A Comparative Study of Version Management Schemes for XML Documents

Shu-Yao Chien , Vassilis J. Tsotras , Carlo Zaniolo

September 6, 2000

TR-51

A TIMECENTER Technical Report

Title	A Comparative Study of Version Management Schemes for XML Documents
Author(s)	Shu-Yao Chien , Vassilis J. Tsotras , Carlo Zaniolo
Publication History	August 2000. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Heidi Gregersen, Dieter Pfoser, Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Bongki Moon

Individual participants
Curtis E. Dyreson, Bond University, Australia
Fabio Grandi, University of Bologna, Italy
Nick Kline, Microsoft, USA
Gerhard Knolmayer, Universty of Bern, Switzerland
Thomas Myrach, Universty of Bern, Switzerland
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, University of Alberta, Canada
John F. Roddick, University of South Australia, Australia
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, amazon.com, USA
Andreas Steiner, TimeConsult, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, University of Mons-Hainaut, Belgium
Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: `<http://www.cs.auc.dk/TimeCenter>`

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

The problem of managing multiple versions for XML documents and semistructured data is of significant interest in many DB applications and web-related services. Traditional document version control schemes, such as RCS, suffer from the following two problems. At the logical level, they conceal the structure of the documents by modeling them as sequences of text lines, and storing a document's evolution as a line-edit script. At the physical level, they can incur in severe storage or processing costs because of their inability to trade-off storage with computation. To solve these problems, we propose version management strategies that preserve the structure of the original document, and apply and extend DB techniques to minimize storage and processing costs. Therefore, we propose and compare three schemes for XML version management, namely, the Usefulness-Based Copy Control, the Multiversion B-Tree, and the Partially Persistent List Method. A common characteristic of these schemes is that they cluster data using the notion of *page usefulness*, which by selectively copying current information from obsolete pages provides for fast version reconstruction with minimal storage overhead. The cost and performance of these version management schemes are evaluated and compared through extensive analysis and experimentation.

1 Introduction

XML is becoming a very popular standard for storing and disseminating semistructured information. Since XML documents evolve (updates, new releases) the problem of efficiently versioning XML documents has become of significant interest for content providers, cooperative work, and information systems in general. XML groups have recognized the importance of version support and are working on proposing new standards for the transport level [12]. In this paper, we instead concentrate on the storage level, and propose techniques for (i) storing versioned documents efficiently, and (ii) retrieving any given version with least processing. This is a classical database problem where the objectives are (i) to minimize the number of secondary storage pages needed to store the multiversion document, and (ii) to minimize the number of pages that must be accessed to reconstruct the particular version of the document requested by a user or an application.

Traditional document version management schemes, such as RCS [10] and SCCS [8], are *line-oriented* and have performance problems. For instance, RCS [10] stores the most current version intact while all other revisions are stored as reverse editing scripts. These scripts describe how to go backward in the document's development history. For any version except the current one, extra processing is needed to apply the reverse editing script to generate the old version. Instead of appending version differences at the end like RCS, SCCS [8] interleaves editing operations among the original document/source code and associates a pair of timestamps (version ids) with each document segment specifying the lifespan of that segment. Versions are retrieved from an SCCS file via scanning through the file and retrieving valid segments based on their timestamps.

Both RCS and SCCS may read segments which are no longer valid for the retrieved (target) version, causing additional processing costs. For RCS, the total I/O cost is proportional to the size of the current version plus the size of changes from the retrieved version to the current one. For SCCS, the situation is even worse: the whole version file needs to be read for any version retrieval. To reduce version retrieval cost RCS maintains an index on the valid segments of each version, but still these segments might be stored sparsely among pages generated in different versions, and this lack of clustering can cost many additional page I/Os.

Finally, RCS and SCCS do not preserve the logical structure of the original document; this makes structure-related searches on the XML documents difficult and expensive to support—reconstruction of a whole version might be needed before its component objects can be identified.

Following [2], we take a database-oriented view to the problem of storing, processing and querying efficiently multiple versions of documents. Therefore, we draw on related work on temporal databases and

persistent object managers and model a document as an ordered sequence of objects (e.g., title, section, paragraph, etc.), that can be inserted, deleted or updated as the document evolves through versions. Based on this evolution, each object is assigned an interval that spans over all versions during which the object was valid for the document.

To ensure that all versions can be reconstructed with I/O cost that is proportional to the version's size, we use a clustering scheme called *page usefulness*, which clusters valid objects of a given version in few data pages. When the number of valid objects in a page falls below a threshold, all page objects that are still valid are copied to another page. Therefore, each version can be reconstructed by only accessing useful pages (i.e., filled mostly with objects that are valid for the version). Reconstructing a particular document version is equivalent to finding the valid objects comprising the version and produce them in the correct logical order.

This paper evaluates and compares three XML document versioning schemes starting with an improvement of the method we proposed in [3], called Usefulness-Based Copy Control (UBCC). This method clusters the document objects separately thus avoiding accessing unrelated objects at version reconstruction. The other two schemes, the Partially Persistent List (PPL) and the Multiversion B-Tree (MVBT) [1], view the document evolution as a partially persistent problem. Traditional data structures are $ephemeral$, that is, a single state of the structure is maintained. A data structure is $persistent$ if it can store and access its past states [4]. It is called $partially$ persistent if the structure evolves by applying changes to its "most current" state. In particular, PPL models the document as an ordered linked list of objects which is then made partially persistent. The third scheme uses a partially persistent B+-tree to capture the document evolution.

The remaining of the paper is organized as follows. Section 2 discusses the page usefulness clustering approach. The UBCC scheme is described in section 3, while section 4 describes the partially persistent approaches. Experimental comparisons of all methods as well as the traditional RCS are presented in section 5. Section 6 concludes the paper with open problems and further research.

2 Page Usefulness

For simplicity assume that the document's evolution creates versions with a linear order: V_1, V_2, ..., where version V_i is before version V_{i+1}. Hence a new version is established by applying a number of changes (object insertions, deletions or updates) to the latest version. Each document object is represented in the database by a record that contains the object id (oid), the object attributes (data, text) and a *lifetime* interval of the form: (insertion_version, deletion_version). The insertion_version is filled with the version when the object was added in the document. An object deletion at version V_i is not physical but logical: the deletion_version of the object's record is updated with the version when the deletion took place. This formulation concentrates on object insertions/deletions. An update of object O at version V_j is represented by an "artificial" deletion of the object followed by an "artificial" insertion of the updated object O at the same version V_j. The artificial insertion creates a new database record that shares the same oid O but has a subsequent non-overlapping lifetime interval.

Initially we may assume that objects in the document's very first version are physically stored in pages according to their logical order. After a number of changes, objects of a specific version may be physically scattered around different disk pages. Moreover, a page may store objects from different versions. Hence, when retrieving a specific version, a page access (read) may not be completely "useful". That is, some objects in an accessed page may be invalid for the target version. For example, assume that at version V_1, a document consists of five objects O_1, O_2, O_3, O_4 and O_5 whose records are stored in data page P. Let the size of these objects be 30%, 10%, 20% 25% and 15% of the page size, respectively. Consider the following evolving history for this document: At version V_2, object O_2 is deleted; at V_3, object O_3 is updated; at V_4,

object O_5 is deleted, and at version V_5, object O_1 is deleted.

We define the *usefulness* of a full page P, for a given version V, as the percentage of the page that corresponds to valid objects for V. Hence page P is 100% useful for version V_1. Its usefulness falls to 90% for version V_2, since object O_2 is deleted at V_2. Similarly, P is 70% useful during version V_3. The update of O_3 invalidates its corresponding record in P (a new record for O_3 will be stored in another page since P is full of records). Finally page P falls to 25% usefulness after V_5.

Usefulness influences how well objects of a given version are clustered into pages. High usefulness implies that the objects of a given version are stored in fewer pages, i.e., this version will be reconstructed by accessing fewer pages. Clearly, a page maybe more useful for some versions and less for others. We would like to maintain a minimum page usefulness over all versions (setting this minimum is a performance parameter of our schemes). When a page's usefulness falls below the minimum the currently valid records in this page are copied to another page. This is similar to the "time-split" operation in temporal indexing [11] [7] [9]. Reconstructing a given version is then reduced to accessing only the useful pages for this version; this is very fast since each useful page contains a good part of the requested version. The details of the copying procedure defer for each scheme. However, it can be proved that the overall space used by the database remains linear in the number of changes in the document's version history.

The usefulness as defined above refers to full pages. If we assume that data records are written in pages sequentially, there can be a single page (the last one) that may not be full of records. We can extend the usefulness definition to include such non-full pages as useful by default. This will not affect performance since for each version there will be at most one such page.

3 Usefulness-Based Copy Control

The RCS scheme performs the best when the changes from a version to the next are minimal. For instance, consider a first case, where only 0.1% of the document is changed between versions. Then, reconstructing the 100^{th} version only requires 10% retrieval overhead. But RCS performs poorly when the changes grow larger. For instance, consider a second case, where each new version changes 70% of the document; retrieving the 100^{th} version could cost 70 times retrieving the first one. In this second case, storing complete time-stamped versions is a much better strategy, costing zero overhead in retrieving each version and only a limited (43%) storage overhead. Most real-life situations range between these two cases— with minor revisions and major revisions often mixed in the history of a document. Thus, we need adaptable self-adjusting methods that, in the first case, operate as RCS, while in the second case tend to store complete time-stamped copies. The UBCC scheme described next achieves this desirable behavior by merging the old RCS scheme with an usefulness-based copy control scheme.

Another RCS' problem solved by UBCC is that RCS stores the editing script together with the new value of the revised objects. This increases the size of the script and the reconstruction time. A better approach is to separate the document objects from the editing script. A final improvement introduced in this paper is that usefulness-based techniques are used to cluster both the document objects and the editing script, thus reducing overhead for very long version histories. We will now summarize the UBCC scheme, omitting details that were covered in [3].

Example. Figure 1 shows three versions of a document and how they are stored in UBCC scheme. The first version is stored in pages P1, P2 and P3. We have assumed that the sizes of document objects, Root, CH A, SEC D, SEC E, CH B, SEC F, SEC G, SEC H, CH C, SEC I, SEC J are 50%, 25%, 10%, 15%, 5%, 30%, 35%, 30%, 5%, 10%, and 5% of a data page size, respectively. Assume that we want to maintain a minimum page usefulness of 70%. Then pages P1 and P2 are well above the threshold for version V_1 (P3 is useful by default for version V_1).

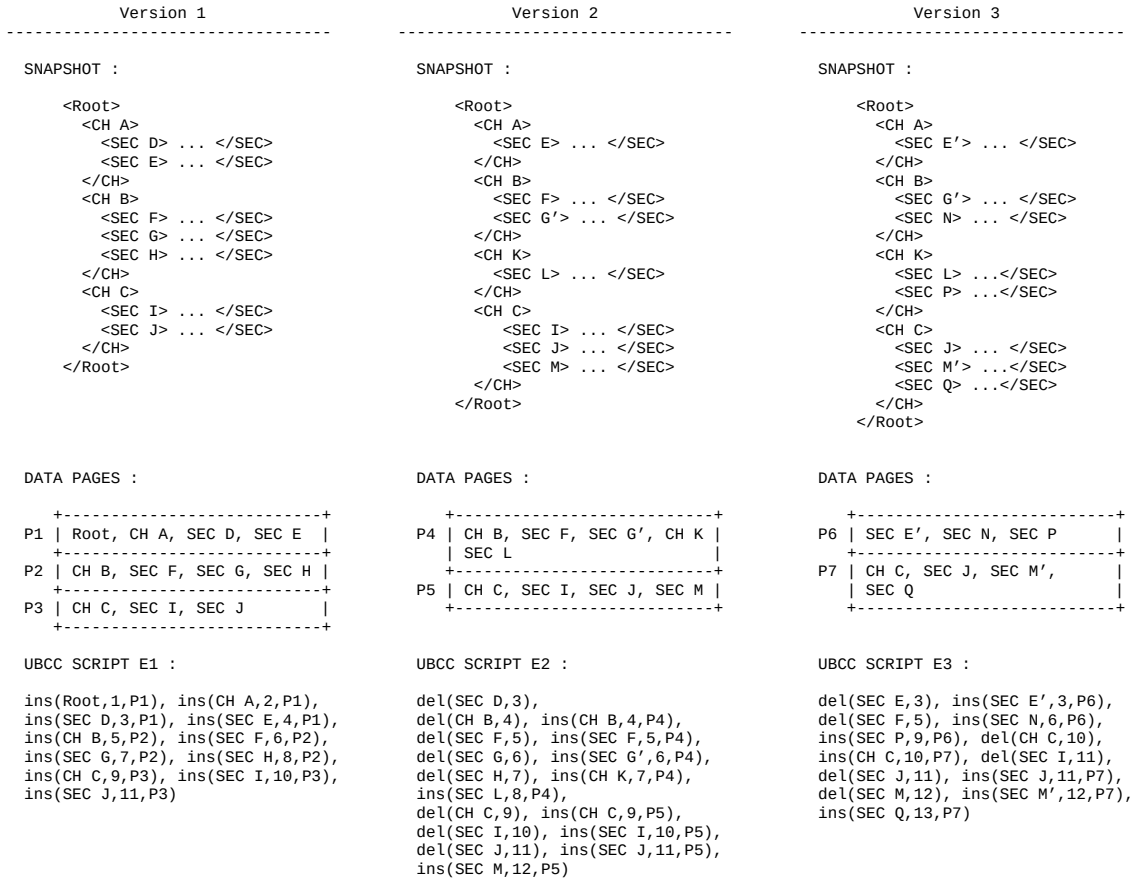

Figure 1: Sample UBCC Versions.

Version V_2 is created by the following changes: *((delete SEC D) , (update SEC G with SEC G') , (delete SEC H) , (insert CH K after CH B) , (insert SEC L after CH K) , (insert SEC M after SEC J))* . The sizes of SEC G', CH K, SEC L and SEC M are 20%, 20%, 25% and 50% of a data page size, respectively. Hence, the logical order of objects in version 2 are: Root, CH A, SEC E, CH B, SEC F, SEC G', CH K, SEC L, CH C, SEC I, SEC J, SEC M. After applying these changes, Page P1 becomes 90% useful (SEC D is not part of version V_2), page P2 becomes 35% useful (since the original SEC G and SEC H are not part of V_2) and page P3 remains 20% useful (no change affected it). Then, pages P2 and P3 are *useless* for the second version and, thus, valid objects in P2 and P3 are copied into a new data page. Copied objects include CH B, SEC F , CH C , SEC I, and SEC J.

After determining which objects need copying, the copied objects are inserted into new pages in their logical order as shown in Figure 1. All new objects and copied objects are stored in page P4 and P5 based on the linear order they appear in Version 2.

UBCC Edit Script. To be able to reconstruct any version, we need to record an *UBCC edit script* for each version. The script for version V_2, E_2, is shown in Figure 1. E_2 is derived from the original edit script as follows:

- Each copied object is treated as a delete operation followed by an insert operation. For example, a delete operation is added for CH B and followed by an insert operation.

- Attach to each operation the position of its target object in the new version. For example, the position of SEC F in the new version is 5.

Notice that, the position for a deleted object is its position in the new version as if it was not deleted. For example, the position of SEC D is 2 in the new version if it was not deleted, so, the delete operation *del(SEC D)* has a position value of 2. These position values are useful for recovering the total order of these objects. Their meaning will be discussed in more detail later.

Version V_3, is generated by the following changes: *((update SEC E with SEC E'),(delete SEC F),(insert SEC N after SEC G'), (insert SEC P after SEC L),(delete SEC I),(update SEC M with SEC M'), (insert SEC Q after SEC M'))*. Here the sizes of SEC E', SEC N, SEC P, SEC M', and SEC Q are 35%,15%,45%, 30% and 29% of the page size, respectively. As a result, pages P1, P4, and P5 become 75%, 70%, and 10% useful, respectively. Thus, valid objects in P5, *CH C, and SEC J*, must be copied. New objects and copied objects are stored into new data pages P6 and P7 in their logical order in Version 3. The *UBCC script*, E_3, for version V_3 is shown in Figure 1. The insertion algorithm is illustrated in [3].

Version Reconstruction. Now, let us discuss how to retrieve a version, say V_i. Since the objects of V_i may be stored in data pages generated in versions V_1, V_2, ..., V_{i-1} and V_i, these objects may not be stored in their logical order. Therefore, the first step is to reconstruct the logical order of V_i objects. The logical order is recovered in a *gap-filling* fashion based on the UBCC script. Let's take the sample version in Figure 1 as an example. We will explain the algorithm by describing how to reconstruct Version 3.

The reconstruction starts by retrieving the first object of Version 3 from its UBCC script, E3. We try to find the first object in the first edit operation. However, we get a *gap* from the operation. The position value of the first operation, *del(SEC E,3)*, is 3. That means, we miss the first two objects and need to *fill the gap* from the previous version, Version 2. Recursively, we start to retrieve the first two objects of Version 2. This retrieval starts from the first operation, *del(SEC D,3)*, of E2. We get a gap again and need to retrieve two objects from the previous version, Version 1. From E1, we find the first two objects of Version 1 and return them to Version 2. Recursively, these two objects are sent back to Version 3. When Version 3 receives these two records, it reads the data page P1 which contains these two objects, Root and CH A, and output them. Page P1 is kept in main memory because it still contains one valid object, SEC E, for Version 3. The reconstruction of Version 3 continues from the previous stop point, *del(SEC D,3)* and the next object is the third object. Since the current operation is a delete, that means its target object is deleted from the previous version. Therefore, Version 3 requests the next object of Version 2 and it is expected to be SEC E. To answer the request for next object, Version 2 needs to retrieve its third object, because its first two objects have been retrieved in the previous run. In a similar manner, Version 2 needs to request one next object from Version 1 because of its *del(SEC D,3)* operation. As expected, the returned record is *ins(SEC D,3,P1)* and it is nullified by the delete operation. However, at this point, the third object of Version 2 has not been retrieved yet. So another next-object request is issued from Version 2 to Version 1 and Version 2 gets back SEC E which is the third object of Version 2. Version 2 sends the *ins(SEC E,4,P1)* record back to Version 3 to reply its next-object request as expected. And this insert record is nullified with the *del(SEC E,3)* record of E3. The search for the third object of Version 3 continues with checking the next edit operation in E3 which is *ins(SEC E',3,P6)*. Then we find the third object because its position value is 3. This gap-filling procedure continues through the script E3 until all objects of Version 3 are retrieved.

Edit Script Snapshots. The above recursive gap-filling algorithm is used to reconstruct any version in the UBCC version file. Reconstructing version V_i may need to involve UBCC script E_1, E_2, ..., E_i, but only useful data pages for each version are read. As a result, the requested version is reconstructed with few page I/Os. However, as the total number of versions grows, the size of total edit scripts will accumulate and, sooner or later, will affect the version retrieval efficiency. To control the overhead of reading edit scripts,

whenever the size of the edit scripts needed for reconstructing a particular version gets over a threshold (e.g. 10% of the size of the version) an *edit script snapshot* is built for this version. The *edit script snapshot* contains one insert record for each object in this version. The edit script $E1$ in Figure 1 is an example of an edit script snapshot. Generating an edit script snapshot will prevent the following versions from back-tracking to edit scripts of earlier versions.

Complexity Analysis. To reconstruct a version, only data pages which are useful for that version need to be read. Take Version 3 as an example. The total order of Version 3 objects is first recovered from edit scripts E3, E2 and E1. The resultant edit operation list is : *ins(Root , 1 , P1), ins(CH A , 2 , P1), ins(SEC E' , 3 , P6), ins(CH B , 4 , P4), ins(SEC G' , 6 , P4), ins(SEC N , 6 , P6), ins(CH K , 7 , P4), ins(SEC L , 8 , P4), ins(SEC P , 9 , P7), ins(CH C , 10 , P7), ins(SEC J , 11 , P7), ins(SEC M' , 12 , P7), ins(SEC Q , 13 , P7)*. Pages pointed by the pointers in these records are read sequentially and objects are retrieved from those pages. Useless pages, such as pages P2, P3, and P5, need not be read because valid objects in those pages have been copied into useful pages. However, consider the way in which the objects of Version 3 are stored. These objects are stored in useful pages generated for Version 3 (page P6 and page P7), Version 2 (page P4) and Version 1 (page P1). Objects generated in the same version are stored in the order in which they appear in the version. Therefore, the retrieving process is actually a merge of these 3 ordered object lists. Merging 3 object lists will involve at most 3 pages at any instance during the process. That means that, with enough memory to hold 3 pages, each useful page of Version 3 needs to be read only once through the whole retrieval process.

The above discussion is applicable to retrieving any version. That is, the I/O cost of reconstructing version V_i , with i pages in memory, is that of

1. reading edit scripts $E_i \cdots E_1$, plus
2. reading useful pages of version V_i.

Let S_i be the size of version V_i (in number of objects) and let B denote the capacity of a page. Clearly, the snapshot of V_i needs S_i/B pages. The number of useful pages created for version V_i is bounded by S_i/BU, where U is the required usefulness.

Database Cost. Let us consider the size of the database first; this is determined by two parts: new objects and copied objects. New objects include first-time inserted new objects and updated objects. Since deleted objects are not removed from storage, deletions do not affect the size of database. Let S_{chg} denote the *total number of changes* in the document evolution (i.e., insertions, updates and deletions). Clearly, the new object part is bounded by $O(S_{chg})$. For the copied object part, the number of objects that got copied once is bounded by $U * S_{chg}$. Objects which got copied twice must be copied from those objects that have already been copied once. Therefore, the total number of objects copied twice is bounded by $U^2 * S_{chg}$. Similarly, the number of objects that got copied i times is bounded by $U^i * S_{chg}$. Collectively, the total number of copied objects is bounded by :

$$\sum_{i=1}^{\infty} U^i * S_{chg} \;=\; S_{chg}/(1-U)$$

Hence the total number of copied objects is $O(S_{chg})$. Combining these two parts, the whole size of the database is linear ($O(S_{chg})$).

Edit Script Cost. The second part of UBCC storage and retrieval costs is due to the edit script. A problem with the original scheme proposed in [3] was that to reconstruct any version, the whole edit script between the start and the current version had to be read. As the number of versions become large the overhead of reading the script can become arbitrarily large; thus we can re-encountered the old RCS problem, albeit scaled-down, because scripts normally require less storage space than the actual objects.

In this paper, we have improved the UBCC policy so that script snapshots are taken at regular intervals, and only the last script snapshot and the changes occurred since then need to be read to reconstruct a new version (this is similar to checkpoints in DB recovery).

Therefore, snapshots based on the current version are recorded in the edit script along with the changes and copy operations. Clearly, too frequent snapshots might cause an excessive use of storage, while too infrequent ones might incur in unacceptable retrieval overhead. Our improved UBCC scheme consists of taking a new snapshot as soon as:

$$\delta E \geq K \times size(V)$$

where,

δE is the increment of the edit script since the last snapshot,

$size(V)$ is the current version size, and

K is a small constant—typically, K is in the order of 0.1.

We can now state the following property:

Proposition *The total size of edit script with snapshots generated by UBCC is linear in the size of the database.*

Proof: The linearity is proved based on two observations. First, the size of the edit script snapshots is linear in the size of the edit scripts. Second, the size of the edit scripts is linear in the size of the database. Now we prove the first observation.

Assume that the an edit script snapshot is generated for version V_m, and the next edit script snapshot is generated for version V_n, where $m < n$ and there is no any edit script snapshot in between. Let T_n denote the snapshot of V_n taken when δE has just surpassed $K \times size(V_n)$.

But a new snapshot is taken whenever

$$K' \times \delta E \geq K \times size(V_n)$$

i.e. as soon as:

$$size(V_n) \leq K'/K \times \delta E$$

with K' denoting the average ratio between the sizes of the representation of changes and the document objects. Therefore, the size of the new snapshot T_n is:

$$size(T_n) = K' \times size(V_n)$$

and, by the above inequality, :

$$size(T_n) \leq (K'^2/K) \times \delta E$$

Summing up the above inequality for all snapshots generated, then the left side is the size of all snapshots and the right side is a constant, K'^2/K, times the size of the whole edit script. Therefore, the size of all snapshots combined is linear in the size of the total size of the edit script.

Next, we are going to prove the total size of the edit script is also linear in the size of the database. Assume that the edit script, E, consists of I insertions, D deletions and U updates and the database is S. Then,

$$size(S) = (I + U) \times S_o$$

where S_o is the average size of objects and,

$$size(E) = (I + U + D) \times S_e$$

where S_e is the average size of edit operation representation, with $S_e/S_o = K'$.

However, the number of deletes must be less than or equal to the number of insertions. That is,

$$D \leq I$$

Therefore, by the above inequality,

$$size(E) \leq (I + U + I) \times S_e < 2 \times (I + U) \times S_e = (2 \times K') \times size(S)$$

So, the size of edit script is linear in the size of database.

Therefore, by the first observation, the total size of edit scripts and snapshots is linear in the size of edit scripts, and by the second observation, is linear in the database size. Ergo, the size of the edit script with snapshots is $O(S_{chg})$. □

Therefore, we conclude that the addition of snapshots adds a small linear overhead to the storage cost. In addition, we can also conclude that $size(T_m)+\delta E$ is linear in the $size(V_m)$ or $size(V_n)$. Thus, taking script snapshots ensures that the retrieval cost for the last snapshot and the edit script remains a small percentage of the retrieval cost for the document.

4 The Partially Persistent Approaches

The logical order of a document relates to the positions of the document's objects. For example, in Figure 1 the first document version has Root in position 1, CH A in position 2, SEC D in position 3, etc. In the second version, SEC E moves up to position 3 and so on. Frequently, when a document is accessed it is requested in its logical order. Hence to store a document, we can utilize a data structure that maintains such an order (for example a B+-tree or an ordered list). Document versioning is then reduced to making this data structure partially persistent.

4.1 Utilizing a Multiversion B-Tree

Assume that the object positions in the first version of a document are directly indexed by a B+-tree. That is, the leaves of this B+-tree contain records with keys 1, 2, 3, etc. where record k stores the object in the $k - th$ position of the document. However, since each object insertion/deletion affects the position number of all the objects after it, updating this B+-tree becomes very inefficient. For example, adding/deleting one object in the beginning of the document would update all the positions after this object. This problem can be resolved if the object positions are encoded in a way not altered by document changes. One simplistic and straightforward solution is to identify object positions by an ordered sequence of large non-consecutive integers. Then a future insertion between positions x and y can be indexed by a number that lies between x and y. For example, assume that the first object in version V_1 is associated with integer 100, the second

object with 200, etc. If in version V_2 an object is added between the first two objects, it can be associated with integer 150 and so on.

The choice of numbers as well as the scheme to associate new numbers for future insertions depends on the document evolution. While at worst this scheme can run out of possible integers (if the number of changes assigned between two positions are more than the difference between the two integers associated with them), we do not expect this to happen in practice especially if large integers are chosen. The advantage of this simple scheme is that the associated integers maintain the logical order of the document while at the same time they can be efficiently indexed by a B+-tree.

Since document changes are provided by object oid (for example, add object with oid O after object with oid Q), each object in the current version must know its associated integer. This is easy to maintain through a hashing scheme. For an object insertion the hashing scheme finds the integer associated with the position of this insertion and the B+-tree is updated. Deletions work similarly. Note that deleted integers can be reused.

There have been various approaches to making a B+-tree partially persistent [1] [7] [13]. In our experiments we used the MVBT since its code was readily available to us.

The MVBT [1] is a directed acyclic graph that "embeds" many B+-trees. It has a number of root nodes, where each root provides access to subsequent versions of the ephemeral B+-tree's evolution. Like all temporal access methods, it appends data records with lifetime intervals of the form (*insertion-version*, *deletion-version*). Records are clustered together in pages based on their indexing attribute values (key space) and their lifetime interval (version space). Index records are appended with lifetime intervals as well.

With the exception of root pages, a page is "useful" as long as it has at least D valid records (D is less than B, the page capacity). Inserting or deleting an object at version V_i is performed by first searching the MVBT for the target leaf page where this change is to be applied. This search is similar as in an ephemeral B+-tree but it also takes into account the lifetime intervals of index records (so that the page that is valid for V_i is reached). A change is called *non-structural* if it is handled within an existing page. A *structural* change creates at least one new page.

For an object insertion, if the target leaf page is not full a new record is inserted in that page and the insertion is completed. Since the deletion-version of the inserted object is yet unknown, its record's lifetime interval is initialized as (V_i, now) where now is a variable representing the ever increasing current version number. If the target leaf page already has B records a *page overflow* is detected. For an object deletion, the data record for the deleted object is identified in the target leaf page. If the number of valid records in this page is greater than D then the record's deletion-version is updated from now to V_i and the deletion is completed. However, if the deletion causes the leaf page to have less than D valid objects a *weak version underflow* is detected [1].

Page overflow and weak version underflow are structural changes and need special handling. More specifically, a version-split is performed on the target leaf-page. This is similar to the time-split of [7] or the page copying of [11]. The version-split on a page P at version V_i, is performed by copying to a new page R the records valid in page P at V_i. Page P is considered non-useful after version V_i.

The resulting new page has to be incorporated in the structure. Briefly, there are three cases for handling the new page R. First, if the number of records in R is between $D+e$ and $B-e$ (where e is a predetermined constant), page R is directly inserted in the MVBT. Constant e works as a buffer that guarantees that a structural change to the new page R can happen only after at least e new changes. The page insertion is carried out by accessing the parent page of page P, marking the index record pointing to page P as deleted at version V_i, and then inserting a new index record pointing to the new page R. Even though these changes occur in an index page, they are similar to insertion and deletion of data records in a leaf-page and are handled identically. Thus a change can propagate upwards until a root is reached. The second case is when the resulting page R has more records than the specified range; this is called a *strong version overflow* and

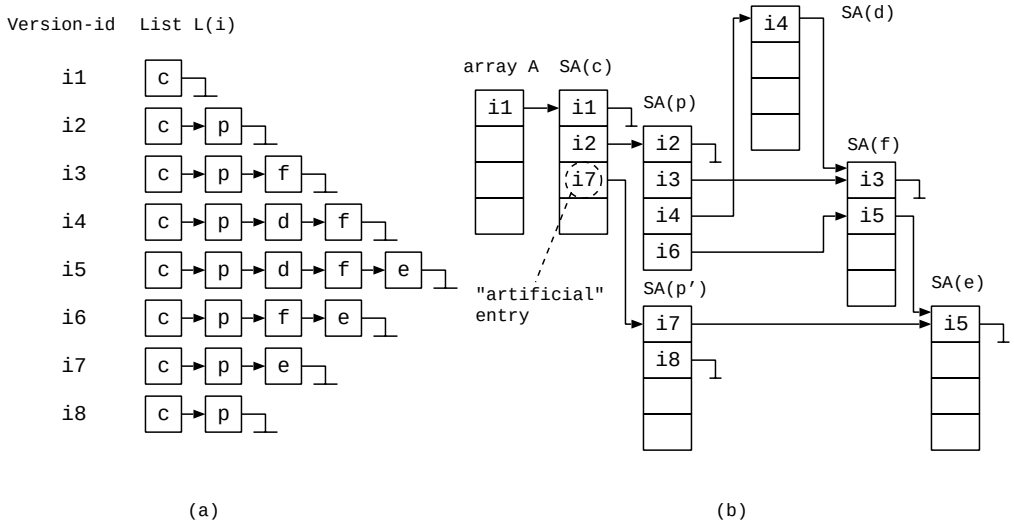

Figure 2: A Partially Persistent List example.

is handled by splitting R into two pages using a key-split. A key-split simply divides the records of R using their key attribute value (this is similar to the traditional page split in a B-tree). The third case is if page R has less records than the specified range. This is called a *strong version underflow* and is handled by merging R with another "sibling" page.

The space used by the above splitting/merging policies is still linear to the total number of changes S_{chg} in the document's evolution. If version V_i of the B+-tree had S_i objects, then the MVBT reconstructs it with $O(log(S_{chg}/B) + S_i/B)$ I/O's.

4.2 The Partially Persistent List

Various notions of partially persistent lists have appeared in the temporal database literature. Our discussion follows the approach outlined in [6] on how to make an ordered list partially persistent. In [5] a scheme to support non-ordered partially persistent lists is presented. [13] presents the C-list, which is a list structure made up of a collection of pages that contain versions of data records clustered by oid. However, a C-list solves a different query: "given an oid and a version interval, find all versions of this oid during this interval".

Let L be an ephemeral list of elements. We assume that there is a pointer to the top element of the list and that each element has a next pointer to its right sibling in the list. We are interested in maintaining the relative positions of the elements in the list starting from the top of the list. Inserting or deleting an element from L corresponds to finding the position of this element in L and performing the update. Let $L(i)$ be the sequence of elements the list had at version V_i.

Our aim is to reconstruct $L(i)$ by accessing only pages that were "useful" during version V_i. This can be achieved if we maintain the list of useful pages. Assume that the very first version of L is stored sequentially into pages. As deletions arrive, some of these pages will become non-useful and thus have to be copied. However, this copying needs to maintain the list logical order, i.e., the relative positions of the list elements. Moreover, since insertions can happen anywhere in the list, some space is needed inside each page for future insertions. Both problems are solved if we use the splitting/merging policies of the MVBT [1]. As before, a hashing scheme can be used to identify the position of a given oid in the current version of the list.

Since list reconstruction starts from the top element in the list, the first page of L must be identified for any given version. This is easily achieved by an array A, which keeps pointers to the first pages that list L

ever had, indexed by the version id. If the first page in the list changes at version V_j (for example this page became non-useful), this array is updated by a record of the form: $< V_j, pointer >$, where $pointer$ points to the new first page. After the first list page at a given version is found, the second page must be found and so on. This is performed by keeping a similar array $SA(P)$ for each list page P. Array $SA(P)$ keeps records of the form $< version, pointer >$ whenever the next page of P changed. However, if the next page in front of page P changes very frequently, array $SA(P)$ can become very large. This affects the list reconstruction, since at worst a logarithmic search would be needed for each SA array in the list. To solve this problem, we allow an SA array to have up to C entries (C is a constant greater than 1). If the next page after a page P changes more than C times while P is a useful page, then P becomes "artificially" useless (even if it still has enough valid records). A new page P' is created that copies all valid records of P, but accompanied with an empty $SA(P')$ array. Page P' replaces P in the list of useful pages. An example appears in Figure 2.

In practice, array $SA(P)$ can be implemented as part of page P. This limits the number of data records that a page can hold, but it allows for fast reconstruction since the next page can be found without further I/O's. It can be shown that this technique still maintains linear space. Moreover, version V_i is reconstructed with $O(log(S_{chg}/B) + S_i/B)$ I/O's.

5 Performance Analysis

We compare the performance of the three usefulness-based XML document version management schemes (namely UBCC, Partially Persistent List, and Multiversion B-Tree) as well as the basic RCS approach. As a baseline case we also report the performance of a "Snapshot" scheme, that simply keeps a copy of each document version. For each method we observed the version retrieval cost and the space consumption. The page size is set to 4K bytes.

We first compare the behavior of all schemes under the same usefulness requirement. For this experiment we used a document evolution with the following characteristics:

- the size of each version is approximately 100 pages;
- each version changes about 20% from the previous version (half of the changes are insertions and the other half are deletions);
- changes are uniformly and randomly distributed among data pages;
- the usefulness requirement is 50%;
- the document evolution had a total of 100 versions.

Figure 3 shows the version retrieval cost measured as the number of page I/O's needed to reconstruct a version. The "Snapshot" scheme clearly has the minimal version retrieval cost, since each version is already stored in its entirety on disk. As expected, all usefulness-based schemes have version retrieval cost that is proportional to the size of the reconstructed version. (In this experiment, the average version size remains the same –about 100 pages–, so the retrieval cost of the three usefulness-based schemes and the Snapshot scheme are approximately parallel to the horizontal axis). The retrieval overhead against the Snapshot scheme is because a useful page includes some non-valid objects. Thus UBCC, MVBT and PPL have to access more pages than the Snapshot scheme. However, this overhead is constant. In particular, the UBCC has the best retrieval performance among the three usefulness-based schemes. The MVBT is slightly better than the PPL because PPL uses some page capacity for the SA arrays. Hence, the answer is distributed among more pages, increasing the retrieval cost. The RCS strategy needs to read the whole database prior

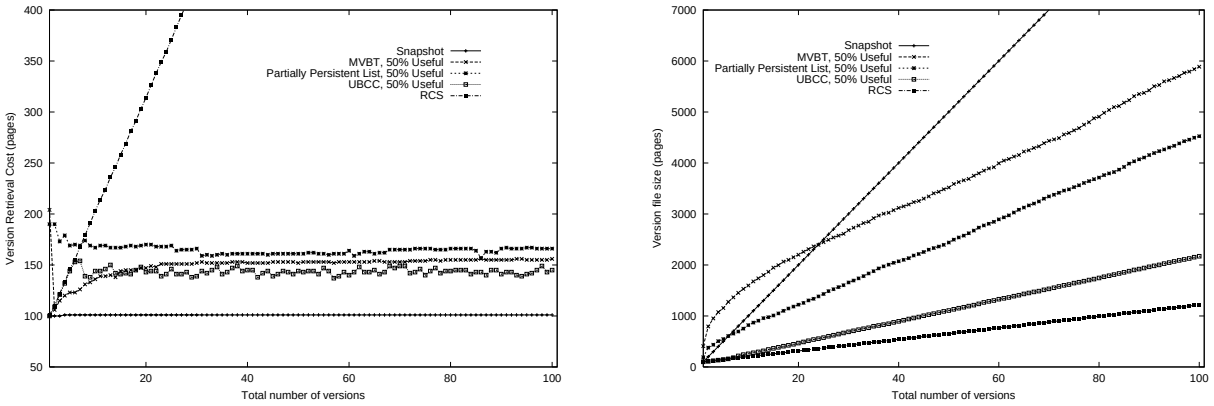

Figure 3: Version retrieval and storage cost with 50% usefulness requirement.

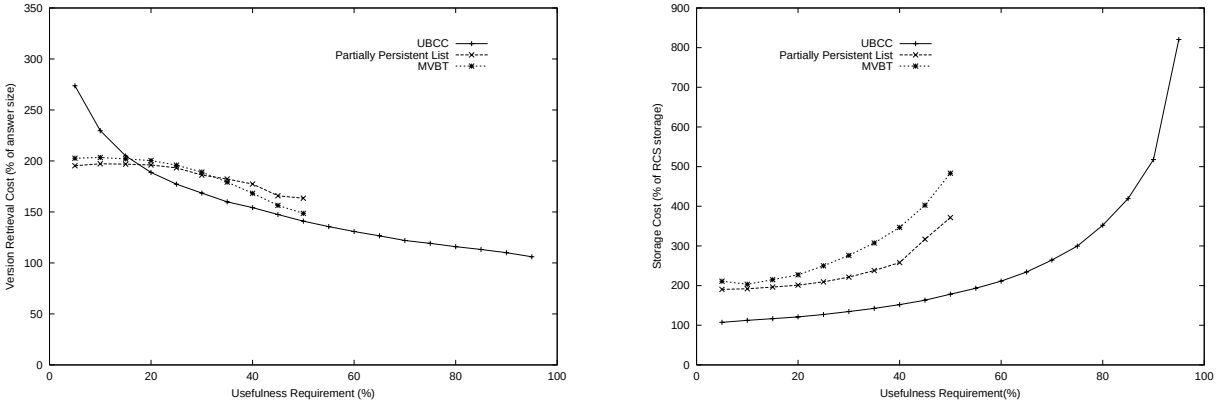

Figure 4: Version retrieval and storage cost v.s. Usefulness

to the target version. Therefore, retrieving later versions gets more expensive than earlier versions. For example, the I/O cost of retrieving the 20^{th} version is 320% of this version's size.

While the Snapshot provided the minimal retrieval cost, its storage cost is too expensive. (at worst it's quadratic). RCS has now the minimal storage cost since it stores only the changes. The space of all usefulness-based schemes grows linearly with the number of changes (which increases with the number of versions). However, each scheme increases at a different rate. In particular, the partially persistent schemes (PPL, MVBT) use more space than UBCC. This is because in the partially persistent schemes copies are triggered either by insertions or by deletions. In contrast, in the UBCC scheme, copies are triggered by deletion operations only. Thus the copying in UBCC is less. Moreover, the MVBT and PPL schemes "reserve" some empty space in a new page for future insertions. This space may remain unused, resulting in higher storage cost.

The effect of usefulness. To study the effect of the usefulness parameter we run the same experiment as above, but for different U's. The experimental results are illustrated in Figure 4. The performance of the PPL and MVBT schemes is depicted until U = 50% since this is the highest usefulness they can achieve. In contrast, the UBCC scheme can achieve higher U's. The version retrieval cost is depicted as the percentage of the answer size. For example, retrieval cost of 140% means that the scheme accessed 40% more pages than the size of the reconstructed version. Clearly, as the usefulness increases, a given version is stored in smaller number of pages (since a page can hold more valid records) and the retrieval cost decreases.

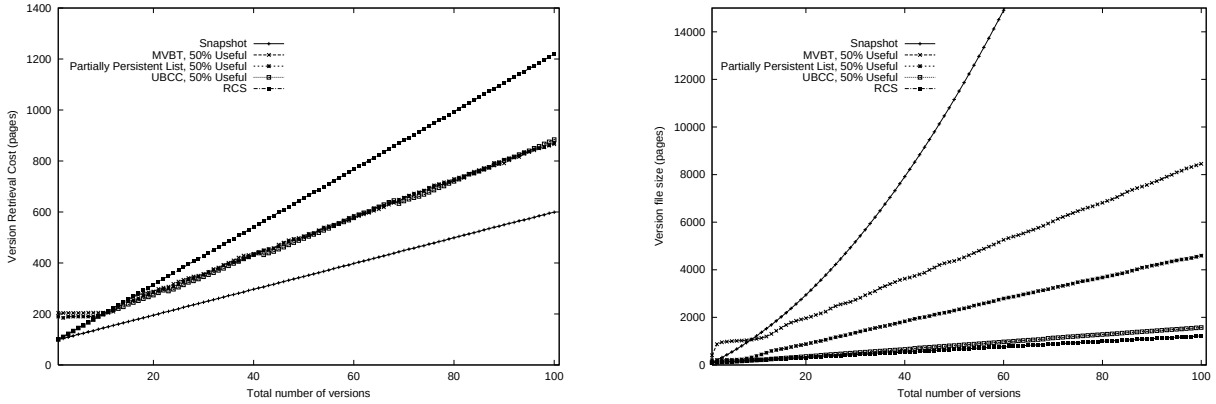

Figure 5: Version retrieval and storage cost with increasing document size.

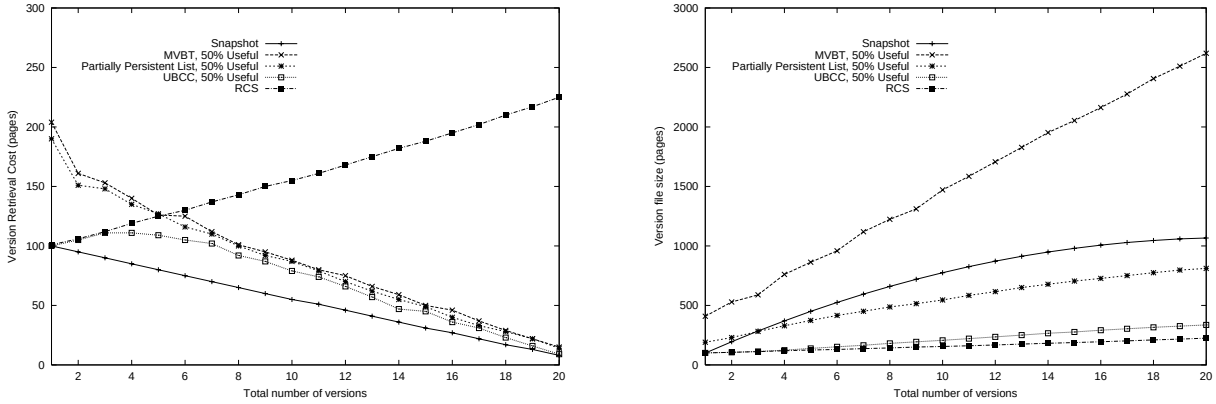

Figure 6: Version retrieval and storage cost with decreasing document size.

Another interesting observation has to do with the behavior at very small U's. Note that the UBCC scheme fills up a new page with records without reserving space for future insertions in this page. As a result, the usefulness of a UBCC page can only decrease due to record deletions. A small U implies that a page will be considered useful even if it has very few valid records. For UBCC this means that many pages may have low usefulness because of deletions. Since these pages are still useful, they are not copied. As a result the answer will be clustered in many UBCC pages, thus increasing the retrieval cost. In contrast, fewer pages in the PPL and MVBT schemes will reach small usefulness since new insertions in the reserved space will increase usefulness.

As expected, when the usefulness increases, the space of all methods increases, too. Figure 4 depicts the storage cost as a percentage over the (minimal) RCS storage. Higher U's imply that the copying threshold will be reached faster and thus more copies are made.

Limited Resources. Setting the usefulness parameter serves as an optimization tool for each of the three schemes. For example, consider the case of limited system resources (storage). That is, a version management system wants to improve its version retrieval performance, but it has only 200% extra free space. According to Figure 4, for that space requirement, the MVBT scheme can guarantee 35% usefulness, PPL 45% usefulness, while the UBCC 75% usefulness. Choosing higher usefulness (UBCC) is definitely preferable since the retrieval time will be better.

Increasing and Decreasing Document Sizes. Our next experiments examine the cases when a document follows an increasing or decreasing size evolution. We first examine an evolution where the document increases by 5% at each version. At each version there are 10% insertions and 5% deletions. The usefulness requirement is again 50%. The results are shown in Figure 5. All usefulness-based schemes have very close version retrieval performance that is proportional to the version size. The UBCC storage cost is very close to the minimal storage of RCS, because the small deletion percentage rarely causes UBCC to copy useless pages. The MVBT and PPL both use more space than UBCC because insertions trigger more copies. The quadratic space of the Snapshot scheme is also observed.

Figure 6 depicts the performance for a document that decreases in size as it evolves. Here the document size decreases by 5% per version. The version retrieval cost of all three schemes decreases as the version size decreases. However, the RCS scheme retrieval grows linearly with the number of changes. Regarding storage cost, the UBCC is again the closest to the minimal storage (RCS).

From the above experimental results, we observe that the usefulness based schemes provide version retrieval cost that is proportional to the the size of target version at the expense of some extra space. The extra space is linear to the total number of changes. The UBCC scheme consumes much less extra space than MVBT and PPL at all levels of usefulness requirement. In particular, the performance of the UBCC can be easily tuned through the usefulness parameter to the appropriate level of performance (depending on the characteristics of the document evolution).

6 Conclusions

In this paper, we investigated several techniques to manage evolving structured documents, while preserving intact its logical structure. We investigate the $UBCC$ scheme that combines RCS version control with the usefulness-based page management, and two other schemes, the multiversion B-trees and partially persistent lists, that are extensions of techniques used in temporal databases and persistent object stores.

We found that the $UBCC$ technique, originally presented in [3] and here improved with respect to script management, is more efficient than the other two, in terms of I/O, and requires minimal storage overhead.

We are currently investigating related issues, including querying and restructuring of versioned documents, and the use of our versioning techniques in supporting cooperative document authoring.

7 Acknowledgments

We would like to thank Bernhard Seeger for kindly providing us the MVBT code.

References

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, *"On Optimal Multiversion Access Structures"*, Proceedings of Symposium on Large Spatial Databases, Vol 692, 1993, pp. 123-141.

[2] Panel Discussion, M.J. Carey (moderator) *Of XML and Databases: Where's the Beef?*, ACM SIGMOD Conference, 2000.

[3] S-Y. Chien, V.J. Tsotras, and C. Zaniolo, *Version Management of XML Documents*, WebDB 2000 Workshop, Dallas, TX, 2000.

[4] J.R. Driscoll, N. Sarnak, D. Sleator and R.E. Tarjan, *Making Data Structures Persistent*, Journal of Comp. and Syst. Sciences, Vol 38, pp 86-124, 1989.

[5] G. Kollios and V.J. Tsotras, *Hashing Methods for Temporal Data*, under revision, IEEE Trans. on Knowledge and Data Engineering, 1999.

[6] A. Kumar, V. J. Tsotras, C. Faloutsos, *Access Methods for Bi-Temporal Databases*, IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 1, 1998, pp 1-20.

[7] D. Lomet and B. Salzberg, *Access Methods for Multiversion Data*, ACM SIGMOD Conference, pp: 315-324, 1989.

[8] Marc J. Rochkind, *The Source Code Control System*, IEEE Transactions on Software Engineering, SE-1, 4, Dec. 1975, pp. 364-370.

[9] B. Salzberg and V.J. Tsotras, *A Comparison of Access Methods for Time-Evolving Data*, ACM Computing Surveys, Vol. 31, No. 2, pp: 158-221, 1999.

[10] Walter F. Tichy, *RCS–A System for Version Control*, Software–Practice & Experience 15, 7, July 1985, pp. 637-654.

[11] V.J. Tsotras, N. Kangelaris, *"The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries"*, Information Systems, An International Journal, Vol. 20, No. 3, 1995.

[12] WWW Distributed Authoring and Versioning (webdav). *http://www.ietf.org/html.charters/webdav-charter.html*

[13] P.J. Varman and R.M. Verma, *An Efficient Multiversion Access Structure*, IEEE Trans. on Knowledge and Data Engineering, Vol.9, No. 3, pp: 391-409, 1997.