath provides support for queries based on the following newly created axes based on tt-axis.

Tt-XPath Axes

In addition to the existing axes, the following are provided by tt-XPath.

tt-Past - to get all the past versions of the current node.

tt-Future- to get the available node versions.

tt-Past-known- to get only the known past versions.

tt-Future-known- to get only the known future versions.

Let the current id be 1.1.3.2.1<200009, 200009,200019>.0.<23-50>. From the node id we know that there are a total of 50 versions of the current node and the current version is the 23rd version which implies that there are 22 past versions. Now, we fetch the previous versions as stated in the paragraph below. We consider only the version range and node number and ignore the timestamps for version comparison.

The node versions of the following form are considered and taken into the result, **1.1.3.2.1**<X,X,X>.0.<**(n-1)-50**> where X can be any time slice and n is the current version number which in this case is 23. Thus, we consider all previous version numbers, which is from 22 to 1.

1.1.3.2.1 <X,X,X>.0.<22-50>

1.1.3.2.1 <X,X,X>.0.<21-50>

.

.

1.1.3.2.1 <X,X,X>.0.<2-50>

1.1.3.2.1 <X,X,X>.0.<1-50>

While traversing through the previous versions, we check for a schema bit of 2. Once we detect the same (as in the example below), we infer a schema change in the past (previous to the node which contains the schema bit as 2). We then check for the entry in the schema change table so as to get the previous version.

Example:

1.1.3.2.1 <X,X,X>.0.<22-50>

1.1.3.2.1 <X,X,X>.0.<21-50>

.

.

1.1.3.2.1 <X,X,X>.**2**.<10-50>

So, the previous version is in a different schema.

We check in the schema change table (Table 9) in the right column (after schema change column) for the entry containing 1.1.3.2.1, and then check for the node number in the corresponding left column for the node number before the schema change.

Before schema change	After schema change
3.2.1<X,X,X>.**1**.<9-9>	1.1.3.2.1<X,X,X>.**2**.<10-50>

Table 9. Schema Change Tracking Table.

This provides us with the node 3.2.1<X,X,X>.1.<9-9> which is the ninth version of the current node. Then, we again trace back to continue finding the previous versions.

3.2.1<X,X,X>.0.<8-9>

.

.

till 3.2.1<X,X,X>.0.<1-9>.

This way, we get the set of all the previous / past versions of the current node with / without schema changes.

Let the current node have the ID 1.1.3.2.1<200009, 200109, uc>.0. <23-50>. From the node ID it is evident that we have node versions up to 50, with the current version as the 23rd. We next find nodes having the following format.

1.1.3.2.1<X,X,X>.0.<24-50>

.

.

.

till 1.1.3.2.1<X,X,X>.0.<50-50>

If there were a schema change, we would notice a node with the schema change bit as 1. We next search in the schema change table an entry for the current version (for example 1.1.3.2.1<X,X,X>.1.<32-32>.

We check for the corresponding node ID after the schema change in the right column, and we check for the entry in the Future Access table (Table 11).

Before schema change	After schema change
1.1.3.2.1<X,X,X>.1.<32-32>.	7.5.2<X,X,X>.2.<33-50>

Table 10. Schema Change Tracking Table.

Node id	Maximum version number (with respect to schema)
1.1.3.2.1	32
7.5.2	50

Table 11. Future Access Table for the current example.

We then learn the maximum version number for the node in the new schema is 50. We continue the search for future versions as follows:

7.5.2<>X,X,X>.0.<34-50>

.

.

till 7.5.2<>X,X,X>.0.<50-50> .

The tt-past-known and tt-future-known axes are similar to the their counterparts tt-part and tt-future, respectively, the only difference being that here the time slice range for comparison is strictly between the creation and read time stamps and not between creation and delete timestamps even if the delete timestamp is defined.

Chapter 8

IMPLEMENTATION

eXist Db Source Code Study

The eXist database provides a free source code for development and enhancements. This project includes various code and test packages used during the current version of the eXist db product. A study of these modules was important to understanding the design and scope of this project and accordingly develop and add our modules to it.

eXist Storage Hierarchy

The eXist Db saves documents into its database in the following hierarchy.

DB->Collections -> Documents.

Thus, a database instance may have one or more collections, and each collection may have one or more documents.

Modules Added

Numbering.ttaxis.item - This package consists of classes that help in creating new item number instances. This package is responsible for the structure and creation of IVTLN objects.

Numbering.ttaxis.item.BerkelyDB -This package contains classes to implement the storage of tt-axis item numbering into Berkeley DB (persistent storage).

Implementation

The eXist db stores documents inside collections. Each collection can have one or more documents. Documents are added using the user interface into the collection.

The following five steps occur upon adding a new or modified document.

1) Document ID is generated for the current document.
2) The document gets parsed using a parser based on Simple API for XML (SAX) parser. During the parsing, the nodes are identified and assigned a type (element, attribute, or text). They are also assigned a NodeId unique to each node.
3) The nodes are stored into the respective index.

   The various types of indexes in eXist are

   - NativeElementIndex: for element type nodes.
   - NativeTextEngine: for text type nodes.
   - NativeValueIndex: for index based on node's value.
   - NativeValueIndexByName: for index based on the node's name.
4) Finally, exist checks if there was a document instance stored previously having the same document ID as the current one. If such a document exists, it is replaced with the current one; else the current one is stored into the repository.
5) Once the document is stored, it is available for querying purposes.

Updating Index in eXist

In eXist, the class named NativeBroker is responsible for creating and updating indexes.

Package: org.exist.storage

Class Name: NativeBroker.java

Description: This is implemented on the basis of the Observer design pattern. Where the 'subject' of observation is this (NativeBroker) class and the 'observers' are the various indexes namely

- NativeElementIndex
- NativeTextEngine
- NativeValueIndex
- NativeValueIndexByName

Once there is some change (transaction causing some change to the storage) in the subject (NativeBroker instance), a notification is sent to the corresponding observer(s) index for modification.

Important Classes

Some of the important classes and their description follow:

- IVTLN.java (Figure 22) - This class holds the information about an item (element). Information stored here includes ID item identifier, TDLN Dewey level number of the item with the timestamp, TIDLN item number of the item with its timestamp, and the VersionTS version of the item with its timestamp.

- Collection.java (Figure 24) - This class represents a collection instance and contains functions to add a document. Documents are stored inside a collection. A collection instance may contain any number of documents.

- DLN.java (Figure 25) - The class that implements the existing (Dewey Level Numbering) plan in eXist.

- Indexer.java - This class implements the document parser for the XML documents. It uses SAX parser.

An item may have one or more instances of TDLN, TIDLN and VersionTS.

The figures in the following pages describe some of the above mentioned classes from a class perspective. Class Diagrams in general present the properties of class i.e. they depict the details about a class's members which are its variables and methods. The class diagrams for the classes Collection.java and DLN.java have been shortened to cover only the essential methods.

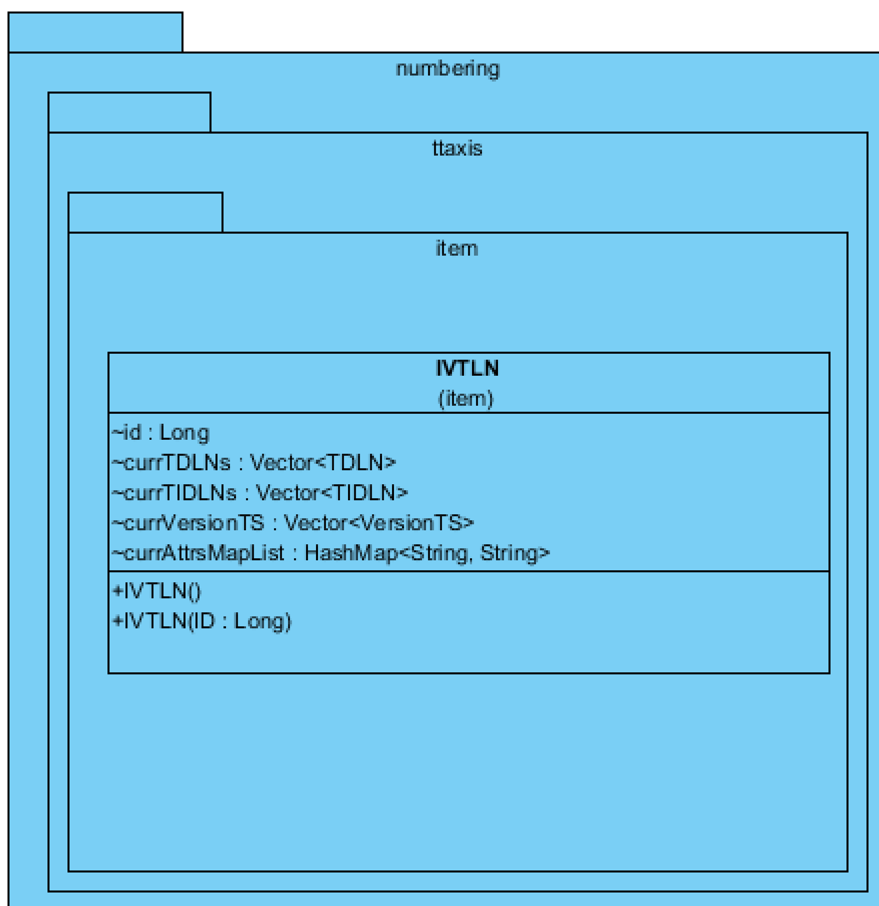


Figure 22. The IVTLN class.

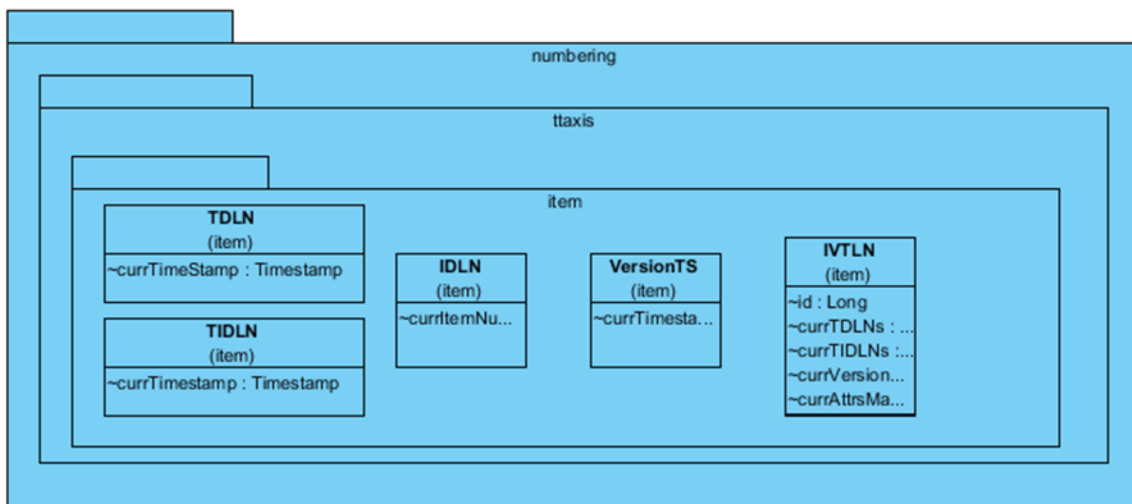


Figure 23. The Item (item) package.

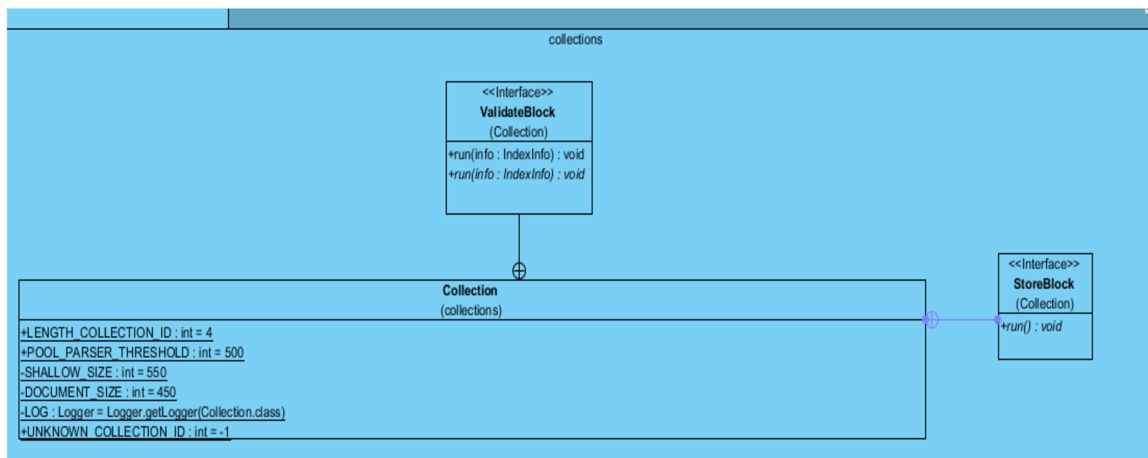


Figure 24. The Collection class.

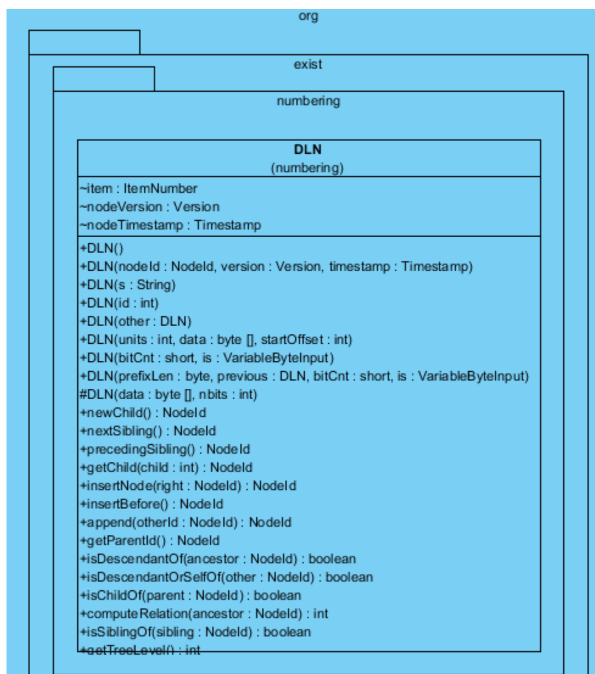


Figure 25. The DLN class in eXist db.

Important Method Sequences

The following subsections describe the important method execution sequences that define our approach for implementation.

Adding a Document Instance to a Collection (Backend). The diagram in Figure 26 illustrates the communication between classes while adding a new document to the collection.

Creation of IVTLN Objects for Items. The process of execution followed during the identification and creation of IVTLN objects for item (elements) is as in Figure 27.

Front-end (GUI) Diagrams. The new plan implementation works only in the backend. The front-end would still remain the same. IVTLN objects are created or updated while a document is added into the collection.

The figures from Figure 28 to Figure 33 show the process of adding a document (bookstore4.xml) to the default collection in an eXist db.

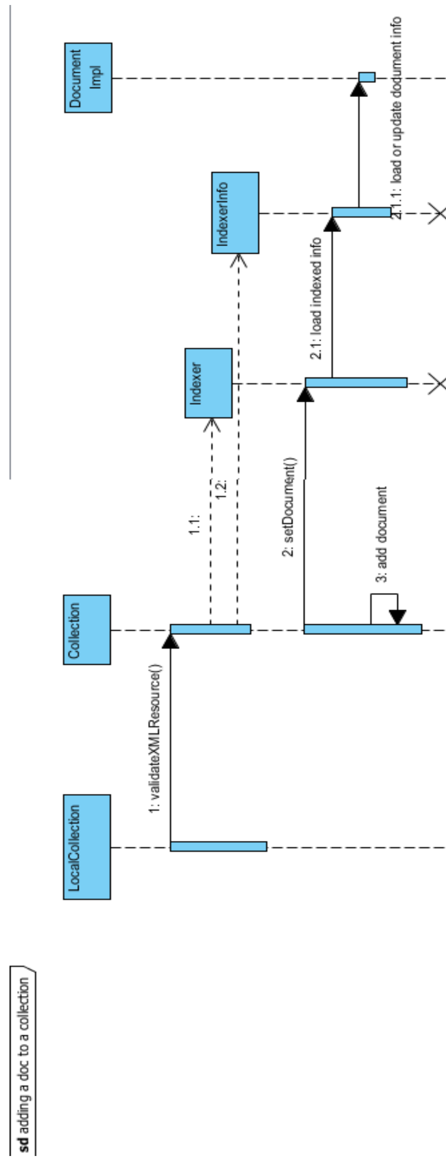


Figure 26. Adding a document to a collection (Sequence Diagram).

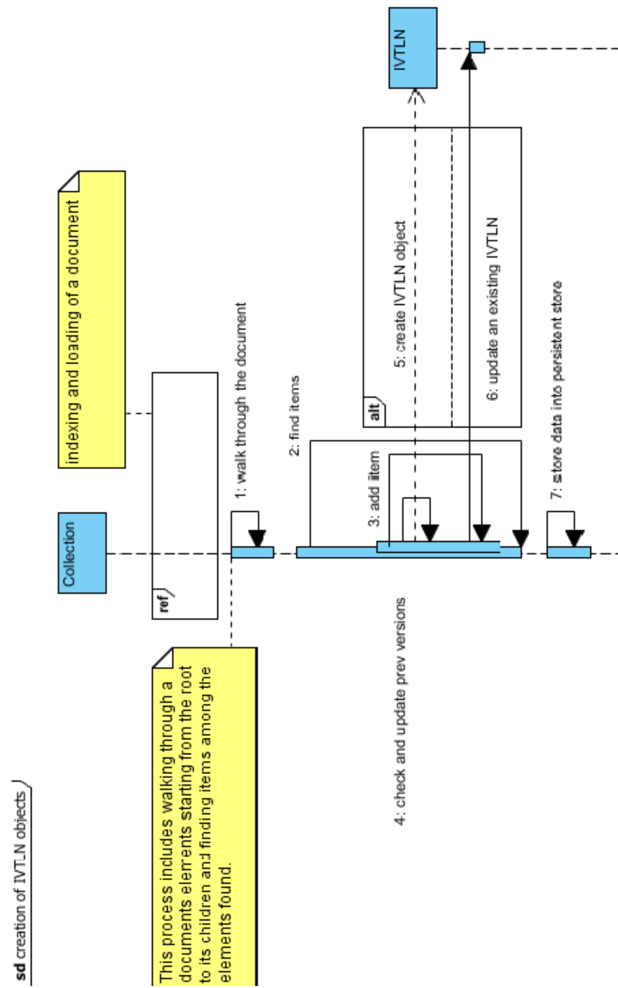


Figure 27. Creation of IVTLN objects (Sequence Diagram).

Figure 28. eXist db user login screen.

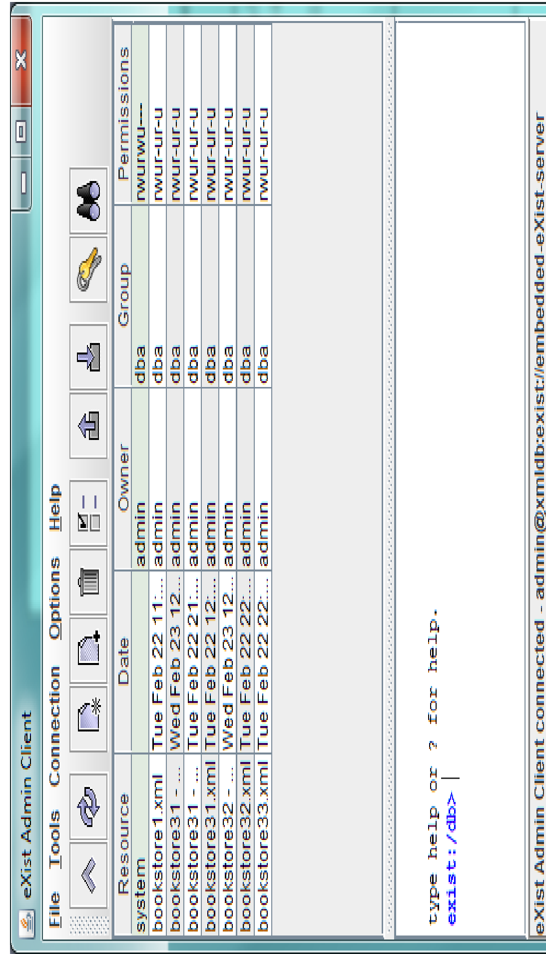


Figure 29. eXist Admin Client Screen.

Upon clicking the menu item “add a document,” the screen shown in Figure 30 appears. This screen allows the user to browse and select a document to add into the collection.

Upon selecting a file the following screen is displayed while the document gets added into the collection (Figure 32).

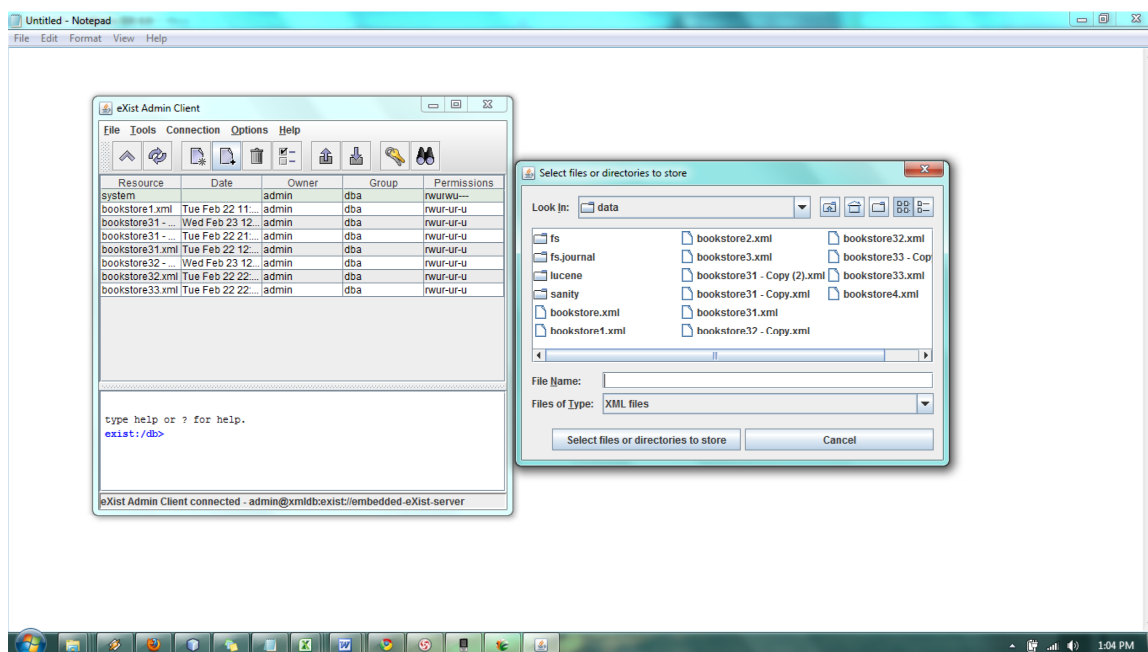


Figure 30. eXist Client (left) and File selection (right) screens.

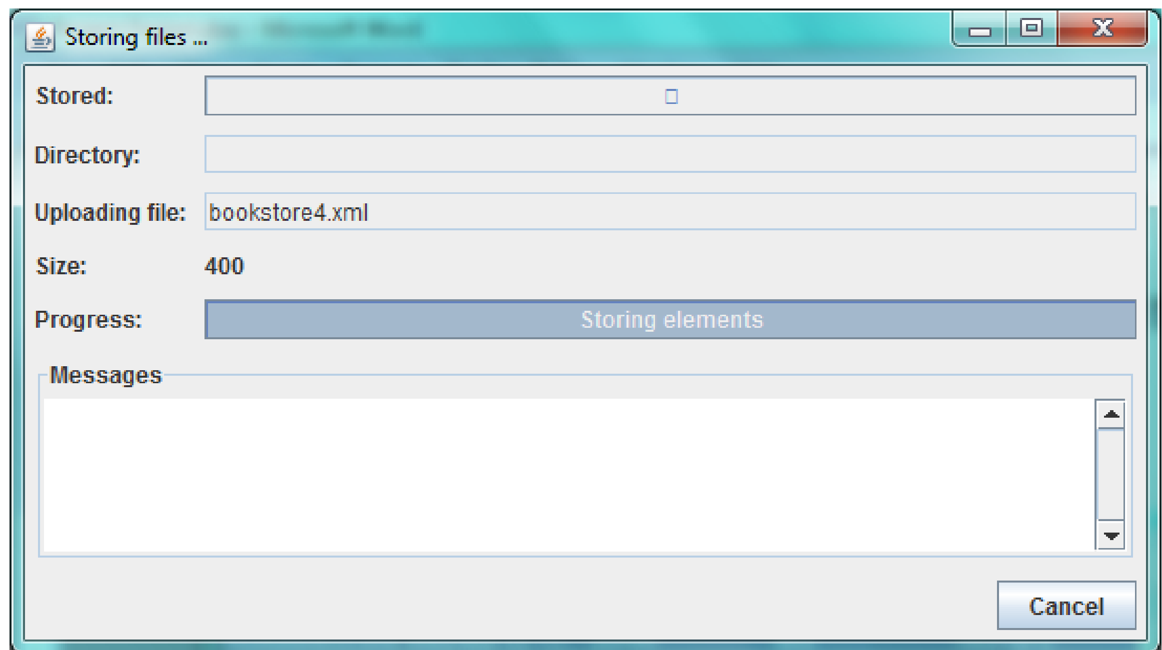


Figure 31. Screen shown while a document is being added.

It is as this point a document gets parsed with the help of SAX parser instance provided in Indexer.java. Once the parsing is completed and the document gets added into the collection, we perform a walkthrough over this document instance.

During this walkthrough, the items (elements) are identified and the corresponding IVTLN objects are created and loaded with version and timestamp information. Once this is done, the screen shown in Figure 33 appears. Notice the newly added document bookstore4.xml in the list (Figure 32).

The new document can be viewed and queried for data. On clicking the document item in the list we may view the document as the shown in Figure 34.

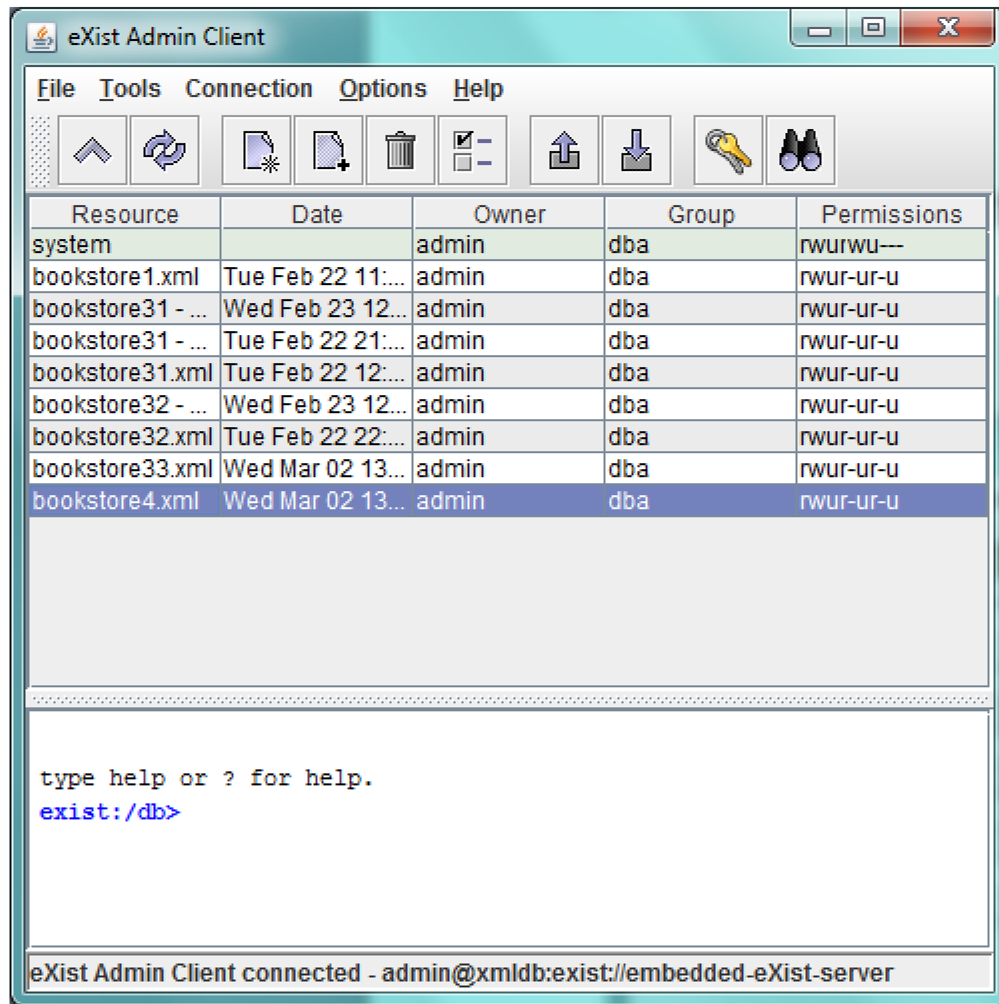


Figure 32. eXist admin client screen (updated with bookstore4.xml).

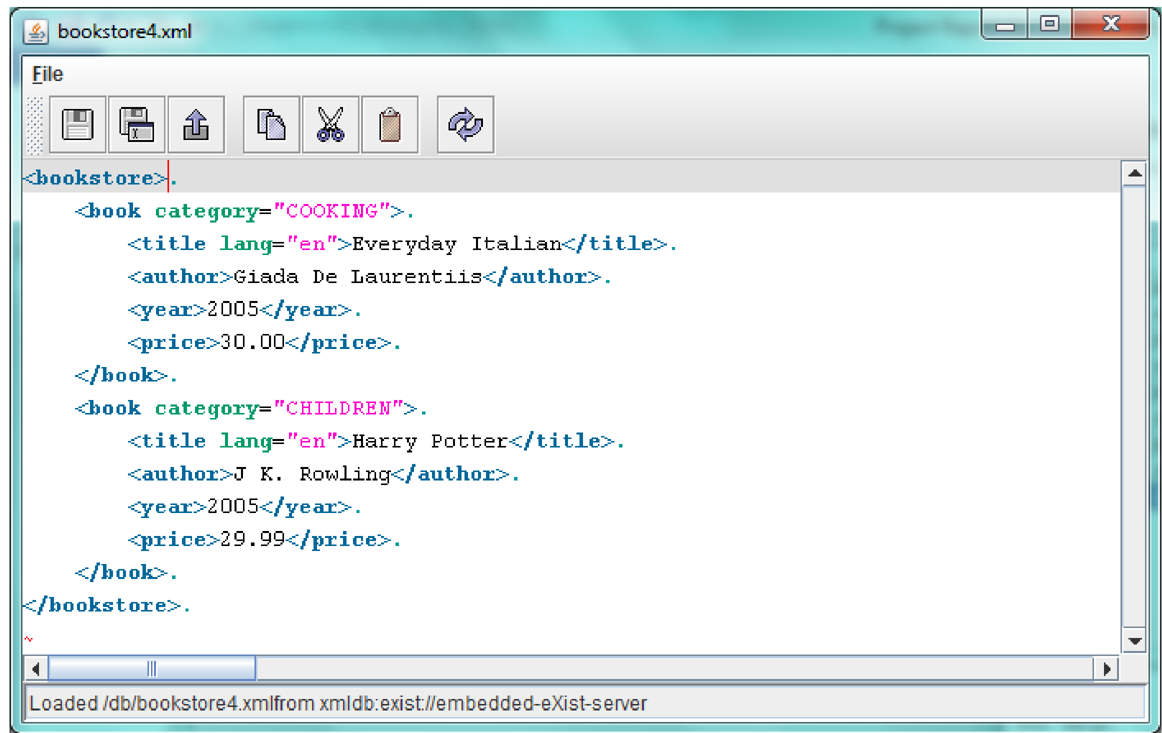


Figure 33. bookstore4.xml.

Chapter 9

MEMORY ANALYSIS

Description

The main trade-off while considering the new numbering plan is between the features added to the additional memory used. We evaluated the following elements from the perspective of memory.

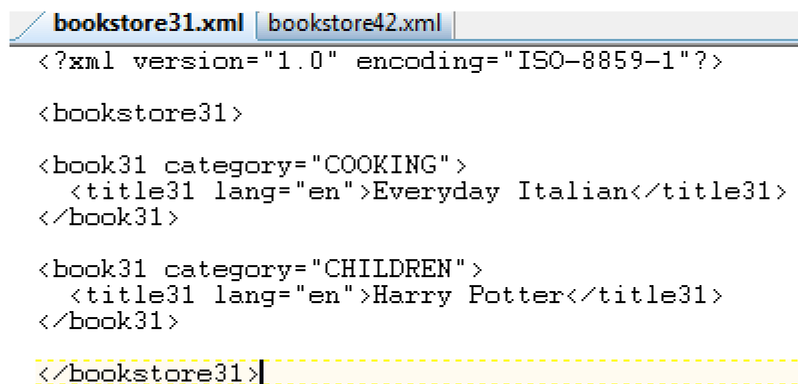
Tool

The tool used here is the Profiler provided by Netbeans IDE.

Memory Perspective

We captured the following memory occupancy of a sample xml file (bookstore31.xml) for analysis.

Bookstore31.xml



The following are the memory occupancy details derived using the Netbeans profiler.

DLN (Objects)

Type: org.exist.numbering.DLN

Memory used (in bytes): 59248

IVTLN and Related Objects

numbering.ttaxis.item.**IVTLN**			616 B	(0%)	11
numbering.ttaxis.item.**ItemNumber**			6,264 B	(0%)	261
numbering.ttaxis.item.**TDLN**			768 B	(0%)	12
numbering.ttaxis.item.**TIDLN**			792 B	(0%)	11
numbering.ttaxis.item.**Timestamp**			8,512 B	(0%)	266
numbering.ttaxis.item.**Version**			6,096 B	(0%)	254
numbering.ttaxis.item.**VersionTS**			384 B	(0%)	12

Type: IVTLN (related classes)

Memory used (in bytes): 23432

As can be seen from the above results, there is an increase in memory usage of about 40% due to the creation and storage of new data. The point to be considered here is the benefits provided by timestamp- and version-based querying to the additional memory occupancy.

Analysis

We now analyze the growth of objects with respect to the memory used across eight different timestamps. The results for the growth in the number of objects and the corresponding memory usage of type DLN and IVTLN related classes can be seen in Figure 34 and Figure 35, respectively.

As can be seen from Figure 34 and Figure 35, there is an increase in the object count and memory used initially due to the addition of the document into the collection and they stay constant after that.

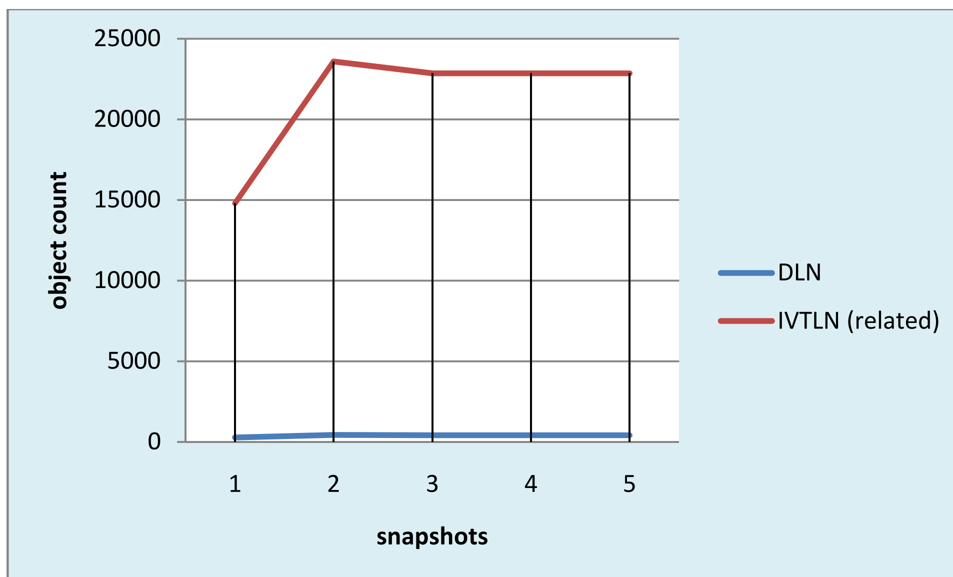


Figure 34. Number of objects across five snapshots.

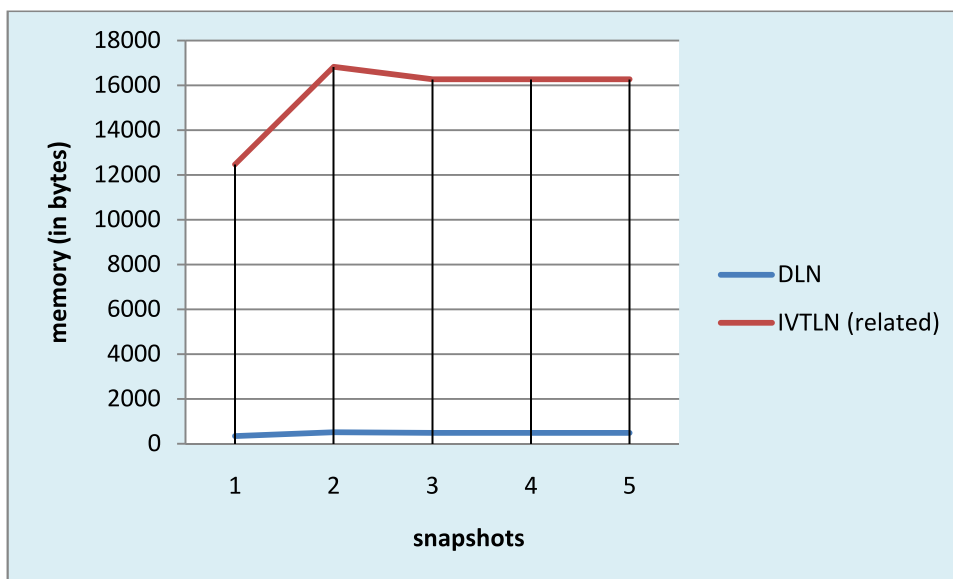


Figure 35. Memory usage (in bytes) across five snapshots.

As can be seen from Figure 36 and Figure 37, the growth in objects and memory usage is linear for both DLN and IVTLN related objects.

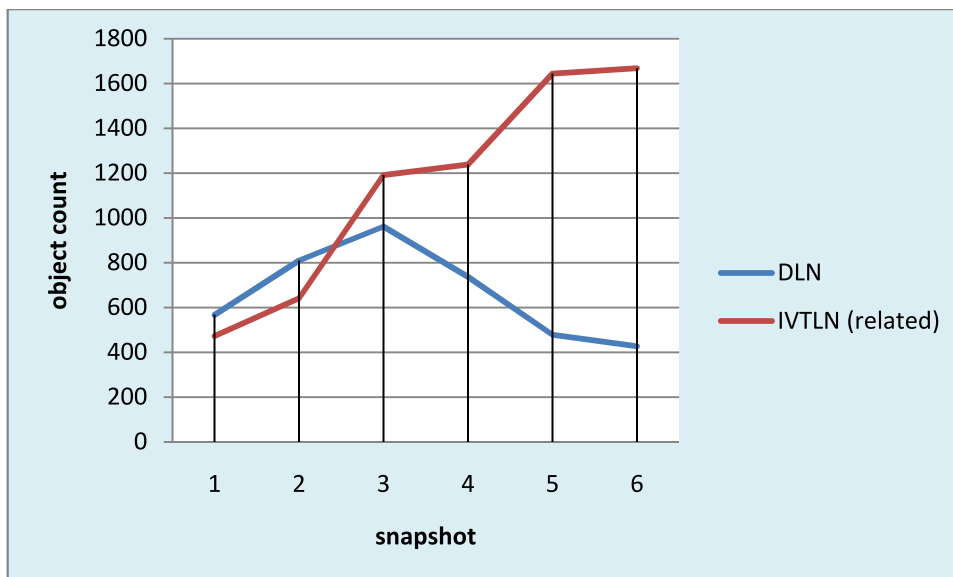


Figure 36. Impact of add, delete, and update node operations on object count.

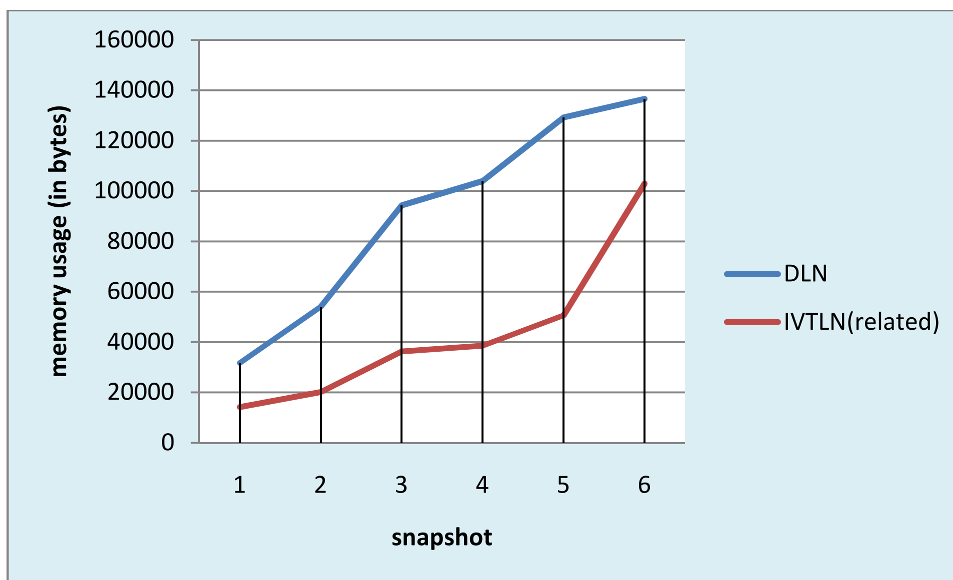


Figure 37. Impact of add, delete, update node operations on memory usage.

Snapshots 1 to 3 represent behavior while adding nodes to the document. Snapshots 3 to 5 represent changes when nodes are deleted from a document. The last snapshot represents the impact of modifying a node in the document.

The growth is the highest when nodes are added. During node deletion, we observe that there is a decline in the count of DLN objects, but there is a rise in IVTLN related objects due to the deletion or modification of existing timestamps and the corresponding creation of new timestamp based objects. During node modification, however, there is not a significant change with respect to the DLN's object count and memory usage. There is some rise in the number of IVTLN objects and the corresponding rise in memory usage due to the creation of objects for the new versions.

Inference

As can be seen from the above results, there is an increase in memory usage of about 40% due to the creation and storage of new data. The tradeoff to be considered here is the features provided by timestamp- and version-based querying to the additional memory occupancy.

Chapter 10

CONCLUSION

This project set out to store and reference data in a time specific manner. There are two primary operations involved in implementing this feature. The first involves storing a node with metadata to include its creation time, deletion, or modification time. The second consists of retrieving the nodes that are filtered based on timestamp and version matching. Both operations require a platform to index nodes based on timestamps (creation and deletion or modification). We call this new index the transaction-time axis (tt-axis).

The tt-axis covers the various versions of a node in a document starting from the earliest to the latest. In addition to the timestamps, the tt-axis also includes version references that provide information about the various versions present for a given node and also for navigating across it. A new numbering plan called item version timestamp level numbering (IVTLN) is introduced to help in the construction of the tt-axis. This new numbering plan extends the current numbering plan Dewey level numbering (DLN) used in eXist db by adding timestamp and version identifiers.

A new strategy called itemization of nodes is introduced wherein the nodes are qualified as items. An item is a node provided with timestamp and version information. In other words, an item is a qualified node provided with the new numbering plan IVTLN. There is just one item instance for all the versions of a node. Therefore, the relation between the various versions of a node to its item is many to one. Currently, itemization of nodes is limited to element nodes, but it may be extended to other node types like text and attribute nodes.

Schema Change is when either a node is moved to a different parent or a node is absent (deleted) in some versions of the parent node. Since such changes need to be tracked and considered while navigating through versions, we introduce a solution using the help of a table known as the Schema Change Tracking table and bits representing schema change in the new numbering plan IVTLN. Also a new table named Future Access table is introduces to minimize the number of updates.

Querying is performed in two phases. The first phase is based on executing XPath expressions, while the second phase takes the result from the first phase and filters the result by comparing the queried time reference with a node version's time stamp range (time between its creation and deletion or modification). A node version is added to the result only if its time stamp range encompasses the requested time slice.

Two new modules were added, namely, numbering.ttaxis.item and numbering.ttaxis.item.berkelyDb, for implementing the IVTLN numbering plan and storing the IVTLN information for document nodes in a persistent manner, respectively. The later has classes to store and update timestamp and version information for documents by creating and providing access to a persistent store.

Our goal is to build an efficient time based querying system and the platform required for it. The current plan (IVTLN) is aimed at replacing the existing numbering plan. i.e., DLN, used in exist db. The trade-off in applying the proposed plan is between the additional features supported to the additional memory utilized by the resources to support the new plan.

Results from the analysis of the memory perspective show that the growth in object count and their memory usage is linear for both IVTLN and DLN. It should be noted that there is more growth in DLN, as currently IVTLN uses DLN. Thus, if IVTLN

were extended to replace DLN by incorporating its features, which is also the wider goal of our project, we would likely see see better results.

The current plan would be of good use in organizations that collect and reference data pertaining over a long period of time, e.g. ,records stored and used in academic institutions, defense studies, etc.

Future Work and Enhancements

The current plan supports itemization of element nodes. This concept of itemization may further be extended to other node types like attribute and text thereby extending the versioning feature to them.

Currently, the implementation does not cover plan change tracking and handling. We could extend our implementation to handle the plan change feature to increase the current plan's scalability.

The current plan could further be extended to include more information so as to reduce number of passes for reading XML document information (xml parsing). There is also some scope for reducing some redundancy (optimization) in data required to execute the tt-axis based functions especially with respect to evaluating tt-axis functions that require a comparison over time or versions.

The current plan can further be applied to other XML-based databases to check for performance and support.

REFERENCES

[1] T. Amagasa, M. Yoshikawa, and S. Uemura. A data model for temporal XML documents. In *DEXA* 2000, 334-344.

[2] T. Bohme and E. Rahm. Supporting efficient streaming and insertion of XML data in RDBMS. In *DIWeb* 2004, 70-81.

[3] P. Buneman, S. Khanna, and W. C. Tan, Why and where: A characterization of data provenance. In ICDT, 2001, 316-330.

[4] P. Buneman et al., Keys for XML. Computer Networks, 2002. 39(5): 473-487.

[5] S. Chien, V. Tsotras, and C. Zaniolo, Efficient schemes for managing multiversion XML documents. VLDB Journal, 2002. 11(4): 332-353.

[6] J. Chomicki, Efficient checking of temporal integrity constraints using bounded history encoding. ACM Transactions on Database Systems, 1995. 20(2): pp. 149-186.

[7] F. Currim, S. Currim, C. Dyreson, R. T. Snodgrass. A tale of two schemas: Creating a temporal XML schema from a snapshot schema with τXSchema. In *EDBT* 2004, 348-365.

[8] C. Dyreson. Observing transaction- time semantics with TTXPath. In WISE, 2001.

[9] C. Dyreson, R.T. Snodgrass, F. Currim, S. Currim. Schema-mediated exchange of temporal XML data. In ER 2006, 212-227.

[10] D. Gao, R. T. Snodgrass. Temporal slicing in the evaluation of XML queries. In *VLDB*, 2003, 632-643.

[11] F. Grandi, An Annotated Bibliography on Temporal and Evolution Aspects in the WorldWideWeb. 2003, TimeCenter Technical Report.

[12] F. Grandi, F. Mandreoli, and P.Tiberio. Temporal modelling and management of normative documents in XML format. Data & Knowledge Engineering, 2005, 54(3): 327-254.

[13] C. S. Jensen and C. Dyreson (editors), A Consensus Glossary of Temporal Database Concepts – February 1998 Version. In *Temporal Databases*: *Research and Practice*, *LNCS 1399*, Springer-Verlag, 1998, 367-405.

[14] C. Li and T. W. Ling. An improved prefix labeling scheme: A binary string approach for dynamic ordered XML. In DASFAA, 2005, 125-137.

[15] A. Marian et al. Change-centric management of versions in an XML warehouse. In VLDB, 2001, 581-590.

[16] S. B. Navathe and R. Ahmed. Temporal relational model and a query Llanguage. Information Sciences, 1989. 49(1): 147-175.

[17] B. Nguyen et al., Monitoring XML data on the web. In SIGMOD, 2001, 437-448.

[18] Oracle Corporation, Application Developer's Guide – Workspace Manager, 10g Release 1, December 2003.

[19] F. Rizzolo and Al. A. Vaisman. Temporal XML: Modeling, indexing and query processing. VLDB Journal, 2007, DOI 10.1007/s00778-007-0058-x.

[20] A.Schmidt, et al. XMark: A benchmark for XML data management. In VLDB 2002, 974-985.

[21] R. Snodgrass, et al. Validating quicksand:Temporal schema versioning in tauXSchema. Data Knowl. Eng. 65(2): 223-242 (2008).

[22] A. Tansel, et al.Temporal databases: Theory, design, and implementation, Benjamin/Cummins Publishing Company, 1993.

[23] J.X. Yu, D. Luo, X. Meng, H. Lu. Dynamically updating XML data: Numbering scheme revisited. World Wide Web 8(1): 5-26 (2005).

[24] Xyleme, A dynamic warehouse for XML data of the web. IEEE Data Engineering Bulletin, 2001. 24(2): p. 40-47.

[25] C. Dyreson and K.G. Mekala, Numbering nodes in temporal XML data instance. Submitted to SSTD 2011, Minneapolis, MN, USA.

[26] K.G. Mekala. Develop a plan to implement ttXpath. Report submitted during Summer 2009, Utah State University, USA.